KTH ROYAL INSTITUTE
OF TECHNOLOGY

Doctoral Thesis in Computer Science

# Debloating Java Dependencies

CÉSAR SOTO VALERO

Stockholm, Sweden 2023

# Debloating Java Dependencies

CÉSAR SOTO VALERO

Academic Dissertation which, with due permission of the KTH Royal Institute of Technology, is submitted for public defence for the Degree of Doctor of Philosophy on Thursday the 1st June 2023, at 13:15 p.m. in D2, Lindstedtsvägen 9, Stockholm.

Doctoral Thesis in Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden 2023

© César Soto Valero

ISBN 978-91-8040-557-7
TRITA-EECS-AVL-2023:36

Printed by: Universitetsservice US-AB, Sweden 2023

**Abstract**

Software systems have a natural tendency to grow in size and complexity. A part of this growth comes with the addition of new features or bug fixes, while another part is due to useless code that accumulates over time. This phenomenon, known as “software bloat,” increases with the practice of reusing software dependencies, which has exceeded the capacity of human developers to efficiently manage them. Software bloat in third-party dependencies presents a multifaceted challenge for application development, encompassing issues of security, performance, and maintenance. To address these issues, researchers have developed software debloating techniques that automatically remove unnecessary code.

Despite significant progress has been made in the realm of software debloating, the pervasive issue of dependency bloat warrants special attention. In this thesis, we contribute to the field of software debloating by proposing novel techniques specifically targeting dependencies in the Java ecosystem.

First, we investigate the growth of completely unused software dependencies, which we call “bloated dependencies.” We propose a technique to automatically detect and remove bloated dependencies in Java projects built with MAVEN. We empirically study the usage status of dependencies in the Maven Central repository and remove bloated dependencies in mature Java projects. We demonstrate that once a bloated dependency is detected, it can be safely removed as its future usage is unlikely.

Second, we focus on dependencies that are only partially used. We introduce a technique to specialize these dependencies in Java projects based on their actual usage. Our approach systematically identifies the subset of functionalities within each dependency that is sufficient to build the project and removes the rest. We demonstrate that our dependency specialization approach can halve the project classes to dependency classes ratio.

Last, we assess the impact of debloating projects with respect to client applications that reuse them. We present a novel coverage-based debloating technique that determines which class members in Java libraries and their dependencies are necessary for their clients. Our debloating technique effectively decreases the size of debloated libraries while preserving the essential functionalities required to successfully build their clients.

**Keywords:** Software debloating, software dependencies, Java bytecode, package manager, static program analysis, dynamic program analysis

**Sammanfattning**

Mjukvarusystem har en naturlig tendens att växa i storlek och komplexitet. En del av denna tillväxt kommer med tillägget av nya funktioner eller buggfixar, medan en annan del beror på onödig kod som ackumuleras över tiden. Detta fenomen, känt som mjukvaru-bloat, ökar med praxis att återanvända mjukvarubibliotek, vilket har överstigit kapaciteten hos mänskliga utvecklare att effektivt hantera dem. Mjukvaru-bloat i tredjepartsbibliotek innebär en mångfacetterad utmaning för applikationsutveckling, som omfattar säkerhets-, prestanda- och underhållsproblem. För att hantera dessa problem har forskare utvecklat mjukvaruavbloatningstekniker som automatiskt tar bort onödig kod.

Trots att betydande framsteg har gjorts inom området för mjukvaruavbloatning, kräver det genomgripande problemet med bloat bland kodberoenden särskild uppmärksamhet. I denna avhandling bidrar vi till området för mjukvaruavbloatning genom att föreslå nya tekniker som specifikt riktar sig mot beroenden i Java-ekosystemet.

Först undersöker vi tillväxten av helt oanvända mjukvaruberoenden, som vi kallar överflödiga (bloated) beroenden. Vi föreslår en teknik för att automatiskt upptäcka och ta bort svullna beroenden i Java-projekt som byggs med Maven. Vi studerar empiriskt användningsstatus för beroenden i Maven Central Repository och tar bort överflödiga beroenden i mogna Java-projekt. Vi visar att när ett överflödigt beroende upptäcks kan det säkert tas bort eftersom det är osannolikt att det kommer att användas i framtiden.

För det andra fokuserar vi på beroenden som endast används delvis. Vi introducerar en teknik för att specialisera dessa beroenden i Java-projekt baserat på deras faktiska användning. Vår strategi identifierar systematiskt den delmängd av funktioner inom varje beroende som är tillräcklig för att bygga projektet och tar bort resten. Vi visar att vår beroendespecialiseringsmetod kan halvera förhållandet mellan projektklasser och beroendeklasser.

Till sist bedömer vi effekten av att avbloata projekt med avseende på klientapplikationer som återanvänder dem. Vi presenterar en ny täckningsbaserad avbloatningsteknik som bestämmer vilka klassmedlemmar i Java-bibliotek och dess beroenden som är nödvändiga för deras klienter. Vår avbloatningsteknik minskar effektivt storleken på avbloatade bibliotek medan man bevarar de väsentliga funktioner som krävs för att framgångsrikt bygga deras klienter.

**Nyckelord:** Mjukvaruavsvällning, mjukvaruberoenden, Java bytekod, pakethanterare, statisk programanalys, dynamisk programanalys

# Dedication

*Para Tony e Idalmis.*

---

*"Veni, vidi, vici"*

# List of Research Papers

The following is a list of papers included in this thesis in chronological order:

(I) C. Soto-Valero *et al.*, "The Emergence of Software Diversity in Maven Central", in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 333–343. DOI: 10.1109/MSR.2019.00059.

(II) C. Soto-Valero *et al.*, "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem", *Springer Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021. DOI: 10.1007/s10664-020-09914-8.

(III) C. Soto-Valero *et al.*, "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 (ESEC/FSE)*, 2021, pp. 1021–1031. DOI: 10.1145/3468264.3468589.

(IV) C. Soto-Valero *et al.*, "Coverage-Based Debloating for Java Bytecode", *ACM Transactions on Software Engineering and Methodology*, pp. 1–34, 2022. DOI: 10.1145/3546948.

(V) C. Soto-Valero *et al.*, "The Multibillion Dollar Software Supply Chain of Ethereum", *IEEE Computer*, vol. 55, no. 10, pp. 26–34, 2022. DOI: 10.1109/MC.2022.3175542.

(VI) C. Soto-Valero *et al.*, "Automatic Specialization of Third-Party Java Dependencies", *In arXiv*, pp. 1–17, 2023. DOI: 10.48550/ARXIV.2302.08370. *Under major revision at IEEE Transactions on Software Engineering (as of February 2023)*.

The following is a list of other papers contributed by the author not included in this thesis, in chronological order:

(I) C. Soto-Valero *et al.*, "Detection and Analysis of Behavioral T-Patterns in Debugging Activities", in *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 110–113. DOI: 10.1145/3196398.3196452.

(II) A. Benelallam *et al.*, "The Maven Dependency Graph: a Temporal Graph-Based Representation of Maven Central", in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 344–348. DOI: 10.1109/MSR.2019.00060.

(III) N. Harrand *et al.*, "The Strengths and Behavioral Quirks of Java Bytecode Decompilers", in *Proceedings of the IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 92–102. DOI: 10.1109/scam.2019.00019.

(IV) C. Soto-Valero and M. Pic, "Assessing the Causal Impact of the 3-Point per Victory Scoring System in the Competitive Balance of LaLiga", *International Journal of Computer Science in Sport*, vol. 18, no. 3, pp. 69–88, 2019. DOI: 10.2478/ijcss-2019-0018.

(V) N. Harrand *et al.*, "Java Decompiler Diversity and Its Application to Meta-Decompilation", *Journal of Systems and Software*, vol. 168, p. 110 645, 2020. DOI: 10.1016/j.jss.2020.110645.

(VI) G. Halvardsson *et al.*, "Interpretation of Swedish Sign Language Using Convolutional Neural Networks and Transfer Learning", *Springer SN Computer Science*, vol. 2, no. 3, p. 207, 2021. DOI: 10.1007/s42979-021-00612-w.

(VII) T. Durieux *et al.*, "DUETS: A Dataset of Reproducible Pairs of Java Library–Clients", in *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 545–549. DOI: 10.1109/MSR52588.2021.00071.

(VIII) N. Harrand *et al.*, "API Beauty Is in the Eye of the Clients: 2.2 Million Maven Dependencies Reveal the Spectrum of Client–API Usages", *Journal of Systems and Software*, vol. 184, p. 111 134, 2022. DOI: 10.1016/j.jss.2021.111134.

(IX) M. Balliu *et al.*, "Challenges of Producing Software Bill Of Materials for Java", *In arXiv*, pp. 1–10, 2023. DOI: 10.48550/arXiv.2303.11102. *Under major revision at IEEE Security & Privacy (as of May 2023).*

(X) J. Ron *et al.*, "Highly Available Blockchain Nodes With N-Version Design", *In arXiv*, pp. 1–12, 2023. DOI: 10.48550/arXiv.2303.14438. *Under review at IEEE Transactions on Dependable and Secure Computing (as of March 2023)*.

# Acknowledgements

First and foremost, I must express my deepest gratitude to my main supervisor, Benoit Baudry, for his unwavering support, guidance, and friendship throughout these five incredible years of research. Your ability to make sense of software through art has been truly inspiring. I remember the moment we first met in person, I told you that you have changed my life, and indeed, now I truly realized how much you have! I could not have wished for a better supervisor. Equally, I am tremendously thankful for my co-supervisor, Martin Monperrus, whose intelligence and high standards in research have taught me how to pursue excellence. You have always wanted the best for your Ph.D. students and I have been fortunate to learn from both of you.

I would like to extend my heartfelt appreciation to the distinguished members of my committee. Professor Diomidis Spinellis, thank you for accepting the role of opponent in my defense; I could not have imagined a better person for this task. My gratitude to Professor Pontus Johnson for his valuable role as the advanced reviewer of my thesis. Professors David Broman, Philipp Leitner, Dr. Valentina Lenarduzzi, and Dr. Antonio Sabetta, thank you all for being a part of my defense committee and for offering your invaluable insights.

To my research colleagues, a sincere thank you for your support and camaraderie. Amine Benelallam, I am truly grateful for your encouragement at the beginning of my Ph.D. journey; your prediction of my success has come true! Thomas Durieux, thank you for your immense help in pushing forward two of our papers; your skills and expertise as a top researcher have been an inspiration. Javier Cabrera Arteaga, “gracias por estar en los momentos importantes, mi amigo.” Javier Ron, thank you for your calming presence, and Nicolas Harrand, for your assistance and mindful technical (and political) conversations. Long Zhang, you have been a role model for us all as Ph.D. students. Deepika Tiwari, thank you for asking thought-provoking questions and pushing me to think critically. I thoroughly enjoyed our collaboration on my last Ph.D. paper, and your insights

have greatly contributed to increasing its quality, and Khashayar Etemadi, your enthusiasm for research is contagious. Fernanda Madeiral, thanks for the tiramisu. Benjamin Loriot, He Ye, Maria Kling, Zimin Chen, Erik Natanael Gustafsson, Aman Sharma, Yuxin Liu, thank you for being amazing colleagues and friends throughout this journey.

I cannot express enough gratitude to my family for their unwavering love and support. To my aunt, Mercedes Novo, thank you for your unconditional help since my childhood. Ermelinda Castellanos, my mother-in-law, thank you for always being there for me. My dear wife, Mabel González Castellanos, I am forever grateful for your patience, support, and love during the most challenging times. You have been my rock, and I am blessed to have you by my side.

Lastly, I would like to acknowledge a few other individuals who have made a significant impact on my life. Marcelino Rodriguez Cancio, "gracias por darme el empujón inicial que cambió mi vida." Boris Camilo Rodríguez Martín, thank you for your passion and for introducing me to the world of research. And to Tim Toady, thank you for never letting me down.

This Ph.D. journey has been an incredible experience, filled with growth, fulfillment, and achievement. I am deeply grateful to each and every person mentioned here for their contributions to my success.[1]

[1]The author of this thesis was financially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

# Contents

**List of Research Papers** v

**Acknowledgements** ix

**I Thesis** 3

**1 Introduction** 5
- 1.1 Software Debloating . . . 7
- 1.2 Debloating Java Dependencies . . . 9
- 1.3 Problem Statements . . . 11
- 1.4 Summary of Thesis Contributions . . . 11
- 1.5 Summary of Research Papers . . . 13
- 1.6 Thesis Outline . . . 18

**2 State of the Art** 19
- 2.1 Code Bloat in the Software Engineering Lifecycle . . . 20
- 2.2 Related Work on Software Debloating . . . 23
  - 2.2.1 Purposes for debloating . . . 28
  - 2.2.2 Code analysis techniques for debloating . . . 34
  - 2.2.3 Granularity of debloating . . . 40
- 2.3 Novel Contributions of This Thesis to Software Debloating . . . 45
- 2.4 Summary . . . 47

**3 Thesis Contributions** 49
- 3.1 Essential Dependency Management Terminology . . . 50
- 3.2 Contribution #1: Removing Bloated Dependencies . . . 52
  - 3.2.1 Novel concepts . . . 52
  - 3.2.2 Bloat detection . . . 53

3.2.3 Bloat removal . . . . . 54
3.2.4 Debloating assessment . . . . . 56
3.2.5 Key insights . . . . . 56
3.3 Contribution #2: Specializing Used Dependencies . . . . . 58
3.3.1 Novel concepts . . . . . 58
3.3.2 Bloat detection . . . . . 59
3.3.3 Bloat removal . . . . . 60
3.3.4 Debloating assessment . . . . . 61
3.3.5 Key insights . . . . . 61
3.4 Contribution #3: Debloating With Respect to Clients . . . . . 62
3.4.1 Novel concepts . . . . . 63
3.4.2 Bloat detection . . . . . 64
3.4.3 Bloat removal . . . . . 64
3.4.4 Debloating assessment . . . . . 65
3.4.5 Key insights . . . . . 66
3.5 Contribution #4: Reproducible Research . . . . . 67
3.5.1 Software tools . . . . . 68
3.5.2 Reproducible datasets . . . . . 70
3.6 Summary . . . . . 72

**4 Conclusions and Future Work** **75**
4.1 Key Experimental Results . . . . . 75
4.2 Reflections on Empirical Software Engineering Research . . . . . 77
4.3 Future Work . . . . . 79
4.3.1 Neural debloating . . . . . 80
4.3.2 Debloating across the whole software stack . . . . . 81
4.4 Summary . . . . . 83

**References** **85**

**II Appended Research Papers** **105**

# Part I

# Thesis

# Chapter 1

# Introduction

> *"This is your last chance. After this there is no turning back. You take the blue pill, the story ends. You wake up in your bed and believe whatever you want to. You take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole goes. Remember, all I'm offering is the truth. Nothing more."*
>
> — Morpheus, *The Matrix*

CODE reuse is a software engineering practice in which developers rely on pre-existing code components, libraries, or modules to build new software applications, rather than implementing everything from scratch [17]. This approach has been advocated as a good practice since the early days of software engineering, as it helps developers to increase productivity [18] and learn from past experiences to create software that is more robust, efficient, and maintainable [19]. As software engineering practices evolve, various mechanisms have been developed to facilitate code reuse, such as object-oriented programming, public APIs, open-source components, and package managers. These techniques and tools have made it even more convenient and efficient for developers to incorporate pre-existing code components into their projects.

In recent years, the use of package managers to handle software dependencies (*a.k.a.* libraries) has become a standard software engineering practice [20]. Software ecosystems and package managers provide developers with a centralized location to find and download the dependencies they need, as well as to keep them up to date [21]. Part of the success of package managers is attributed to their effectiveness in helping developers navigate the escalating complexity of code reuse within the current software engineering lifecycle [22]. Package managers

boost software reuse by creating a clear separation between the application and its third-party dependencies [23]. As a result, software ecosystems and package managers have become an essential part of modern software development and a key enabler of the rapid pace of innovation in this field [24]. There exist package managers for most programming languages, such as MAVEN for Java [25], NPM for JavaScript [26], and PIP for Python [27]. Each of them effectively handles the massive demands of code reuse across millions of dependencies hosted in public repositories, such as the Maven Central repository [28] for the Java ecosystem. This has greatly simplified the process of managing dependencies, making it easier for developers to build and maintain complex software systems.

Software dependencies pervade the landscape of modern software development. For example, in 2022 the average Java application depends on more than 40 third-party dependencies [29]. Despite the myriad of advantages that package managers offer, such as streamlining software reuse and simplifying dependency management, their widespread adoption has introduced new challenges that developers must contend with [30]. Developers of software applications must effectively overcome the challenges of managing these third-party dependencies [31] to avoid entering into the so-called "dependency hell" [32]. These challenges relate to ensuring high-quality dependencies [33], keeping the dependencies up-to-date [34], or making sure that heterogeneous licenses are compatible [35]. Consequently, the effective management of software dependencies has become an indispensable aspect of modern software development.

Dependencies are reusable software components that are commonly designed for multiple uses and platforms [36]. For example, the Apache PDFBOX library [37] is a versatile and multi-functional project, serving a wide array of features designed to run on various development environments. The PDFBOX APIs enable developers to create, process, and extract content from PDF files, accommodating diverse use cases like text extraction, form filling, and PDF rendering. This multi-functionality, while advantageous in providing diverse features and capabilities to its users, often has an engineering cost. When used as a dependency by another project, the Apache PDFBOX library may introduce a considerable amount of unnecessary code, commonly referred to as "software bloat" [38]. This is because PDFBOX is designed to cater to numerous use cases and platforms, many of which may not be relevant to a specific user. As a result, applications that rely on the PDFBOX and other multi-purpose libraries may suffer from increased code complexity, memory usage, longer compilation times, and larger distribution package sizes, potentially affecting the overall performance and user experience in its dependent applications.

The problems associated with the presence of software bloat aggravates as developers rely more on pre-existing code. The number of dependencies used in a project can quickly add up, resulting in large amounts of unnecessary code [39]. Moreover, the excess of code not only takes up more disk space but can lead to a number of problems, such as a higher risk of software vulnerabilities [40], increased memory usage [41], and longer build times [42]. Additionally, as software dependencies are often updated independently of the main project, it can be difficult to keep track of the version of dependencies that a project relies on and this could be a potential source of bugs [21]. As the challenges associated with the phenomenon of software bloat escalate, researchers are turning their attention to innovative solutions to mitigate its negative effects.

## 1.1 Software Debloating

To address the phenomenon of software bloat, researchers are exploring a technique known as "software debloating," which aims to remove unnecessary code and features from software applications. Effectively debloating software involves addressing three key challenges: 1) detecting the bloated code, 2) removing it, and 3) assessing that the debloated artifact preserves its original behaviour. The first challenge entails a thorough examination of the codebase and the software development lifecycle to pinpoint areas containing unnecessary or redundant code [43]. The second challenge involves surgically removing the bloated code through code-specific transformation techniques [44]. Finally, assessing the validity of the debloated artifact requires comprehensive testing and validation to ensure that the removal of bloated code has not introduced new errors or adversely impacted the application's functionality, performance, or reliability [45]. By effectively executing these tasks, developers can create leaner, more efficient software, and ensure a better user experience.

Detecting code bloat is notably difficult due to the intricacies and complexities associated with modern software systems. Identifying the unnecessary or redundant code segments requires a deep understanding of the application's functionality, its dependencies, and the relationships between different code components. Bloated code might be intertwined with essential functionalities, making it difficult for developers to discern which parts are truly unnecessary. Current techniques to detect code bloat rely on static [43] and dynamic [46] program analysis to accurately determine the code segments contributing to bloat. Although they are effective in most circumstances, often difficulties arise due to the dynamic features that modern programming languages and libraries may include, such

as reflection, dynamic loading, or runtime code generation [47]. These features make it challenging to determine the precise set of code segments that are used or unused at runtime, complicating the debloating process [48]. Moreover, the effectiveness of these techniques may be limited by factors such as the scalability of the bloat detection algorithm, the use of code obfuscation tools in the target application, or the lack of well-defined criteria for determining the targeting code bloat. Consequently, researchers continue to explore new methodologies and tools to enhance the accuracy and efficiency of techniques to effectively detect code bloat.

Removing bloated code presents its own set of challenges, as the process involves finding a way to eliminate the unnecessary code parts without compromising the necessary functionalities of the applications or introducing new bugs [49]. One significant challenge to this task lies in the interdependencies present in complex software systems [50]. Software components are often tightly interconnected, and removing a seemingly unnecessary piece of code (*e.g.* changing a single line of code in a configuration file) could inadvertently break other parts of the application that depend on it, either directly or indirectly [51]. On the other hand, dependencies between code components may not always be immediately apparent, leading to the inadvertent removal of critical code. Consequently, the act of removing bloated code might result in unintended side effects, such as performance degradation, instability, or altered application behavior. To mitigate these risks, developers must adopt sound code transformation techniques, coupled with thorough testing to ensure that the debloating process does not introduce unforeseen issues.

Assessing the integrity of a debloated artifact is another critical aspect of the software debloating process that poses unique challenges [52]. Ensuring that the removal of the bloated code has not introduced new errors or adversely impacted the application's functionality, performance, or reliability requires comprehensive testing and validation. Designing and executing a robust debloating assessment mechanism that effectively covers all aspects of the application's behavior can be a time-consuming and resource-intensive task. Current debloating methodologies depend on pre-existing applications' test suites to assess the efficacy of the debloating approaches [53]. Nonetheless, false positives or negatives during the testing process may result in unforeseen errors arising long after debloating has taken place. Therefore, a thorough evaluation is required to ensure that all relevant code paths are covered and that the removed code does not affect the application's functionality [54]. Overall, researchers must ensure that debloating techniques do not significantly impact the maintainability and readability of the code. Striking

the right balance between removing the bloated code and preserving its integrity and maintainability is still an open research endeavor.

## 1.2 Debloating Java Dependencies

In the context of this thesis, we investigate the use of debloating techniques to remove the software bloat resulting from the addition of third-party dependencies. Tackling software bloat within third-party dependencies poses unique challenges, primarily due to the restricted influence that developers possess over the internals of these libraries [55], which complicates the process of identifying and removing unnecessary code without altering the libraries' binaries. Moreover, bloated code resulting from the practice of code reuse can manifest at various granularity levels, from entire software modules to individual lines of code, adding to the complexity and time-consuming nature of the debloating process [56]. Overcoming these obstacles necessitates substantial engineering efforts, a thorough evaluation of the debloated artifact, and a profound understanding of the target application and its downstream dependencies.

In Java, as with many other programming languages, code reuse is a fundamental practice to increase developers' productivity [57, 39, 58]. Package managers, like MAVEN or GRADLE, streamline this practice by facilitating the task of reusing dependencies hosted in external repositories [8]. However, effectively handling Java dependencies poses several challenges for developers [59]. For example, each package manager has its own unique set of protocols, tools, and mechanisms that govern how dependencies are coordinated in software projects. This means that developers must not only familiarize themselves with the specific package manager's syntax and conventions but also adapt their needs to its particular dependency resolution algorithms and dependency versioning schemes. Furthermore, developers should also pay attention to the design choices made by public software repositories hosting the dependencies they incorporate into their projects For instance, software artifacts hosted in Maven Central are immutable, once an artifact is uploaded and published, it cannot be removed or modified [1]. Consequently, Maven Central accumulates all the versions of all the dependencies ever released there, and applications that declare a dependency towards a library must ensure to pick the right version. Although MAVEN provides features allowing developers to visualize the dependencies they utilize, managing dependency updates proves challenging due to the intricate nature of dependency trees [21]. For example, MAVEN could benefit from mechanisms that ascertain whether a declared

dependency is truly essential for the project using it [2]. These complexities and challenges associated with dependency management contribute significantly to the emergence of code bloat in the Java ecosystem.

We have observed that code bloat is a prevalent issue that can emerge when utilizing Java dependencies. To alleviate its detrimental effects, developers need to carefully consider the dependencies they incorporate, ensuring that only those vital to the project are included [43]. For instance, when using functionalities from the Apache PDFBox library, developers should assess their specific requirements and only add the necessary features into their project [60]. If the project solely involves extracting text from PDF files, there is no need to include the entire PDFBox library [61]. In this case, by selectively incorporating only the relevant modules or classes for text extraction, developers can effectively reduce software bloat. In addition, developers should also be aware of the different available versions of a dependency, and use the most recent and stable one to avoid vulnerabilities and issues associated to dependency conflicts [62].

Several software debloating techniques have been proposed to reduce the size and complexity of applications through the removal of unnecessary third-party code. For Java, various debloating techniques have emerged in the last two decades. Most of these techniques rely on static analysis [63] and dynamic analysis [48] to detect code bloat. While static and dynamic code analysis have shown promising results in identifying unused features [64] and other types of bloat in Java applications, there is still a need to extend their applicability to third-party libraries. Thus, as new software features and libraries are developed, debloating techniques must continue to evolve to keep up with the ever changing landscape of modern software development.

On the other hand, when undertaking the process of debloating a software project, t is essential to consider the potential impact on clients who will reuse its code as a dependency [65]. The removal of seemingly unnecessary or redundant code could inadvertently break the functionality of dependent projects if they rely on the removed parts in their codebases. This interdependency between several client projects can create challenges to the debloating efforts, as developers must carefully balance the need to optimize their software while ensuring the continued functionality of clients that rely on their code [66]. Despite some progress in this area, there is still work to be done to fully debloat Java applications and reduce their overall size and complexity. Comprehensive assessment of the debloating results, as well as communication with the clients of the projects, are essential in this context, as they help ensure that the debloating process does not compromise the stability, functionality, or performance of the dependent software applications.

## 1.3 Problem Statements

According to the discussions above, we identify three key problems to be addressed in the field of software debloating in Java:

- **P1:** The pervasive practice of software reuse, fueled by the increase in the supply of software dependencies leads to dependency bloat in the Java ecosystem.
- **P2:** Most of the code shipped with the used dependencies is unused by the dependent software projects.
- **P3:** Debloating software libraries could affect the clients that depend on these libraries, and the extent of such an impact is currently unclear.

## 1.4 Summary of Thesis Contributions

The essence of this thesis is on tackling the code bloat that arises as a result of the increasing complexity in software systems. The problems listed above represent the various facets of this phenomenon for a particular software ecosystem: the Java MAVEN ecosystem. In particular, our contributions focus on the fact that current debloating techniques for Java lack the ability to detect and remove code bloat coming from third-party dependencies. To overcome the existing limitations, we propose novel debloating techniques that prioritize minimally invasive changes in the dependency tree of software projects, thereby making it easier for developers to adopt them. Unlike existing debloating methods that focus on producing leaner binaries and enhancing the precision of static and dynamic program analysis for debloating, our contributions are centered on a different aspect. We target the removal of code originating from the software supply chain of third-party libraries, which we have identified as a fundamental source of code bloat. This not only contributes to enhancing the maintainability of the applications, but also reduces the attack surface and improves the projects' build performance. By leveraging the developers' familiarity with build systems, we implement debloating techniques that can readily debloat Java applications at build time. the development of MAVEN-based debloating tools has not only demonstrated significant value in addressing this challenge, but also facilitated user adoption.

In this thesis, we make the following technical contributions to the field of software debloating:

- **C1 Removing Bloated Dependencies:** In order to address **P1**, regarding the increase of dependency bloat in the Java ecosystem, we propose a software debloating approach to help developers identify and remove bloated dependencies in Java projects that build with MAVEN. Our approach is implemented in a tool called DEPCLEAN, which automatically removes direct, transitive, and inherited dependencies and produces a fully debloated version of the project's dependency tree. The corresponding paper is published in the journal *Springer Empirical Software Engineering* [2]. Moreover, armed with DEPCLEAN, we performed a longitudinal study of bloated dependencies in the Java ecosystem. We analyze the usage status of dependencies over time in order to determine to what extent a bloated dependency is likely to be used in the future. Our results are published as a conference paper in the *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* [3]. We present **C1** in details in Section 3.2.

- **C2 Specializing Used Dependencies:** In order to address **P2**, we develop a novel technique that specializes the individual dependencies in the dependency tree to the specific needs of Java projects. We implement this technique in a tool called DEPTRIM, which removes unused class files in third-party dependencies of projects that build with MAVEN. The corresponding paper is currently submitted to the journal *IEEE Transactions on Software Engineering*, and the PDF preprint is available on arXiv [6]. We present the details of **C2** in Section 3.3.

- **C3 Debloating *w.r.t.* Clients:** To address **P3** regarding the lack of insights about the impact of debloating libraries on their clients, we propose a novel debloating technique based on dynamic analysis that relies on the collection of execution traces from a diverse set of code-coverage tools to determine which class members in the Java libraries and their dependencies are actually necessary for their clients. We implement this technique in a tool called JDBL, and assess the applicability of this debloating technique on a large collection of Java libraries. The paper is published in the journal *ACM Transactions on Software Engineering and Methodology* [4]. We discuss **C3** in Section 3.4.

In addition to the technical contributions outlined earlier, this thesis also provides valuable experimental findings and makes meaningful contributions to public research.

Table 1.1: Mapping of the contributions in this thesis to the appended research papers.

| CONTRIBUTIONS | RESEARCH PAPERS | | | | | |
|---|---|---|---|---|---|---|
| | I [1] | II [2] | III [3] | IV [4] | V [5] | VI [6] |
| **C1** Removing Bloated Dependencies | | ✓ | ✓ | | | |
| **C2** Specializing Used Dependencies | | | | | | ✓ |
| **C3** Debloating *w.r.t.* Clients | | | | ✓ | | |
| **C4** Reproducible Research | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

- **C4 Reproducible Research:** For each proposed technical contribution (**C1**, **C2**, and **C3**), we design and carry out empirical studies that systematically assess the effectiveness of our software debloating approaches. Our methodologies, research protocols, and experimental outcomes serve as a valuable guide for researchers interested in exploring dependency usage and developing software debloating techniques in the future. Moreover, the datasets collected and curated by the author of this thesis offer a solid foundation for additional inquiries in this area. In support of open science, we share the complete source code of our research tools, datasets, experiment scripts, and results on GitHub and Zenodo.

Table 1.1 provides an overview of the technical contributions presented in the papers included in Part II of this thesis. Each paper has a distinct emphasis on the various technical contributions (**C1**, **C2**, and **C3**). Additionally, each technical contribution is evaluated through rigorous experimental protocols, ensuring their reliability and reproducibility. We have made a commendable effort in releasing our proposed software solutions as open-source code, together with the associated experiments and datasets, thereby promoting transparency and reproducibility of our research. Overall, our papers contribute significantly to the field of software debloating and dependency analysis in Java, offering experimental results, research software prototypes, and datasets to further advance the field (**C4**).

## 1.5 Summary of Research Papers

This is a compilation thesis that includes six research papers, each of which is summarized below. The papers are ordered based on the way in which the contributions are presented in this thesis.

**Paper I:** ***"The Emergence of Software Diversity in Maven Central"***

**César Soto-Valero**, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry
*In Proceedings of the 16th International Conference on Mining Software Repositories (2019)*

**Summary:** The Maven Central repository is immutable, which means that any artifact uploaded to Maven Central cannot be removed or altered, and upgrading a dependency requires the release of a new version. As a result, Maven Central accumulates all the versions of libraries published there, and any application declaring a dependency on a library has the freedom to choose among any version of that library. In this paper, we hypothesize that the immutability of MAVEN artifacts, coupled with the flexibility of the clients to choose any version, is conducive to the emergence of software diversity within Maven Central. To test our hypothesis, we conduct an analysis of 1,487,956 artifacts, which represent all versions of 73,653 libraries. Our findings reveal that more than 30 % of libraries have multiple versions that are actively being used by the latest artifacts. For popular libraries, over 50 % of their versions are utilized. Moreover, more than 17 % of libraries have multiple versions that are significantly more frequently used than others. Our results demonstrate that the immutability of artifacts in Maven Central supports a sustainable level of diversity among library versions in the repository. This paper contributes to **C4**.

**Own contributions:** The author of this thesis wrote the paper and established all technical results, with extensive feedback from discussions with the co-authors.

**Paper II:** ***"A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem"***

**César Soto-Valero**, Nicolas Harrand, Martin Monperrus, and Benoit Baudry
*Springer Empirical Software Engineering (2021)*

**Summary:** The prevalent practice of software reuse, driven by the growth in the availability of software dependencies, results in an accumulation of excessive dependencies within Java projects. This problem, presented in **P1** and discussed in Section 1.3, is known as dependency bloat. We propose a new technique, implemented in a tool called DEPCLEAN, that automatically detects and removes bloated dependencies in MAVEN projects. Bloated dependencies refer to third-

party libraries that are included in the application binary, yet are unnecessary for the application to function properly. DEPCLEAN detects bloated dependencies by constructing a call graph of the Java bytecode class members by capturing annotations, fields, and methods, and accounts for a limited number of dynamic features such as class literals. DEPCLEAN produces a variant of the dependency tree without bloated dependencies (*i.e.*, a debloated `pom.xml`). We evaluate DEPCLEAN both quantitatively and qualitatively. First, we analyze 9,639 Java artifacts hosted on Maven Central, which include a total of 723,444 dependency relationships. Our empirical results show that 75 % of the dependencies in Maven Central are bloated (*i.e.*, it is feasible to reduce the number of dependencies of MAVEN artifacts to 1/4 of its current count). Our qualitative assessment of DEPCLEAN with 30 notable open-source projects indicates that developers pay attention to bloated dependencies when they are notified of the problem: 21/26 answered pull requests proposing the removal of these dependencies were accepted and merged by developers, removing 140 bloated dependencies in total. This paper contributes specifically to **C1**.

**Own contributions:** The author of this thesis wrote the paper, implemented DEPCLEAN, and performed the experimental evaluation. The co-authors contributed significantly to motivate the importance of removing "bloated dependencies" and provided useful feedback during technical discussions.

**Paper III:** ***"A Longitudinal Analysis of Bloated Java Dependencies"***

**César Soto-Valero**, Thomas Durieux, and Benoit Baudry
*In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2021)*

**Summary:** In order to address **P1** regarding the uncertainty of developers when coming across bloated dependencies, we perform a longitudinal study that delves into the evolution and impact of bloated dependencies in the Java ecosystem. We use DEPCLEAN to determine the usage status of dependencies (*i.e.*, used or bloated) across the the history of 435 Java libraries. This represents analyzing a collection of 48,469 dependencies spanning a total of 31,515 versions of MAVEN dependency trees. Our results indicate a steady increase of bloated dependencies over time, with 89.2 % of direct dependencies labeled as bloated remaining as such in subsequent versions of the studied projects. Our empirical evidence suggests that developers can confidently remove bloated dependencies to streamline

application performance. Additionally, we discovered novel insights regarding the unnecessary maintenance efforts induced by dependency bloat. Notably, we found that 22 % of dependency updates made by developers were performed on bloated dependencies, and that DEPENDABOT, an automated dependency update bot, suggests a similar ratio of updates on bloated dependencies. By contributing these insights, we aim to inspire software developers to pay more attention to their dependency trees and take immediate actions to address the issue of bloated dependencies. This paper contributes to **C1**.

**Own contributions:** The author of this thesis wrote the paper in close collaboration with co-authors. The author of this thesis led the work on the experimental evaluation and the co-authors helped significantly with the data collection phases.

#### Paper IV: *"Coverage-Based Debloating for Java Bytecode"*

**César Soto-Valero**, Thomas Durieux, Nicolas Harrand, and Benoit Baudry
*ACM Transactions on Software Engineering and Methodology (2022)*

**Summary:** In order to address **P3**, related to the need for more knowledge regarding the impact of debloating software libraries for the clients that depend on these libraries, we develop a new debloating technique based on dynamic analysis, which we coined as "coverage-based debloating." For its implementation, we leverage state-of-the-art Java bytecode coverage tools to precisely capture which class members of a Java project and its dependencies are necessary to execute a specific workload. We implement this technique in a tool called JDBL. We use the client's test suite as a workload to remove code bloat and generate a debloated version of the packaged libraries. The evaluation of JDBL using a dataset of 94 open-source Java libraries yielded that coverage-based debloating achieves the removal of 68.3 % of the libraries' bytecode and 20.3 % of their total dependencies while maintaining the syntactic correctness and original functionality of the debloated libraries. Furthermore, our results demonstrate that 81.5 % of the clients with at least one test using the library successfully compile and pass their test suite when the original library is replaced by its debloated version. Our technique represents an advance in the field of software debloating using dynamic analysis. We offer a research tool for addressing the challenges posed by software bloat in modern Java application development. This paper contributes specifically to **C3**.

**Own contributions:** The author of this thesis wrote the paper, implemented JDBL,

and performed the experimental evaluation with the help of co-authors.

**Paper V:** ***"The Multibillion Dollar Software Supply Chain of Ethereum"***

**César Soto-Valero**, Martin Monperrus, and Benoit Baudry
*IEEE Computer (2022)*

**Summary:** The advent of blockchain technologies has sparked a flurry of activity in the research community, coding enthusiasts, and serious investors over the past decade. Ethereum, as the largest programmable blockchain platform to date, has enabled the trading of cryptocurrency, facilitated the creation of digital art, and ushered in a new era of decentralized finance through the use of smart contracts. The operation of the Ethereum blockchain is supported by a complex network of nodes, which rely on a vast array of third-party software dependencies, maintained by various organizations. The reliability and security of Ethereum are therefore directly influenced by these software suppliers. In this paper, we conduct a rigorous analysis of the software supply chain of third-party dependencies of BESU and TEKU, the two major Java Ethereum nodes. Our results uncover the inherent challenges in maintaining and securing the dependencies of both cutting-edge blockchain software projects. This paper contributes to **C4**.

**Own contributions:** The author of this thesis wrote the paper and performed the data analysis in close collaboration with co-authors. The original idea of the paper is from co-authors.

**Paper VI:** ***"Automatic Specialization of Third-Party Java Dependencies"***

**César Soto-Valero**, Deepika Tiwari, Tim Toady, and Benoit Baudry
*Under major revision at IEEE Transactions on Software Engineering (as of February 2023)*

**Summary:** In **C1**, we remove bloated dependencies entirely from the dependency trees of MAVEN projects. However, the partial use of remaining dependencies indicates potential for further reduction of third-party code. **P2** focuses on addressing the presence of this unused code in non-bloated dependencies. To tackle this issue, we introduce a novel technique that specializes Java dependencies based on their actual usage. We implement our technique in a tool called DEPTRIM, which systematically identifies the required subset of each dependency's bytecode necessary for building the, eliminating the unnecessary code parts. DEPTRIM

repackages the specialized dependencies and integrates them into the projects' dependency trees. We evaluate DEPTRIM with 30 notable open-source Java projects. DEPTRIM specializes $86.6\,\%$ of the dependencies in these projects, successfully rebuilding each with a specialized dependency tree. Through this specialization, DEPTRIM removes $47.0\,\%$ of unused classes from the dependencies, decreasing the ratio of dependency classes to project classes from $8.7\times$ in the original projects to $4.4\times$ after specialization. Our results emphasize the relevance of dependency specialization, as it can significantly reduce the share of third-party code in Java projects. This paper contributes to **C2**.

**Own contributions:** The author of this thesis wrote the paper, implemented DEPTRIM, and performed the experimental evaluation with the help of co-authors.

## 1.6 Thesis Outline

As a compilation thesis, this document consists of two parts. In Part I, Chapter 1 introduces the problem of debloating Java dependencies and summarizes the research papers included in this thesis that contribute to solving this particular problem. Chapter 2 presents a state-of-the-art of the field of software debloating and discusses the novelty of our contributions. Chapter 3 offers more details regarding our technical contributions. Chapter 4 concludes the thesis and discusses the potential future work. Part II of the thesis includes all the papers discussed in Part I.

# Chapter 2

# State of the Art

*"La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer."*

— Antoine de Saint-Exupéry

SOFTWARE bloat refers to code that is packaged in an application but is actually not necessary to run the application. In this chapter, we present an overview of the phenomenon of software bloat in the software development lifecycle and offer a comprehensive review of the most relevant research papers in the field of software debloating, consolidating the necessary background knowledge to comprehend our contributions. This consolidation of the literature is essential for understanding the complexities and challenges associated to software bloat, enabling researchers and practitioners to develop more effective debloating techniques in order to improve software efficiency, security, and maintainability. Our review involves a thorough examination of the pertinent published research papers that investigate this subject. In particular, our investigation reveals that the majority of the current literature can be categorized based on three fundamental aspects: purposes for debloating, code analysis technique for debloating, and granularity of the bloated code removal. We structure this chapter accordingly to reflect these salient concepts.

In the last part of this chapter, we position our contributions to the field of software debloating in relation to the most closely related tools and techniques. This provides a more concrete understanding of the unique and novel aspects of our contributions. In addition, we also draw attention to the current resources available, such as tools and datasets, which can be utilized as groundwork or benchmarks for forthcoming studies on software debloating.

## 2.1 Code Bloat in the Software Engineering Lifecycle

Software systems have a natural tendency to grow in size and complexity over time whether or not there is a need for it. This happens due to various factors such as advancements in hardware [56], contemporary programming practices [67], or sometimes for no apparent reason at all [38]. Consequently, software bloat emerges as a result of the natural increase in software complexity [68], *e.g.*, through the addition of non-essential features, bug fixes, or just by the accumulation of useless code that adds up over time [69]. This phenomenon has several unfortunate consequences. For example, it needlessly increases the size of the packaged software artifacts [38], makes software harder to understand and maintain [70], increases the attack surface [71], and degrades the overall performance [41]. The existence of software bloat poses challenges in the software development landscape. Therefore, it becomes increasingly important for developers and researchers to devise efficient strategies to mitigate its adverse effects for enhancing software quality.

> **Software bloat** refers to code that is packaged in an application but is actually not necessary to build and execute the application to provide a given functionality.

As software systems grow in size and sophistication, software stacks have also evolved to be more intricate and layered [72]. Modern applications are built on top of runtimes, which are in turn built on top of operating systems that depend on specific hardware architectures, and so on. Each layer adds its own set of features and dependencies, which may not be essential to the correct execution of one specific, user-facing application. Therefore, the escalating complexity throughout the entire software stack contributes to the increase of software bloat, making room for the introduction of unnecessary features, dependencies, and redundancies at various stages of the software development lifecycle [73]. In particular, software bloat increases when building on top of software frameworks [71], as well as with the practice of code reuse [74]. Moreover, software bloat accumulates across the entire software system, leading to performance issues, increased memory usage, and longer development and deployment times. This increasing level of complexity across the software engineering lifecycle makes it more difficult for developers to control the diverse components of applications [75], which further exacerbates the problem of software bloat.

Figure 2.1 illustrates the pervasive presence of software bloat throughout the software engineering lifecycle. The figure highlights three crucial phases of this

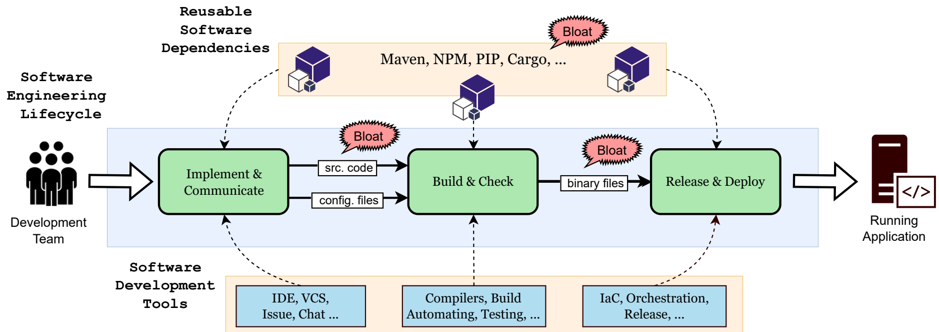



Figure 2.1: Presence of software bloat in the software engineering lifecycle when developing and deploying a software application.

process: implement & communicate, build & check, and release & deploy (depicted as green rounded rectangles). At the top of the figure, we represent dependencies as reusable software components managed by package managers, such as MAVEN for Java, NPM for JavaScript, and PIP for Python, which developers utilize during all software development phases.

First, in the implementation phase, developers fetch dependencies from external repositories to local repositories in order to reuse functionalities and expedite the application development process. Upon the compilation of the developers' source code, the second phase involves testing and building the application (*i.e.*, packaging the application's code along with the third-party code from dependencies, generally resulting in a single binary file). When the binary file is prepared, it is released and deployed into an execution environment, typically external servers that provide abstraction and isolation for reliable and efficient application execution (*e.g.*, cloud services powered by Docker and Kubernetes clusters). Figure 2.1 also displays software development tools at the bottom, assisting developers in each development phase (*e.g.*, IDEs, build automation tools, IaC, monitoring tools). For instance, in the case of a Java application, the Open JDK comprises the Java Runtime Environment (JRE) and additional tools necessary for building a Java application, including the Java compiler, debugger, and other development tools.

Figure 2.1 pinpoints three critical stages where software bloat appears, according to our experience. First, software bloat can occur after the implementation phase when developers include redundant source code or unnecessary features in their software projects [76]. This can encompass bloat in the code directly written by developers, as well as in the remaining configuration files required to build and check the software application. Second, when the software is built, compilers

and other tools may transform software artifacts (*e.g.*, when adding the code from third-party dependencies or inserting logging traces across the application for monitoring purposes). This additional code transformations can be a significant source of software bloat. In particular, compiled third-party dependencies are fetched from external repositories, added entirely, and packaged alongside the application's binaries.

By reusing dependencies developers are able to build more complex and powerful software systems with less effort. However, they can substantially contribute to software bloat, particularly when developers rely heavily on code coming from third-party libraries and frameworks. Furthermore, we observed that software repositories themselves may contain unnecessary or redundant dependencies. For example, each dependency is available in multiple versions, and each version contains its own set of downstream dependencies [1]. On the other hand, it is important to note that although the hardware layer supporting the running application is not a direct contributor to software bloat, more powerful hardware can encourage software developers to incorporate potentially bloated features [77].

As depicted in Figure 2.1, the engineering lifecycle of software applications is adversely affected by increased exposure to software bloat. This results from the challenges in identifying and eliminating redundant or unnecessary code within the numerous development phases and the inherent complexity of modern software systems. For example, one of the causes of software bloat is known as "feature creep," where software developers add new functionalities to software applications without considering their impact on the overall size and efficiency of the application [78, 79, 80]. We observe that the practice of code reuse can inadvertently contribute to increased software bloat. This practice can lead to the accumulation of unnecessary code and features that bloat the software and make it more difficult to maintain and optimize. Another cause of software bloat is code duplication, where developers copy and paste code without considering its relevance or impact on the overall software structure [81]. Furthermore, developers have limited control over certain stack components, such as the operating system or hardware, making it challenging to eliminate code bloat from these sources. Therefore, it is essential for developers to proactively address and manage the sources of bloat that are within their control, mitigating its adverse effects on the deployed software applications.

Software bloat affecting applications has been a widely-discussed topic in software engineering research. Numerous research papers have investigated the causes and consequences of software bloat, proposing various code removal techniques to eliminate unnecessary code and optimize software performance.

Specifically, software bloat has been identified as a significant challenge concerning software size, maintenance, performance, and security. Recent studies have concentrated on measuring the impact of software bloat across the software stack, encompassing user-level programs [36], OS kernels [82], and virtual machines [45]. Other research efforts have focused on elucidating the implications of software bloat on global energy consumption [44, 83, 84]. Lately, the research community has shown interest in examining the effects of software bloat on the software supply chain of dependencies [31], as it can contribute to increased complexity, diminished performance, and vulnerabilities [50]. In summary, the research findings demonstrate that software bloat is widespread and significant, affecting a substantial portion of code throughout the software development lifecycle. This situation is a unique opportunity for researches to develop innovative techniques for software debloating.

## 2.2 Related Work on Software Debloating

To address the issue of software bloat, various debloating techniques have been proposed in the research literature. One prevalent approach involves using static program analysis methods to identify unused or redundant code within compiled software applications [43], followed by code transformations and synthesis to remove these parts. Another approach employs dynamic analysis tools, which instrument and execute the application using a workload to detect code areas unnecessary for the workload execution [85], subsequently removing them. More recently, researchers have suggested employing a combination of both static and dynamic analysis techniques to enhance the accuracy and completeness of the bloat detection process. The effectiveness of the debloating task is enhanced when focusing on pinpointing code areas causing performance problems or consuming excessive resources.

**Software debloating** is the process of automatically detecting and removing software bloat across the software development lifecycle.

Despite the existence of debloating techniques, removing code bloat is an active research field in software engineering. Automatic debloating software poses three key challenges: 1) determining the location of the bloated parts [79], 2) removing these parts effectively [86], and 3) ensuring that debloated artifacts preserve the original behavior and provide useful features [85]. One major

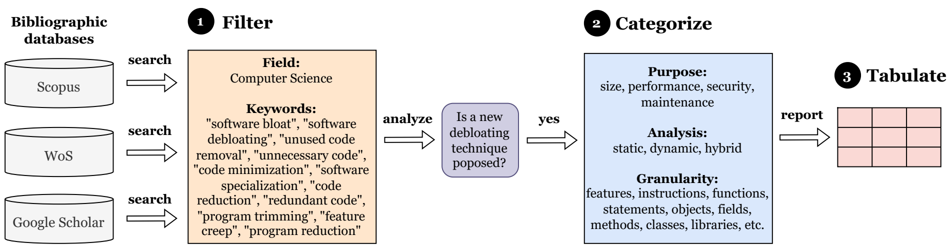



Figure 2.2: Overview of the methodology that we followed to find, categorize, and tabulate the state-of-the-art research papers on software debloating.

incentive for debloating is the complexity of modern software applications [87], which often consist of thousands or even millions of lines of code, leading to increasing technical debt [67]. This technical debt contributes to code bloat as developers may prioritize addressing urgent tasks or implementing new features over refactoring and optimizing existing code, resulting in the accumulation of redundant, unnecessary, or inefficient code segments [72]. Identifying and removing bloated code in such large-scale applications can be a daunting task, demanding significant time and resources from practitioners. Additionally, the interdependencies between different parts of software applications can make it difficult to remove code without breaking other parts of the software stack that serves the application.

Software debloating is a widely studied topic in the software engineering domain. In the the following, we present a comprehensive literature review on this topic. To provide a solid foundation for understanding the current state of research on software debloating, we first identify a list of papers covering the area according to a set of specific criteria. In particular we focus on papers in which a software debloating tool is proposed or an experiment to address software bloat is performed. We have read the selected papers carefully to consolidate a comprehensive knowledge of the field. Based on our analysis, we identified three aspects that characterize the state-of-the-art on this topic, which we propose as part of our contribution: (*i*) the objective or purpose of the debloating task, (*ii*) the code analysis technique employed to detect and remove code bloat, and (*iii*) the granularity at which the bloated code is removed. Our literature review highlights the more relevant tools and techniques, as well as the granularity at which bloat is addressed, based on this categorization.

Figure 2.2 illustrates the main steps of the methodology that we adopt in order to find the most relevant related work as of early 2023. Throughout the development of this thesis, we have been surveying the state-of-the-art, and now

we aim to consolidate a comprehensive list of relevant work using the methodology described as follows.

First ❶, we curated a list of keywords after careful consideration of the software debloating research field. Then, we search for relevant research papers using these keywords in three prominent databases: Scopus, WoS, and Google Scholar. Second ❷, we filter the list of papers obtained based on our expertise in the software debloating domain, ensuring that only the most relevant ones were included in further analysis. After filtering, we manually organize the papers by author names, venue of publication, title, and programming language used. Then, we categorize the papers based on their main debloating purposes, code analysis techniques employed, and granularity of the code removal approach. This categorization process facilitates a comprehensive analysis of the papers and helps identify trends and patterns among the previous contributions to this research field. Finally ❸, we organize and tabulate the relevant resulting papers, presenting a thorough and up-to-date overview of software debloating.

Table 2.1 presents the comprehensive list of research papers on software debloating published between 2002 and 2022. The table encompasses all the categories previously mentioned, offering a clear and detailed insight into the research landscape. By following the methodology outlined earlier, we provide an extensive overview of the pivotal research papers in this domain. We believe that this compilation could serve as a valuable resource for researchers and practitioners interested in the field of software debloating.

As a result of our analysis of papers published in various venues (column VENUE), we observe that previous works on software debloating propose diverse techniques, each tailored to a specific programming language (column PL). Notably, significant efforts have been dedicated to debloating C/C++ executable binaries, while debloating approaches for programming languages other than C/C++, Java, and JavaScript are almost nonexistent in the literature. In this context, we observe that the debloating process operates on programs that have already statically compiled and linked dependencies [88, 85], disregarding the bloat that arises from other aspects of the software engineering lifecycle, *e.g.*, from the usage and reliance on package managers. We also note that the majority of debloating efforts primarily focus on reducing program size, with less emphasis on improving maintainability (column PURP.). This imbalance in focus leads to the unintended consequence of creating software that is smaller in size but still difficult to maintain, update, and extend, ultimately hindering long-term software quality and manageability. Most works predominantly rely on static analysis to detect unreachable code, such as [89], [63], and [64], which is the most frequently employed

technique (column ANLYS.). Regarding debloating granularity (column GRAN.), a considerable amount of work is dedicated to removing bloat at the applications' fine-grain levels. However, we observe that there is a limited amount of research on debloating code from third-party dependencies introduced across various stages of the software development lifecycle. In the subsequent sections, we provide a more detailed overview of the key research papers for each of the distinguishing categories: debloating purpose, code analysis technique, and debloating granularity.

Table 2.1: Categorization of research papers on software debloating (years 2002 – 2022).

| REF. | VENUE | TITLE | PL | PURP. | ANLYS. | GRAN. |
|---|---|---|---|---|---|---|
| [90] | FSE | Cimplifier: automatically debloating containers | C/C++ | size | dynamic | Docker containers |
| [91] | TOSEM | Guided feature identification and removal for resource-constrained firmware | C/C++ | size | dynamic | features |
| [92] | FEAST | CARVE: Practical security-focused software debloating using simple feature set mappings | C/C++ | size | dynamic | features |
| [93] | GECCO | Removing the Kitchen Sink from Software | C/C++ | size | dynamic | features |
| [94] | SIGPLAN | Automatic feature selection in large-scale system-software product lines | C/C++ | size | dynamic | features |
| [85] | USENIX | RAZOR: A Framework for Post-deployment Software Debloating | C/C++ | size | dynamic | instructions |
| [95] | TECS | Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries | C/C++ | size | hybrid | libraries |
| [96] | SAC | Automated software winnowing | C/C++ | size | static | functions |
| [97] | DIMVA | BinTrimmer: Towards static binary debloating through abstract interpretation | C/C++ | size | static | instructions |
| [98] | ICSE | Perses: Syntax-guided program reduction | C/C++ | size | static | tokens |
| [42] | FMICS | Wholly!: a build system for the modern software stack | C/C++ | size, performance | dynamic | packages |
| [99] | CCS | Effective program debloating via reinforcement learning | C/C++ | size, performance | static | features |
| [100] | EuroSec | Configuration-driven software debloating | C/C++ | size, security | dynamic | features |
| [79] | ASE | TRIMMER: application specialization for code debloating | C/C++ | size, security | dynamic | features |
| [101] | FEAST | TOSS: Tailoring online server systems through binary feature customization | C/C++ | size, security | dynamic | features |
| [102] | CODASPY | Code specialization through dynamic feature observation | C/C++ | size, security | dynamic | instructions |
| [103] | USENIX | LIGHTBLUE: Automatic profile-aware debloating of bluetooth stacks | C/C++ | size, security | static | features |
| [88] | USENIX | Debloating software through piece-wise compilation and loading | C/C++ | size, security | static | features |
| [104] | DTRP | Large-scale debloating of binary shared libraries | C/C++ | size, security | static | functions |
| [105] | ASPLOS | One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries | C/C++ | size, security | static | instructions |
| [106] | ASIACCS | Pacjam: Securing dependencies continuously via package-oriented debloating | C/C++ | size, security | static | packages |
| [107] | NIER | Program debloating via stochastic optimization | C/C++ | size, security | static | statements |
| [108] | ACSAC | Nibbler: debloating binary shared libraries | C/C++ | size, security | static | libraries |
| [109] | PLDI | Blankit library debloating: Getting what you want instead of cutting what you dont | C/C++ | size, security, performance | dynamic | features, functions |

Continued on next page

Table 2.1: Categorization of research papers on software debloating (years 2002 – 2022). (Continued)

| | | | | | | |
|---|---|---|---|---|---|---|
| [53] | TSE | Trimmer: An automated system for configuration-based software debloating | C/C++ | size, security, performance | hybrid | instructions |
| [110] | CCS | Binary control-flow trimming | C/C++ | size, security | dynamic | features |
| [111] | USENIX | DECAF: Automaticlasses, adaptive de-bloating and hardening of COTS firmware | C/C++ | size, security | static | instructions |
| [112] | FSE | Cachetor: Detecting cacheable data to remove bloat | Java | performance | dynamic | collections |
| [113] | ISMM | A bloat-aware design for big data applications | Java | performance | dynamic | objects |
| [44] | ECOOP | Reuse, recycle to de-bloat software | Java | performance | dynamic | objects |
| [114] | PLDI | Detecting Inefficiently-Used Containers to Avoid Bloat | Java | performance | hybrid | objects |
| [115] | OOPSLA | Combining concern input with program analysis for bloat detection | Java | performance | static | statements |
| [78] | TSE | Xdebloat: Towards automated feature-oriented app debloating | Java | size | dynamic | features |
| [116] | MOBILE-Soft | Identifying features of android apps from execution traces | Java | size | dynamic | features |
| [117] | SCP | Slimming a Java virtual machine by way of cold code removal and optimistic partial program loading | Java | size | dynamic | JVMs |
| [48] | FSE | JShrink: In-Depth Investigation into Debloating Modern Java Applications | Java | size | hybrid | functions, methods, classes |
| [55] | FSE | Binary reduction of dependency graphs | Java | size | static | classes |
| [118] | ISSRE | RedDroid: Android application redundancy customization based on static analysis | Java | size | static | classes, methods |
| [89] | TOPLAS | Practical extraction techniques for Java | Java | size | static | functions, methods, classes |
| [63] | COMP-SAC | JRed: Program customization and bloatware mitigation based on static analysis | Java | size, security, maintenance, performance | static | classes, methods |
| [119] | CCS | Dissecting Residual APIs in Custom Android ROMs | Java | size, security | static | APIs |
| [70] | SIEP | Piranha: Reducing feature flag debt at Uber | Java | size, maintenance | static | features |
| [64] | HASE | Feature-based software customization: Preliminary analysis, formalization, and methods | Java | size, security | static | features |
| [120] | WWW | Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat | JS | size | dynamic | browser extensions |
| [80] | IST | Slimming JavaScript applications: An approach for removing unused functions from JavaScript libraries | JS | size | hybrid | functions |
| [121] | TSE | Evolving JavaScript code to reduce load time | JS | size | static | source code |
| [122] | TSE | Momit: Porting a JavaScript interpreter on a quarter coin | JS | size, performance | dynamic | features |
| [46] | EMSE | Stubbifier: debloating dynamic server-side JavaScript applications | JS | size, security, performance | hybrid | functions |
| [123] | CCS | Slimium: debloating the chromium browser with feature subsetting | JS | size, security | static | features |
| [124] | USENIX | Mininode: Reducing the Attack Surface of Node.js Applications | JS | size, security | static | files |
| [125] | EISA | JSLIM: Reducing the known vulnerabilities of JavaScript application by debloating | JS | size, security | static | functions |
| [126] | ACSAC | DeView: Confining Progressive Web Applications by Debloating Web APIs | JS | size, security | dynamic | APIs |
| [127] | OOPSLA | Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach | CSS | maintenance | static | statements |

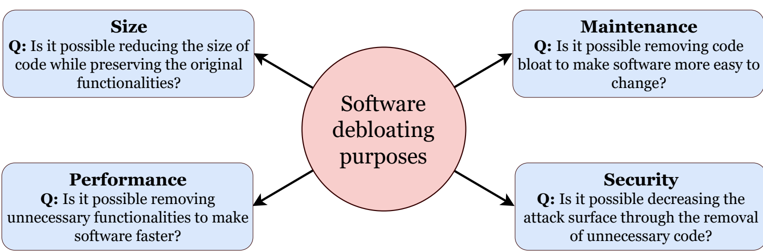



Figure 2.3: Illustration of the four main purposes for software debloating and their respective relevant research questions.

### 2.2.1 Purposes for debloating

We found that there are four key objectives of debloating that are widely acknowledged in the software engineering community: reducing applications' size, improving their performance, enhancing their security, and making software easier to maintain and update. Figure 2.3 depicts these objectives along with their corresponding critical research questions. In the following sections, we explore each of these purposes in detail.

#### Debloating for code size reduction

A primary goal of debloating software is to minimize its size. Bloated software can consume substantial disk space and bandwidth, posing challenges for users with limited storage or slow internet connections. By removing unnecessary code and other resources, debloated software artifacts can be accommodated on smaller devices and transferred more swiftly, resulting in improved download and upload times for users. From an engineering standpoint, smaller applications require fewer build resources, potentially reducing deployment costs and mitigating build errors [128].

Significant research effort has been directed towards reducing software size by removing unused API members, as there is evidence that a considerable proportion of API members are not widely used [14], *e.g.*, many classes, methods, and fields of popular Java libraries are provided but they are not used in practice [129]. Seminal work by Tip *et al.* [89] presents a set of techniques for reducing the size of Java applications. They propose a uniform approach for modeling dynamic language features and supplying additional user input through a modular specification language, reducing the class file archives of Java programs to 37.5 % of their original size. Pham *et al.* [130] implement a bytecode-based analysis tool to learn about the actual API usage of Android frameworks. The empirical evaluation

based on 200 K Android apps shows that most APIs usages are confined to a limited set of functionalities, which can be effectively learned and predicted to offer highly accurate API recommendations. Hejderup [131] study the actual usage of modules and dependencies in the Rust ecosystem and propose PRÄZI, a tool for constructing fine-grained call-based dependency networks for the Cargo package manager [132]. Using PRÄZI, the authors found that packages call only 40 % of their resolved dependencies, which emphasizes the need of reducing the size of those dependencies. Lämmel *et al.* [133] perform a similar large-scale study on API usage based on the migration of Abstract Syntax Trees (AST) code segments. Other studies have focused on understanding how developers use APIs on a daily basis [66, 134]. Some of the motivations include improving API design [135], reducing the amount of dependency code [14], and increasing developers' productivity [136]. Agadakos *et al.* [108] propose NIBBLER: a system that identifies and erases unused functions within shared libraries. NIBBLER works in tandem with defenses like continuous code re-randomization and control-flow integrity, enhancing them without incurring additional runtime overhead. The authors developed and tested a prototype of NIBBLER on x86-64 Linux. NIBBLER reduces the size of shared libraries and the number of available functions by up to 56 % and 82 %, respectively in a set of real-world programs.

Beyond APIs, the reduction of Docker container sizes has the advantage of decreasing the amount of data that needs to be transferred during applications' deployment or scaling, ultimately leading to lower network traffic and associated costs. In this context, the work of Rastogi *et al.* [90] specifically targets container debloating. They introduce a tool called CIMPLIFIER, designed to address bloat concerns in Docker containers by utilizing user-defined constraints. CIMPLIFIER partitions containers into streamlined, isolated units that communicate only when necessary and include solely the essential resources for their functionalities. Evaluations performed on popular DockerHub containers indicate that CIMPLIFIER not only preserves the original functionality but also significantly reduces image sizes by up to 95 %, efficiently processing even large containers in under 30 seconds.

**Insights on Debloating for Code Size Reduction**

Despite significant progress in software debloating for reducing code size, there is still ample opportunity for further research and development in this area. For example, exploring innovative debloating techniques for a broader range of programming languages and focusing on debloating dependencies can lead to more effective and efficient size reductions across various software ecosystems.

### Debloating for performance improvement

Debloating software not only reduces size but also enhances its performance. Bloated software frequently includes redundant or unnecessary code, leading to slower execution due to increased resource consumption. For instance, in Java, class initializers might create unused objects, resulting in higher memory usage and unnecessary overhead at runtime [137]. Eliminating such language specific code initializers through debloating could streamline Java applications, enabling faster execution times and improving overall performance, ultimately benefiting users.

Runtime bloat could significantly impair the performance and scalability of software systems. Xu and Rountev [114] introduce static and dynamic analysis tools for identifying inefficient container usage in Java programs. Their experiments reveal notable performance optimization opportunities for statically-identified containers, particularly those with high memory allocation frequency at runtime. Bhattacharya *et al.* [44] concentrate on detecting bloat arising from the temporary creation of containers and `String` objects within loops and propose a source-to-source transformation for efficient object reuse. The proposed method substantially reduces temporary object allocations and execution time, especially in programs with high churn rates or memory-intensive demands. Bhattacharya *et al.* [115] suggest leveraging feature information in program analysis to estimate the propensity to execute bloated code chunks in Java programs with optional concerns. The proposed approach enables the identification of specific statements likely causing bloat, which reveals the negative impact of optional features on runtime performance.

A large body of debloating techniques focuses on reducing applications build time. Celik *et al.* [138] present MOLLY, a build system to lazily retrieve dependencies in Continuous Integration (CI) environments and reduce build time. They show that MOLLY can speed-up the build time 45 % on average compared to the standard MAVEN build pipeline for a set of studied projects. Yu *et al.* [139] investigated the presence of unnecessary dependencies in header files of large C projects. They proposed a graph-based algorithm to statically remove unused code by pre-processing dependencies at the program units level, resulting in minimized build time. Nguyen and Xu [112] propose a novel runtime profiling tool called CACHETOR, which uses dynamic dependence profiling and value profiling to identify and report operations that generate identical data values, addressing the runtime bloat issues affecting modern object-oriented software by identifying optimization opportunities for performance improvement. Gelle *et al.* [42] present WHOLLY,

a tool designed to achieve reproducible and verifiable builds of optimized and debloated software that runs uniformly on traditional desktops, the cloud, and IoT devices. WHOLLY uses the `clang` compiler to generate LLVM bitcode for all produced libraries and binaries to allow for whole program analysis, specialization, and optimization. Furthermore, it uses Linux containers to ensure the integrity and reproducibility of the build environment.

**Insights on Debloating for Performance**

Although various techniques have been developed to reduce runtime bloat and optimize build times, further research is needed to explore new methods and enhance existing ones for even better performance gains. By continuing to investigate debloating strategies, the software engineering community can effectively tackle performance-related challenges, ensuring faster, more efficient software and building systems that ultimately benefit users and developers alike.

**Debloating for security enhancement**

Bloated software can contain hidden vulnerabilities that hackers can exploit to gain unauthorized access to systems and steal sensitive data. By removing unnecessary code and eliminating redundant features, software debloating can reduce its attack surface and improve its overall security. For example, the "Heartbleed" vulnerability [140], discovered in 2014 in the OpenSSL cryptographic software library, was caused by a buffer over-read vulnerability in OpenSSL's implementation of the Transport Layer Security (TLS) protocol's heartbeat extension. Using software debloating techniques to remove unused or rarely used features, such as the heartbeat extension [105], can reduce the attack surface and make the codebase easier to audit and more secure for its clients.

Significant work has focused on decreasing the attack surface of program binaries compiled to LLVM bitcode. Brown and Pande [92] propose CARVE, a simple yet effective security-focused debloating technique that utilizes static source code annotation to map software features, introduces debloating with replacement and removing vulnerabilities in four network protocol implementations across 12 scenarios. CARVE eliminates the need for advanced software analysis during debloating and reduces the overall level of technical sophistication required by the user when compared with other tools. Ghaffarinia and Hamlen [110] introduce a new method for automatically reducing the attack surfaces of binary software by removing unwanted or unused features, even in the absence of formal specifications or metadata, through a combination of runtime tracing, machine

learning, in-lined reference monitoring, and contextual control-flow integrity enforcement, resulting in low overhead and successful elimination of zero-day vulnerabilities. Koo *et al.* [100] propose a software debloating approach to mitigate the proliferation of code reuse attacks. The proposed debloating technique reduces the number of instruction sequences that may be useful for an attacker and eliminates potentially exploitable bugs. This approach is configuration-driven and removes feature-specific code that is exclusively needed only when certain configuration directives are specified, which are often disabled by default. The technique identifies libraries solely needed for a particular functionality and maps them to certain configuration directives, so feature-specific libraries are not loaded if their corresponding directives are disabled.

The prevailing goal of reducing the number of gadgets (*a.k.a.* features) available in a software package to reduce its attack surface and improve security has received significant interest from researchers and practitioners [88]. Decreasing the number of gadgets available in a software package reduces its attack surface and makes mounting gadget-based code reuse exploits, such as those based on return-oriented programming (ROP), more difficult for an attacker [53]. Brown and Pande [45] propose new metrics based on quality rather than quantity for assessing the security impact of software debloating. They show evidence that the process of software debloating can effectively reduce gadget counts at high rates. However, it may not effectively constrain an attacker's ability to fabricate an exploit. Furthermore, in certain situations, the reduction in gadget count may obscure the introduction of new quality gadgets, leading to a worsening of security rather than an improvement, such as in smartphone applications [141]. Koishybayev and Kapravelos [124] discuss the use of JavaScript as a programming language for both client-side and server-side logic, enabled by Node.js and its package manager, NPM. The paper introduces MININODE, a static analysis tool for Node.js applications that measures and removes unused code and dependencies, which can be integrated into the building pipeline of Node.js applications to produce applications with significantly reduced attack surface. MININODE was evaluated by analyzing 672 K Node.js applications, identifying 1,660 vulnerable packages, and successfully removing 2,861 of these packages while still ensuring builds succeed. More recently, Oh *et al.* [126] propose a tool called DEVIEW for reducing the attack surface of progressive web applications (PWAs) by blocking unnecessary but accessible web APIs. DEVIEW tackles PWA debloating challenges through record-and-replay web API profiling and compiler-assisted browser debloating, maintaining original functionality and preventing 76.3 % of known exploits on average.

**Insights on Debloating for Security Enhancement**

There remains a substantial amount of work to be done on debloating for security purposes, particularly in addressing vulnerabilities arising from third-party dependencies, which are known sources of security issues [142]. As the research has shown, there is potential for further exploration in this area, including enhancing security by mitigating gadget-based code reuse exploits, refining metrics for assessing the impact of debloating on long-term security, and improving the safety of software that relies heavily on code reuse.

### Debloating for maintenance

Bloated software can be more difficult to maintain and update, particularly if it contains redundant or poorly designed code. Debloating software projects can improve maintainability resulting in better overall software quality and developers satisfaction [143]. For example, current web applications include a large set of JavaScript files, some of which contain code that is never executed. Part of this code may have been added during the development process, but it is no longer needed for the application to function correctly [126]. Removing these unnecessary JavaScript files would decrease the size of the application, and with less code to worry about, developers can more easily understand and modify the codebase, which can reduce the amount of time it takes to make changes or fix bugs. In addition, debloated software can also lead to a more reliable and stable application because there are fewer opportunities for bugs or errors to be introduced [144]. Smaller codebases are also easier to test and can have faster testing times, which can lead to faster release cycles and more frequent updates and deployments.

There is scarce research work on the use of debloating for maintainability purposes. Jiang *et al.* [63] use a set of well-known code complexity metrics, including Chidamber and Kemerer (CK) object-oriented metrics [145], to assess the impact of debloating on code quality. They found that debloating can help reduce code complexity and increase code quality, but the degree of these improvements depends on the program's design and the nature of the application functions. Hague *et al.* [127] introduce an approach to detect redundant CSS rules in HTML5 applications by using an abstraction based on monotonic tree-rewriting, establishing the precise complexity of the problem, and proposing an efficient reduction to an analysis of symbolic push-down systems that yields a fast method for checking redundancy in practice, with demonstrated efficacy. They show

that code complexity is significantly reduced. Ramanathan *et al.* [70] presents PIRANHA, an automated code refactoring tool that generates differential revisions to remove code related to stale feature flags. PIRANHA analyzes the program's ASTs to generate refactoring suggestions and assigns the diff to the author of the flag for further processing before the application is landed. This tools has been implemented in multiple apps within Uber for removing unnecessary features in code written in Objective-C, Java, and Swift.

**Insights on Debloating for Maintenance**

Despite the existing evidence that debloating can improve code quality, reduce its complexity, and facilitate faster release cycles, there remains a significant need for more research to better understand its impact on maintainability. By further investigating debloating techniques and their applications, the software engineering community can work towards producing more maintainable, reliable, and efficient software systems that lead to higher user satisfaction and better overall software quality.

### 2.2.2 Code analysis techniques for debloating

In the last few years, a range of techniques has been developed by researchers to detect code bloat. Detecting code bloat is a challenging task as it requires the identification of unnecessary code or code that is almost never executed, which may be intertwined with necessary code segments that are often executed. Code bloat may be caused by various factors, such as excessive code reuse, lack of refactoring, or inadequate configurations, which makes it difficult to pinpoint a specific source of bloat. Existing bloat detection techniques rely on static analysis, dynamic analysis, or a hybrid approach that utilizes both techniques. Static analysis is useful for detecting potential sources of code bloat by analyzing the source code without actually executing it [146]. However, static analysis is more conservative and may fail to identify certain types of code bloat, such as those that are only apparent under specific conditions [147, 148, 149, 47]. On the other hand, dynamic analysis techniques are more aggressive, and the accuracy of the debloating heavily depends on the completeness of the workload employed.

Listing 2.1 shows a code example illustrating the challenges of using static and dynamic analysis for debloating, specifically when dealing with the dynamic features of the Java programming language. In this example, the method named `unusedMethod` is never called (line 31), and it could be safely detected and removed by debloating techniques that rely on static analysis. However, static analy-

```java
import java.lang.reflect.Method;
import java.util.Scanner;

public class Foo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            String className = "Foo";
            String methodName = "greet";
            String personName = scanner.next();
            // Dynamically loading a class
            Class<?> clazz = Class.forName(className);
            // Dynamically invoking a method using reflection
            Method method = clazz.getDeclaredMethod(methodName, String.class);
            method.invoke(null, personName);
        } catch (Exception e) {
            // Catch the exception
        }
    }

    // This method is invoked using reflection
    public static void greetAlice(String name) {
        if (name.equalsTo("Alice"){
            System.out.println("Hello, " + name);
        } else {
            System.out.println("Sorry, I don't know you");
        }
    }

    // This method is never called and could be removed by debloating
    public static void unusedMethod() {
        System.out.println("This method is never used.");
    }
}
```

Listing 2.1: Example of the challenges when using static and dynamic program analysis techniques to detect code bloat in a Java program that uses reflection.

sis techniques struggle to accurately identify the dependencies and relationships between classes and methods [150] when reflection is used [151]. For example, the class `Foo` is loaded via reflection (line 12) and the method `greetAlice` is invoked using reflection (line 15). Traditional static analyzers have difficulty identifying the relationship between this method and its invocation, leading to potential debloating errors. On the other hand, dynamic analysis involves the execution of the code and can identify instances of code bloat that appear only under specific conditions. Dynamic analysis techniques rely on the completeness of the workload or test suite to identify which parts of the code are actually used during execution. However, if the test suite or workload does not cover all possible use cases [152], there is a risk that the debloating process might remove code that is actually required in certain scenarios, leading to application failures when removing too much code. In this case, the value of the variable `personName` depends on the user-provided input (line 10), and therefore it is not possible to infer

which branch of the `if-else` statement will be executed (line 23) in all possible cases. Notice that if the user-provided workload is the String `Alice` then line 24 is executed, otherwise line 26 is executed instead. Dynamic program analysis may be computationally expensive as it requires executing the code, and may not cover all code paths. Combining the dynamic and static analysis approaches can improve the accuracy and efficiency of code debloating efforts.

It is worth noting that, while the Java compiler performs optimizations during compilation, it typically does not remove unused methods at this stage [137]. The Java Virtual Machine (JVM) and its Just-In-Time (JIT) compiler conduct more extensive optimizations at runtime, such as inlining methods and eliminating dead code. Nevertheless, these runtime optimizations usually do not remove unused methods from the generated class files or JAR files. As a result, although unused classes and methods may not impact the performance of the running application, they still add to the size of the compiled binary files [153]. To address this, debloating techniques and other post-compilation optimizations can be utilized to remove unused code, minimize the binary size, and enhance the overall maintainability of the codebase.

#### Debloating using static analysis

Using static analysis for debloating involves examining the source code of a software application to identify potential sources of code bloat. Sources of bloat include unused variables, functions, and classes, as well as code that is redundant or can be simplified. Static analysis tools use a range of algorithms and heuristics to identify code that can be removed or refactored, and some tools can even suggest alternative implementations that can improve performance. An advantage of static analysis techniques lies in their scalability and performance, as there is no need to execute the code, which is an expensive task (*e.g.*, when running tests or building artifacts).

Most debloating techniques for C/C++ are built upon static analysis and are conservative in the sense that they focus on detecting unreachable code (*i.e.*, sections of a program's code that can never be executed during the program's execution). Redini *et al.* [97] propose BinTrimmer, a tool to perform static program debloating on binaries. The authors propose a novel abstract domain technique, based on abstract interpretation, to improve the soundness of static analysis to reliably perform program debloating. According to the evaluation, BinTrimmer is 98 % more precise than the related work. Malecha *et al.* [96] propose "winnowing", a static analysis and code specialization technique that uses partial evaluation. The process preserves the normal semantics of the original

program, that is, any valid execution of the original program on specified inputs is preserved in its winnowed form. Invalid executions, such as those involving buffer overflows, may be executed differently. Biswas *et al.* [102] propose ANCILE, a code specialization technique that leverages fuzzing (based on user-provided seeds) to discover the code necessary to perform the functions required by the user.

In the Java ecosystem, Jiang *et al.* [63] propose JRED, a static analysis tool built on top of the SOOT framework to automatically detect unused code from both Java applications and the JRE. Additionally, the same authors present a novel approach [64] for customizing Java bytecode through static dataflow analysis and enhanced programming slicing, enabling developers to tailor Java programs based on users' requirements or remove redundant features in legacy projects. In the context of Android applications, Jiang *et al.* [118] conducts a comprehensive study of software bloat, categorizing it into compile-time and install-time redundancy, and proposes a static analysis-based approach for effectively identifying sources of code bloat in Android applications.

**Insights on Debloating using Static Analysis**

Debloating using static analysis has proven to be an effective approach for debloating software applications, providing scalability and performance advantages due to the absence of code execution. While existing tools such as JRED, BINTRIMMER, and ANCILE have demonstrated success in debloating Java and C/C++ applications, further research and development of debloating techniques are necessary to expand their applicability and effectiveness. For example, there is still room for improvement and innovation in developing novel tools that not only address code bloat in compiled applications but also tackle bloat issues related to configuration files and third-party dependencies.

### Debloating using dynamic analysis

Using dynamic analysis for detecting code bloat involves running a software application and monitoring its behavior to identify sources of code bloat. For example, this technique can be used to identify code that is rarely executed, code that consumes excessive resources, or code that can be optimized to reduce its size. Debloating based on dynamic analysis techniques is more aggressive and could remove reachable code [154], *i.e.*, the parts of an application that can be reached statically but that may not be executed at runtime, within a specific period, in a production environment. Dynamic analysis tools use a range of profiling and trac-

ing techniques to monitor the execution of a software application, and some tools can even automatically generate test cases to exercise code that is rarely executed.

In recent years, there has been a growing interest in developing debloating techniques for program specialization using dynamic analysis. These techniques aim to create smaller, specialized versions of programs that consume fewer resources and reduce the attack surface Azad *et al.* [71]. However, capturing complete and precise dynamic usage information for debloating is challenging, especially at scale, due to dynamic language features such as type-induced dependencies [155], dynamic class loading [149], and reflection [47]. Debloating techniques based on dynamic analysis have been applied to various contexts, ranging from C command line programs [103] and JavaScript frameworks [80] to fully containerized applications [90]. Sun *et al.* [98] propose PERSES, an approach that reduces programs by exploiting their formal syntax and focuses on smaller, syntactically valid variants, while Heo *et al.* [99] presents a C program reducer based on the syntax-guided Hierarchical Delta Debugging algorithm, which uses reinforcement learning to aggressively remove redundant code and improve processing time.

Dynamic analysis-based debloating has led to several novel approaches, such as the work by Landsborough *et al.* [93], which presents two distinct methods. The first approach employs dynamic tracing to safely remove specific program features but is limited to removing code reachable in a trace when an undesirable feature is enabled. The second approach utilizes a genetic algorithm to mutate a program until a suitable variant is found, potentially removing any non-essential code for proper execution, but possibly breaking program semantics unpredictably. Additionally, Sharif *et al.* [79] proposes TRIMMER, a tool using dynamic analysis to debloat applications based on user-provided configuration data, offering application specialization benefits by eliminating unused functionalities within a user-defined context. To further mitigate the construction of malicious programs, Porter *et al.* [109] introduces a demand-driven approach to reduce dynamically linked code surfaces by loading only the necessary set of library functions at each call site within the application at runtime, leveraging a decision-tree-based predictor and optimized runtime system.

**Insights on Debloating using Dynamic Analysis**

Debloating using dynamic analysis has demonstrated potential in generating specialized, efficient programs and reducing attack surfaces, leveraging runtime information. However, scalability challenges and the reliance on comprehensive workloads covering all use cases present significant barriers to its widespread adoption. To improve these debloating techniques, research should also concentrate on identifying code bloat in third-party dependencies, which frequently contribute to increased application size and complexity.

### Debloating using hybrid techniques

Using a hybrid approach for debloating involves combining both static and dynamic analysis techniques to identify and remove code bloat. This approach typically starts by executing static analysis to identify potential sources of code bloat and then using dynamic analysis to refine the code removal phase or validate the debloating results. Hybrid approaches for debloating can be more effective than using either static or dynamic analysis alone, as they strike a balance between the aggressiveness of dynamic analysis and the conservative advantages of static analysis. This allows for more comprehensive identification and removal of code bloat.

Bruce *et al.* [48] develop an end-to-end bytecode debloating framework called JSHRINK. It augments traditional static reachability analysis with dynamic profiling and type dependency analysis and renovates existing bytecode transformations to account for new language features in modern Java. The authors highlight several nuanced technical challenges that must be handled properly and examine behavior preservation of debloated software via regression testing. Qian *et al.* [85] introduces a debloating framework called RAZOR, which aims to reduce the size of bloated code in deployed binaries without requiring access to the program source code. RAZOR uses control-flow heuristics to infer complementary code necessary to support user-expected functionalities and generates a functional program with minimal code size. The framework has been evaluated on commonly used benchmarks and real-world applications, showing that it can reduce over 70 % of code from bloated binaries without introducing new security issues, making it a practical solution for debloating real-world programs. Quach *et al.* [88] introduce a generic inter-modular late-stage debloating framework. It combines static (i.e., compile-time) and dynamic (i.e., load-time) approaches to systematically detect and automatically eliminate unused code from program memory. This can be thought of as a runtime extension to dead code elimination. Unused code

is identified and removed by introducing a piece-wise compiler that not only compiles code modules (executables, shared resources, and static objects) but also generates a dependency graph that retains all compiler knowledge on which function depends on what other function(s).

**Insights on Debloating using Hybrid Techniques**

Debloating using hybrid approaches that combine static and dynamic analysis techniques for debloating offer a balance between the aggressive nature of dynamic analysis and the conservative benefits of static analysis. This can lead to more comprehensive identification of code bloat. There is a growing need to develop tools utilizing this approach and evaluate their effectiveness on real-world software applications to further enhance the soundness of static analysis for debloating purposes.

### 2.2.3 Granularity of debloating

One important aspect of debloating is the granularity at which it is performed. This ranges from coarse-grained debloating of entire features or modules to low-level debloating of individual program instructions or statements (as illustrated in Figure 2.4). The effectiveness of debloating at different levels of granularity depends on the specific software application and the goals of the debloating process. For example, coarse-grained debloating can be effective in removing a large amount of software bloat in an application but it may also remove useful functionalities for some particular users. On the other hand, fine-grained debloating can yield removing more targeted code pieces but it could be time-consuming and more challenging to implement. Multiple studies have been performed at different debloating granularities. Overall, care must be taken when removing code at each granularity level, as excessive removal may have unintended consequences that could negatively impact the program's behavior [156]. We discuss below the three main levels: level debloating, fine-grained, and coarse-grained debloating.

#### Debloating at low-level

The lowest level of granularity in debloating is instruction-level debloating, which involves identifying and removing individual source code pieces or program statements that are not essential to the core functionality of the software application. For instance, a particular instruction may have been added during the development process for debugging purposes or to accommodate a particular hardware

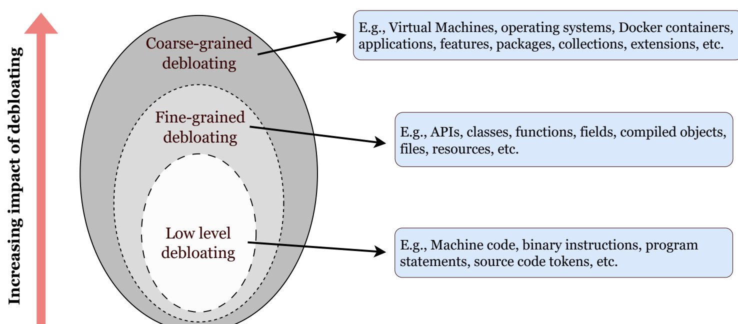



Figure 2.4: Granularity of debloating techniques and their impact according to the amount of bloated code removed.

architecture, but may not be necessary for the program to function properly. By removing such instructions, the size of the deployed code is reduced, which could result in faster execution times and improved performance. Overall, low-level debloating is challenging to implement due to the interdependencies between the different components of the software stack.

Wagner *et al.* [117] present a method to mitigate the bloatware problem in "always connected" embedded devices. Specifically, by storing the library code in a remote server. The instructions that are needed will be downloaded on demand. In addition, by applying some more sophisticated analysis, some library code can be downloaded in advance before they are actually executed to improve runtime performance. Morales *et al.* [122] proposes a multi-objective optimization approach, called MoMIT, to miniaturize JavaScript apps to run on IoT devices with limited memory, storage, and CPU capabilities, which reduces code size, memory usage, and CPU time while allowing the apps to run on additional devices. Xin *et al.* [107] propose a general approach that allows for formulating program debloating as a multi-objective optimization problem. The approach defines a suitable objective function, so as to be able to associate a score to every possible reduced program, and tries to generate an optimal solution (*i.e.* one that maximizes the objective function). According to Ziegler *et al.* [95], in the domain of embedded systems, there is a significant shift towards adopting commodity hardware and moving away from special-purpose control units in industrial sectors such as the automotive industry and avionics. As a result, there is a consolidation of heterogeneous software components to run on commodity operating systems during this transition. They propose an approach towards lightweight binary tailoring.

In addition, some studies have also examined debloating at the level of control

flow and data flow techniques in order to generate smaller program variants [110, 115, 157, 158]. Control-flow debloating involves identifying and removing redundant control structures such as loops or conditionals, while data-flow debloating involves identifying and removing redundant data structures or data accesses. Although these approaches have proven to be highly effective in reducing software bloat and improving performance, they may require more sophisticated tools and validation techniques.

**Insights on Debloating at the Low Level**

Debloating at low-level involves identifying and removing individual source code pieces or program statements that are not essential to the core functionality of the software application. Despite existing approaches, there is a growing need for the development of more sophisticated tools that can tackle debloating challenges at the level of control flow and data flow techniques in order to generate smaller program variants. By creating and evaluating such tools on real-world software applications, researchers can continue to improve the efficiency and performance of software systems while reducing bloat.

### Debloating at the fine-grained level

At a finer level of granularity, debloating can be performed at the level of API members, such as classes, functions, or variables. This approach involves identifying and removing entire classes or methods that are not used or are redundant within the software application. Fine-grained debloating can be more effective than lower-grained debloating in reducing software bloat, but it can also be more time-consuming and require more manual effort.

Tip *et al.* [89] explore extraction techniques, such as removing unreachable methods, inlining method calls, and transforming the class hierarchy to reduce application size, and introduces a uniform approach that relies on a modular specification language called MEL for supplying additional user input for modeling dynamic language features and extracting software distributions other than complete applications, while discussing associated issues and challenges with embedded systems applications extraction. Vázquez *et al.* [80] define the notion of Unused Foreign Function (UFF) to denote a JavaScript function contained in dependent libraries that are not needed at runtime. Also, they propose an approach based on dynamic analysis that assists developers to identify and remove UFFs from JavaScript bundles. The results show a reduction of JavaScript bundles of 26 %. Also for JavaScript, Turcotte *et al.* [46] present a fully automatic technique

that identifies unused code by constructing static or dynamic call graphs from the applications tests and replacing code deemed unreachable with either file- or function-level stubs. If a stub is called, it will fetch and execute the original code on-demand, thus relaxing the requirement that the call graph be sound. Kalhauge and Palsberg [55] presents a general strategy for reducing dependency graphs in input such as C#, Java, and Java bytecode, which has been a challenge for delta debugging. The authors present a tool called J-REDUCE, which achieves more binary reduction and is faster than delta debugging on average, enabling the creation of short bug reports for Java bytecode decompilers.

**Insights on Debloating at the Fine-Grain Level**

Most debloating approaches have focused on fine-grained debloating. There is a growing need to improve the application of these techniques to other programming languages and software ecosystems, as well as to debloat code elements from third-party dependencies. To address this, researchers could explore new strategies and tools that can effectively streamline dependency graphs, while ensuring compatibility with different programming languages and build systems.

### Debloating at the coarse-grained level

At the coarsest level of granularity, debloating can be performed at the level of entire features or modules. This approach involves identifying and removing entire code segments that are not essential to the core functionality of the software application. Coarse-grained debloating can be effective in reducing software bloat and improving performance, but it may also lead to the removal of useful or important functionalities.

Ruprecht *et al.* [94] propose an automated approach for d tailoring the system software for special-purpose embedded systems by completely removing unnecessary features. The goal is to optimize functionality and reduce memory usage, as exemplified by the significant memory savings (between 15 % and 70 %) achieved in tailored Linux kernels for Raspberry Pi and Google Nexus 4 smartphones. Rastogi *et al.* [90] propose a technique for debloating application containers running on Docker. They decompose a complicated container into multiple simpler containers with respect to a given user-defined constraint. Their technique is based on dynamic analysis to obtain information about application behaviors. The evaluation on real-world containers shows that this approach preserves the original functionality, leads to a reduction of the image size of up

to 95 %, and processes even large containers in under thirty seconds. Chen *et al.* [101] presents an approach called TOSS that automates the customization of online servers and software systems by identifying desired code using program tracing and tainting-guided symbolic execution, and removing redundant features through static binary rewriting to create a customized program binary. The approach was evaluated on MOSQUITTO, and it successfully created a functional program binary with only desired features, resulting in a significant reduction of the potential attack surface.

Bu *et al.* [113] propose a bloat-aware design paradigm towards the development of efficient and scalable Big Data applications in object-oriented GC-enabled languages. It points out that the negative impact on performance caused by bloatware has been significant on software specifically designed to handle large amounts of data, such as GIRAPH and HIVE. Qian *et al.* [123] present SLIMIUM, a debloating framework for the web browser CHROMIUM that harnesses a hybrid approach for fast and reliable binary instrumentation. The main idea behind SLIMIUM is to determine a set of features as a debloating unit on top of a hybrid (*i.e.*, static, dynamic, and heuristic) code analysis, and then leverage feature subsetting to code debloating. Starov *et al.* [120] investigate to what extent the page modifications that make browser extensions fingerprintable are necessary for their operation. By analyzing 58,034 browser extensions from the Google Chrome App Store, they discovered that 5.7 % of them were unnecessarily identifiable because of extension bloat. Agadakos *et al.* [104] present NIBBLER: a system that identifies and erases unused functions within dynamic shared libraries. NIBBLER works in tandem with defenses like continuous code re-randomization and control-flow integrity, enhancing them without incurring additional runtime overhead. NIBBLER reduces the size of shared libraries and the number of available functions.

**Insights on Debloating at the Coarse-Grain Level**

Debloating at the coarse-grained level has shown promise in reducing software bloat and improving performance. However, this approach may also lead to the removal of useful or important functionalities. Future work should focus on refining coarse-grain debloating techniques to maintain critical features while still optimizing software systems, exploring the application of these methods to various programming languages and software ecosystems, and evaluating their effectiveness in real-world scenarios.

Table 2.2: Comparison of existing Java debloating tools and techniques. TARGET is the type of artifact considered for debloating: bytecode (B), or source code (S); ANALYSIS refers to the type of code analysis performed for debloating: Static, Dynamic, or Hybrid; EXP. SCALE counts the number of study subjects used to evaluate the technique; GRANULARITY is the code level at which debloating is performed: field (F), method (M), class (C) or dependency (D). The four columns in EVALUATION CRITERIA present the criteria used to assess the validity the debloating technique: compilation (COMP.), test suite (TESTS), client applications (CLIENTS), and human developers via pull requests (DEVS). The last column, OUTPUT, is the outcome of the debloating techniques.

| REF. | TARGET | ANALYSIS | EXP. SCALE | GRANULARITY | | | | EVALUATION CRITERIA | | | | OUTPUT |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | F | M | C | D | COMP. | TESTS | CLIENTS | DEVS | |
| [63] | bytecode | Static | 9 libs | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ | ✗ | Debloated JARs |
| [55] | bytecode | Dynamic | 3 apps | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ | ✗ | Debloated JARs |
| [48] | bytecode | Hybrid | 26 projects | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ | ✓ | Debloated JARs |
| **C1** [2] | src. code | Static | 30 projects | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ | ✓ | Debloated POMs |
| **C2** [6] | bytecode & src. code | Hybrid | 30 projects | ✗ | ✗ | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | Specialized POMs |
| **C3** [4] | bytecode | Dynamic | 395 libs 1,370 clients | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Debloated JARs |

## 2.3 Novel Contributions of This Thesis to Software Debloating

Similar to other software stacks, Java applications often suffer from the detrimental effects of software bloat. Part of this bloat comes with the addition of new features, whereas another part is a result of reusing third-party dependencies. Dependency bloat negatively impacts the size of the applications, affects the project's maintenance, degrades performance, and potentially compromises security. To address this issue, we propose propose various techniques for debloating Java applications using code analysis techniques in order to detect and remove code bloat from third-party dependencies. In the following, we proceed to highlight the distinctive aspects of our contributions compared to the current state-of-the-art debloating techniques for Java.

Table 2.2 positions the research papers proposed in our contributions that come along with a software tool (*i.e.*, DEPCLEAN in **C1** [2], DEPTRIM in **C2** [6], and JDBL in **C3** [4]) in relation to the more related tools and techniques for software debloating in Java (*i.e.*, JRED in [63], J-REDUCE in [55], and JSHRINK in [48]). First, we note that all prior techniques focus on debloating Java bytecode rather than targeting source code. This is because targeting Java bytecode offers a more general and efficient method for bloat removal (*e.g.*, enabling debloating for JVM languages like Scala, Groovy, or Kotlin) while source code debloating introduces

extra complexities associated to compilation inconsistencies. In contrast, our tools DEPCLEAN and DEPTRIM focus on debloating dependency trees through the analysis of dependency and the subsequent transformation of *pom.xml* files. In addition to the technical contributions, we perform the first empirical study that explores and consolidates the concept of bloated dependencies in the MAVEN ecosystem and is the first to investigate the reaction of developers to the removal of bloated dependencies.

Existing techniques for detecting code bloat in Java predominantly utilize static and dynamic program analysis, with some employing hybrid approaches to tackle potential issues arising from the Java dynamic language features. As with our tools, existing debloating techniques primarily rely on static (JRED) and dynamic (J-REDUCE) program analysis algorithms to detect code bloat. In the case of JSHRINK, it adopts a hybrid approach to address the potential unsoundness of static analysis for detecting used code. In the case of DEPTRIM, it implements a novel variant of the hybrid approach in which the versions of the specialized dependency trees are validated based on the results of the project's tests when building with the specialized version of the dependency.

With regards to the scale of our experiments, both DEPCLEAN and DEPTRIM are assessed on a significant set of 30 notable MAVEN projects, surpassing the scope of prior studies. It is important to note that each contribution requires the projects to be built both before and after debloating, ensuring the integrity of the build process and of the debloated artifacts. Remarkably, we evaluate JDBL on 395 libraries and 1,370 client applications, which is an order of magnitude larger than previous work. JDBL stands as the pioneering debloating tool that utilizes a large set of clients of the debloated software artifacts for validation purposes.

With respect to the granularity of the code bloat removal, state-of-the-art Java tools focus on removing fields, methods, and classes. All prior tools excise classes, with only JSHRINK targeting fields. Besides removing methods and classes, our tools address bloat within third-party dependencies. For instance, DEPCLEAN eliminates entirely unused dependencies, while DEPTRIM removes classes from partially used dependencies in addition to discarding completely unused ones.

With respect to the debloat evaluation criteria, all previous works rely on compilation and tests (except JRED). Both JSHRINK and DEPCLEAN also involve a user evaluation with developers through pull requests. Utilizing developers via pull requests serves as an effective evaluation assessment for software debloating, as it leverages their expertise and familiarity with the codebase, ensuring the proposed debloating changes are relevant, maintain functionality, and align with the project's objectives. Furthermore, JDBL remains the sole study that incorporates client

applications' tests to evaluate the debloated artifacts' usability, extending beyond the confines of the project's scope.

In conclusion, a review of the literature on debloating for the Java ecosystem reveals that previous analysis techniques focus on fine-grained debloating, such as removing fields, methods, and classes. Although these existing debloating techniques can be effective at reducing program size and improving performance, they may not address all sources of code bloat, such as third-party dependencies in libraries and frameworks. As pointed out in Section 1.2, software dependencies in Java projects are responsible for a large amount of the shared code size in the compiled and packaged artifacts. Therefore, we identify a need to address dependency-related bloat in addition to fine-grained debloating, in order to reduce the overall size of a Java application and improve its performance, size, and maintainability.

## 2.4 Summary

In this chapter, we introduce software bloat, a pervasive problem affecting all layers of the modern software stack. We discussed how software bloat has emerged across the software development lifecycle, needlessly increasing the size of software applications, making them harder to understand and maintain, widening the attack surface, and degrading the overall performance. This phenomenon is rooted in several factors, including excessive code reuse, feature creep, code duplication, and other human and technology-related factors. We identified various software debloating techniques that have been proposed to mitigate software bloat at different granularities. However, we observe that removing code bloat remains a significant challenge due to the intricate nature and complexity of modern software applications and their interdependencies. As software complexity and feature richness continue to grow, tackling software bloat will remain a critical research area in software engineering.

# Chapter 3

# Thesis Contributions

> *"No te detengas avanza / Lucha prosigue y camina / Que el que no se determina / Nada de la vida alcanza / Nunca pierdas la esperanza / De realizar tus ideas / Cuando abatido te veas / Juega el todo por el todo / Y verás que de ese modo / Lograrás lo que deseas / No le temas al fracaso / Que el que por su bien batalla / No hay barrera ni muralla / Que le detengan el paso / Camina y no le hagas caso / Al que te hable con pesimismo / Busca la dicha en ti mismo / Como el hombre valeroso / Mira que el hombre penoso / Nunca sale del abismo."*
>
> — Mi abuelo, *Un día cualquiera hace años*

With the increasing complexity of Java applications and their reliance on third-party libraries, debloating Java dependencies has become an essential engineering task. In this chapter, we present the main contributions of this thesis to address the problem of software bloat in the Java ecosystem. We start with an overview of the Maven dependency management system and of its essential terminology, which constitutes the foundation for comprehending the technical contributions. As introduced in Section 1.4, our work contributes to the field of software debloating across three different aspects. First, we provide a mechanism to detect and remove bloated Java dependencies, thereby streamlining the dependency trees of software projects that build with Maven. Second, we specialize used dependencies to reduce the amount of third-party code, which yields even more benefits in terms of code size reduction. Finally, we evaluate the impact of debloating Java libraries in relation to their client applications through a novel coverage-based debloating technique, thus providing valuable insights into the efficacy of this debloating technique. Furthermore, we outline the tools and datasets we have contributed to promote reproducible research in this field.

```
1  <groupId>org.p</groupId>
2  <artifactId>p</artifactId>
3  <version>0.0.1</version>
4  <packaging>jar</packaging>
5  . . .
6  <dependencies>
7    <dependency>
8      <groupId>org.d1</groupId>
9      <artifactId>d1</artifactId>
10   </dependency>
11   <dependency>
12     <groupId>org.d2</groupId>
13     <artifactId>d2</artifactId>
14   </dependency>
15     <dependency>
16     <groupId>org.d3</groupId>
17     <artifactId>d3</artifactId>
18   </dependency>
19 </dependencies>
20 . . .
```

Listing 3.1: Excerpt of a MAVEN pom.xml file declaring three dependencies: $d_1$, $d_2$, and $d_3$.

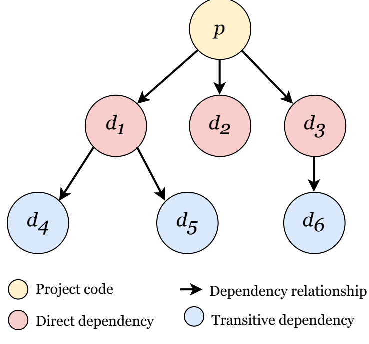



Figure 3.1: Dependency tree from the pom.xml file of Listing 3.1. The project $p$ declares the direct dependencies $d_1$, $d_2$, and $d_3$. The dependencies $d_4$, $d_5$, and $d_6$ are transitive dependencies of $p$.

## 3.1 Essential Dependency Management Terminology

MAVEN [25] is a popular package manager and build automation tool for Java projects and other programming languages that compile to the Java Virtual Machine (JVM), such as Scala, Kotlin, Groovy, Clojure, or JRuby. MAVEN is primarily designed to handle the dependencies within a software project. In addition to this crucial functionality, it also handles other tasks during the project build process, such as testing, packaging, and deployment. We define the key concepts associated with handling dependencies in the MAVEN ecosystem below.

*Maven Project.* We consider a project a collection of Java source code files and configuration files organized to be built with MAVEN. A MAVEN project declares a set of dependencies in a specific configuration file known as `pom.xml` (acronym for Project Object Model), which is located in the project's root directory. The `pom.xml` contains specific metadata about the project construction, its dependencies, and its build process. MAVEN projects are usually packaged and deployed to external repositories as single artifacts (JAR files). Listing 3.1 shows an excerpt of the dependency declaration in the `pom.xml` of a project $p$. In this example, developers explicitly declare the usage of three dependencies: $d_1$, $d_2$, and $d_3$. Note that the `pom.xml` of a Maven project is a configuration file subject to constant change and evolution: developers usually commit changes to add, remove, or update the version of a dependency.

*Maven Dependency.* A MAVEN dependency defines a relationship between a project $p$ and another packaged project $d \in \mathcal{D}$. Dependencies are compiled JAR

files, *a.k.a.* artifacts, uniquely identified with a triplet (`G:A:V`) where `G` is the `groupId`, `A` is the `artifactId`, and `V` is the `version`. Dependencies are defined within a scope, which determines at which phase of the MAVEN build cycle the dependency is required (*i.e.*, `compile`, `runtime`, `test`, `provided`, `system`, and `import`). Listing 3.1 shows an example of dependency relationships. By declaring a dependency towards $d_1$, the project $p$ states that it relies on some part of the API of $d_1$ to build and execute correctly. Dependencies are deployed to external repositories to facilitate reuse. Maven Central [28] is the most popular public repository to host MAVEN artifacts.

*Direct Dependency.* The set of direct dependencies $\mathcal{D}_{\text{direct}} \subset \mathcal{D}$ of a project $p$ is the set of dependencies explicitly declared in $p$'s `pom.xml` file. Figure 3.1 shows the direct dependencies in the first level of the dependency tree of $p$, *i.e.*, there is an edge between $p$ and each dependency $[d_1, d_2, d_3] \in \mathcal{D}_{\text{direct}}$. Direct dependencies are declared in the `pom.xml` by the developers, who explicitly manifest the intention of using the dependency.

*Transitive Dependency.* The set of transitive dependencies $\mathcal{D}_{\text{transitive}} \subset \mathcal{D}$ of a project $p$ is the set of dependencies obtained from the transitive closure of direct dependencies. Figure 3.1 shows the transitive dependencies in the second level of the dependency tree of $p$, *i.e.*, there is an edge between the direct dependencies of $p$ and each dependency $[d_4, d_5, d_6] \in \mathcal{D}_{\text{transitive}}$. Transitive dependencies are resolved automatically by MAVEN, which means that developers do not need to explicitly declare these dependencies. Note that all the bytecode of these transitive dependencies is present in the classpath of project $p$, and hence they will be packaged with it, whether or not they are actually used by $p$.

*Dependency Tree.* The dependency tree of a MAVEN project $p$ is a direct acyclic graph that captures all dependencies of $p$ and their relationships, where $p$ is the root node and the edges represent dependency relationships between $p$ and the dependencies in $\mathcal{D}$. Figure 3.1 illustrates the dependency tree of the project $p$, which `pom.xml` file is presented in Listing 3.1. In this example, $p$ has three direct dependencies, as declared in its `pom.xml`, and three transitive dependencies, as a result of the MAVEN dependency resolution mechanism.

*Maven Dependency Resolution Mechanism.* To construct the dependency tree, MAVEN relies on its specific dependency resolution mechanism [159]. MAVEN resolves dependencies in two steps: 1) based on the `pom.xml` file of the project, it determines the set of direct dependencies explicitly declared, and 2) it fetches the JAR files of the dependencies that are not present locally from external repositories such as Maven Central. Dependency version management is a key feature of the dependency resolution mechanism, which MAVEN handles with a specific

dependency mediation algorithm that avoids having duplicated dependencies and cycles in the dependency tree of a project [159].

*Maven Dependency Graph.* The Maven Dependency Graph (MDG) is a vertex-labeled graph, where vertices are MAVEN artifacts (uniquely identified by their `G:A:V` coordinates), and edges represent dependency relationships among them [8]. Formally, the MDG is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where: $\mathcal{V}$ is the set of artifacts in the Maven Central repository; and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represent the set of directed edges that determine dependency relationships between each artifact $v \in \mathcal{V}$ and its dependencies.

## 3.2 Contribution #1: Removing Bloated Dependencies

Our first contribution focuses on solving a specific challenge of dependency management: the existence of bloated dependencies. This refers to packages that are included as dependencies in a sofwtare project, and therefore get included in its dependency tree, but that are actually not necessary for building or running the project We develop a technique to effectively assess the impact of bloated dependencies across the entire MAVEN ecosystem, as well as to effectively eliminate them within MAVEN projects.

### 3.2.1 Novel concepts

For a set of dependencies $\mathcal{D}$, and in the context of a MAVEN project, we introduce the concept of bloated dependency in [2] as follows:

*Bloated Dependency.* A dependency $d \in \mathcal{D}$ in a software project $p$ is said to be bloated if there is no path in the dependency tree of $p$, between $p$ and $d$, such that none of the elements in the API of $d$ are used, directly or indirectly, by $p$.

We found this type of dependency relationship between software artifacts intriguing: from the perspective of the dependency management systems such as MAVEN that are unable to avoid it, and from the standpoint of developers who declare dependencies but do not actually use them in their applications. The major issue with bloated dependencies is that the final deployed binary file includes more code than necessary: an artificially large binary is an issue when the application is sent over the network (*e.g.*, web applications) or it is deployed on small devices (*e.g.*, embedded systems). Bloated dependencies could also embed vulnerable code that can be exploited while being actually useless for the application [160]. Overall, bloated dependencies needlessly increase the difficulty of managing and

evolving software applications, thereby making it imperative for developers to detect and remove them.

### 3.2.2 Bloat detection

The first task to eliminate dependency bloat is to detect bloated dependencies. Our proposed solution entails performing an in-depth analysis of the usage relationships among the class members of the entire dependency tree of MAVEN projects, which enables us to determine the usage status of each individual dependency (*i.e.*, used or bloated). By doing so, we can identify if the dependency is used or not, and take appropriate actions to remove bloated dependencies. We define the usage status of a dependency as follows:

*Dependency Usage Status.* The usage status of a dependency $d \in \mathcal{D}$ determines if $d$ is used or bloated *w.r.t.* to $p$, at a specific time of the development of $p$.

We implement dependency usage analysis in a software tool called DEPCLEAN [2]. DEPCLEAN builds a static call graph of the bytecode calls between the class members of a compiled MAVEN project and its dependencies. To study the distinctive aspects regarding the usage status of all dependencies in the dependency tree of artifacts in the MAVEN ecosystem, we introduce a new data structure, called the Dependency Usage Tree (DUT) as follows:

*Dependency Usage Tree.* The DUT of a project $p$, defined as $\text{DUT}_p = (\mathcal{V}, \mathcal{E}, \nabla)$, is a tree, whose nodes are the same as the MAVEN dependency for $p$ and which edges are all of the $(p, p_i)$, for all nodes $p_i \in \text{DUT}_p$. A labeling function $\nabla$ assigns each edge one of the following six dependency usage types: $\nabla : \mathcal{E} \rightarrow \{\text{ud, ui, ut, bd, bi, bt}\}$ such that:

$$\nabla(\langle p, d \rangle) = \begin{cases} \text{ud}, & \text{if } d \text{ is used and it is directly declared by } p \\ \text{ui}, & \text{if } d \text{ is used and it is inherited from a parent of } p \\ \text{ut}, & \text{if } d \text{ is used and it is resolved transitively by } p \\ \text{bd}, & \text{if } d \text{ is bloated and it is directly declared by } p \\ \text{bi}, & \text{if } d \text{ is bloated and it is inherited from a parent of } p \\ \text{bt}, & \text{if } d \text{ is bloated and it is resolved transitively by } p \end{cases}$$

Figure 3.2 shows an hypothetical example of DUT of a project $p$. Suppose that $p$ directly calls two sets of instructions in the direct dependency $d_1$ and the transitive dependency $d_6$. Then, the subset of instructions called in $d_1$ also calls

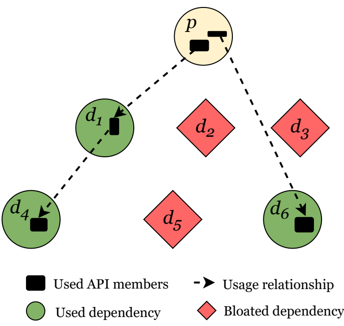



Figure 3.2: Dependency usage tree of used and bloated dependencies corresponding to the dependency tree presented in Figure 3.1.

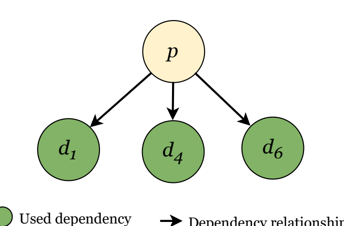



Figure 3.3: Debloated dependency tree after removing bloated dependencies with DEPCLEAN, based on the DUT of Figure 3.2.

instructions in $d_4$. In this case, the dependencies $d_1$, $d_4$, and $d_6$ are used by $p$, while dependencies $d2$, $d3$, and $d5$ are bloated dependencies. For a MAVEN project, DEPCLEAN constructs a DUT at build time and returns a report with the usage status of each individual dependency.

Although bloated dependencies are present in the dependency tree of software projects, bloated dependencies are useless and, therefore, developers should consider removing them. In the next section, we discuss the approach implemented in DEPCLEAN to remove bloated dependencies.

### 3.2.3 Bloat removal

A challenge when addressing bloated dependencies is to remove them from the project without compromising the build's success. Our solution relies on the existing MAVEN dependency handling mechanisms to remove and exclude bloated dependencies `pom.xml` files [159]. DEPCLEAN generates as output a variant of the `pom.xml` file with all the bloated dependencies removed. DEPCLEAN addresses both direct and transitive dependencies by modifying the XML entry corresponding to the bloated dependency. Listing 3.2 shows an excerpt of the diff of such a change in the `pom.xml` file for the example presented in Listing 3.1. Note that, in MAVEN, there is two ways to remove bloated dependencies:

(i) If the bloated dependency is explicitly declared in the `pom.xml`, then we remove its declaration clause directly (lines 12 to 19 in Listing 3.2);

(ii) If the bloated dependency is induced transitively from a direct dependency, then we exclude it from the dependency tree (lines 5 to 10 in Listing 3.2). This

```
1  <dependencies>
2    <dependency>
3      <groupId>org.d1</groupId>
4      <artifactId>d1</artifactId>
5  +   <exclusions>
6  +     <exclusion>
7  +       <groupId>org.d5</groupId>
8  +       <artifactId>d5</artifactId>
9  +     </exclusion>
10 +   <exclusions>
11   </dependency>
12 -   <dependency>
13 -     <groupId>org.d2</groupId>
14 -     <artifactId>d2</artifactId>
15 -   </dependency>
16 -   <dependency>
17 -     <groupId>org.d3</groupId>
18 -     <artifactId>d3</artifactId>
19 -   </dependency>
20 +   <dependency>
21 +     <groupId>org.d6</groupId>
22 +     <artifactId>d6</artifactId>
23 +   </dependency>
24 </dependencies>
```

Listing 3.2: Transformations peformed in the pom.xml file of Listing 3.1 to remove the bloated dependencies $d_2$, $d_3$, and $d_5$.

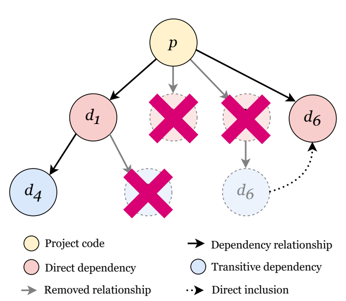



Figure 3.4: Transformations in the dependency tree of $p$ as a result of the changes in the pom.xml file indicated in Listing 3.2.

exclusion consists in adding an `<exclusion>` clause inside a direct dependency declaration entry, specifying the `groupId` and `artifactId` of the transitive dependency to be excluded. Excluded dependencies are not added to the classpath of the compiled artifact by way of the dependency in which the exclusion was declared.

Figure 3.3 shows the result of the modified dependency tree after using DEPCLEAN to remove bloated dependencies. Figure 3.4 illustrates the transformations made to the dependency tree to reach this state. Note that the transitive dependency $d_6$ was included as a direct dependency in the `pom.xml` (lines 20 to 23) because it is actually used by $p$, but the direct dependency $d_3$ from which it is induced is bloated and therefore removed. It is worth mentioning that during this removal process, DEPCLEAN does not perform any modifications to the source code, compiled bytecode, or configuration files in the project under analysis. DEPCLEAN is specifically designed to be non-invasive for the project, ensuring that it does not modify the build process while performing its debloating operations. The details about this procedure are described in Algorithms 1 and 2 in **??**.

### 3.2.4 Debloating assessment

Assessing the impact of removing bloated dependencies is crucial to ensure that the project build remains unaffected. It is equally important that the debloating process aligns with the project's requirements and makes sense from a practical standpoint. We use DEPCLEAN to perform two types of assessments: a large-scale quantitative analysis of dependency bloat in the Maven Central repository, and a qualitative analysis of bloated dependencies in 30 MAVEN projects involving developers.

The quantitative assessment consists in measuring the amount of dependencies that can be removed. For this we leverage the MDG from our previous research [8] to collect and analyze a large set of artifacts from Maven Central. We download the JAR files of all the selected artifacts and their `pom.xml` files. We resolve all their direct and transitive dependencies to our local repository and compute the usage status of all dependency relationships for each artifact using DEPCLEAN. We report the collected metrics and analyze how the specific reuse strategies of the MAVEN package management system relates to the existence of software bloat.

The qualitative assessment consists in evaluating the relevance of the removal of bloated dependencies in software projects. For this we systematically select 30 notable open-source projects hosted on GitHub and conduct an analysis of dependency bloat. For each project, we use DEPCLEAN to analyze the dependency tree and build the project with the debloated `pom.xml` file. If the project builds successfully, we propose a corresponding change to the developers in the `pom.xml` file in the form of a pull request. We engage developers in discussions regarding the value of each pull request on GitHub and gather their feedback. Note that although the submitted pull requests contain a small modification in the `pom.xml`, the amount of bloated code removed is significant.

DEPCLEAN operates under the premise that a bloated dependency at a given time will consistently remain bloated, hence it makes sense to remove it. We further explore the validity of this assumption in the context of Java projects. To do so, we performed a longitudinal study of bloated dependencies and analyze how the usage status of dependencies evolves over time, from used to bloated, or vice versa. Our empirical assessment shows that our hypothesis holds: the large majority of the bloated dependencies stay bloated in all subsequent versions of the dependency trees of the studied projects.

### 3.2.5 Key insights

We use DEPCLEAN to analyze the 723,444 dependency relationships of 9,639 artifacts hosted in Maven Central. Our findings indicate that 75.1 % of these dependen-

cies are bloated (2.7 % are direct dependencies, 57 % are transitive dependencies, and 15.4 % inherited dependency relationships in `pom.xml` files). Based on these results, we distill two potential causes of bloat in the Java MAVEN ecosystem: 1) the cascade of bloated transitive dependencies induced by direct dependencies, and 2) the dependency heritage mechanism in multi-module MAVEN projects.

We supplement our quantitative investigation of bloated dependencies with a comprehensive qualitative analysis of 30 popular Java projects. We use DEPCLEAN to examine the dependency trees of these projects and submit the derived results as pull requests on GitHub for evaluation by developers. Our results indicated that developers are willing to remove bloated-direct dependencies: 16 out of 17 answered pull requests were accepted and merged by the developers in their codebase. On the other hand, we find that developers tend to be skeptical about excluding bloated-transitive dependencies: 5 out of 9 answered pull requests were accepted. Overall, the feedback from developers reveals that the removal of bloated dependencies is clearly worth the additional analysis and effort.

We conduct a longitudinal analysis of dependency usage across 31,515 versions of MAVEN dependency trees in 435 Java projects. Our findings provide evidence of bloat stability: once bloated, 89.2 % of direct dependencies persist as bloated, emphasizing the importance of bloat removal. Furthermore, we present evidence indicating that developers expend unnecessary maintenance effort on bloated dependencies. Our qualitative examination of the origins of bloated dependencies uncovers that the primary contributing factor to this form of software bloat is the addition of dependencies at the early stages of the project development.

**Summary of Contribution #1**

We conduct a systematic, large-scale study of bloated dependencies in the MAVEN ecosystem. We implement a tool called DEPCLEAN, designed to automatically detect and remove bloated dependencies in MAVEN projects. We found empirical evidence that dependency bloat is widespread among Java artifacts within the Maven Central repository. Our study is the first to measure the extent of dependency bloat on a large scale and perform a qualitative assessment of the opinion of developers regarding the removal of bloated dependencies. We found that developers are willing to remove bloated dependencies to a large extend. Moreover, we demonstrate that a dependency, once bloated, it is likely to stay bloated in the future.

☞ This contribution is presented in Research Papers II [2] and III [3].

## 3.3 Contribution #2: Specializing Used Dependencies

Our second contribution focuses on advancing the state-of-the-art of dependency tree reduction by introducing an innovative technique that specialized dependencies specifically to a project's requirements. We implement this technique in a tool called DEPTRIM, which systematically identifies and removes unused classes across the dependencies of a MAVEN project. After debloating, DEPTRIM repackages the used classes into a specialized version of each used dependency, and substitutes the original dependency tree of a project with this specialized variant. This approach enables building a minimal project binary containing only the code that is relevant to the project, thereby optimizing resource utilization, improving build performance, and reducing potential security risks associated with unused code in third-party dependencies.

### 3.3.1 Novel concepts

We introduce the concept of specialized dependencies and specialized dependency trees as follows:

*Specialized Dependency.* A dependency is said to be specialized with respect to a project if all the classes within the dependency are used by the project, and all unused classes have been identified and removed. Consequently, there is no class file in the API of a specialized dependency that is unused, directly or indirectly, by the project or any other dependency in its dependency tree.

*Specialized Dependency Tree.* A specialized dependency tree is a dependency tree where at least one dependency is specialized and the project still correctly builds with that dependency tree. This means that in at least one of the used dependencies, unused classes have been identified and removed. A specialized dependency tree may be one of the following two types:

- *Totally Specialized Tree* (TST): A dependency tree where all used dependencies are specialized and the project build is successful.
- *Partially Specialized Tree* (PST): A dependency tree with the largest possible number of specialized dependencies, such that the project build is successful.

We implement a tool called DEPTRIM that automatically generates a TST or PST for MAVEN projects. DEPTRIM systematically identifies the required subset of classes in each dependency that is necessary to build the project. The specialized dependencies are repackaged and incorporated into the project's dependency tree,

yielding a tailored dependency tree specific to the project's needs and requirements.

### 3.3.2 Bloat detection

In order to detect bloat in used dependencies, DEPTRIM relies on static analysis to determine their API usage from the project compiled sources. This process involves constructing a static call graph by utilizing the compiled dependencies resolved by MAVEN and the compiled project sources. The call graph is generated using the bytecode class members of the project as entry points. By leveraging the API usage information from the static call graph, DEPTRIM can directly infer and report class usage information from the bytecode, without the need to load or initialize classes. The resulting report captures the dependencies, classes, and methods that are actually used by the project, *i.e.*, those that are reachable via static analysis. This information is stored in data structure in order identify the minimal set of classes in each dependency that are necessary to successfully build the project.

Recalling the example of debloated dependency tree presented in Figure 3.3, we observe that the debloated dependency tree of project $p$ uses a subset of the classes in dependencies $d_1$, $d_2$, and $d_3$ (see Figure 3.5). Therefore, these dependencies could be specialized with respect to $p$, by detecting and removing the unused classes.

The completeness of the call graphs is crucial for successful dependency specialization. If a necessary class member cannot be reached through static analysis, DEPTRIM considers it unused and proceeds to remove it in a subsequent phase. To overcome this limitation, DEPTRIM employs state-of-the-art static analysis techniques of Java bytecode to capture invocations between classes, methods, fields, and annotations (from the project and its direct and transitive dependencies). This comprehensive approach ensures accurate detection of used and unused classes, enabling the creation of a specialized dependency tree tailored to the project's requirements.

It is worth mentioning that that DEPTRIM also analyzes the constant pool of class files to capture dynamic invocations from string literals, such as when loading a class using its fully qualified name via reflection. The constant pool is a data structure in Java class files that stores constants and symbolic references, including literals and external references. By examining the constant pool, DEPTRIM can identify instances of dynamically invoked classes, ensuring a more precise and thorough dependency analysis. Moreover, the integration of DEPTRIM within the

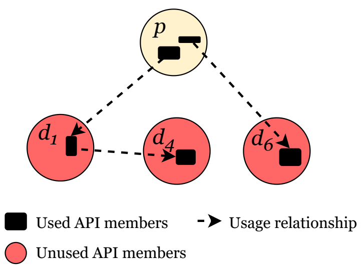



Figure 3.5: Used and unused API members in the debloated dependency tree from Figure 3.3.

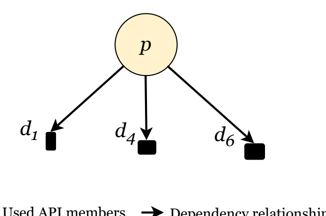



Figure 3.6: Specialized dependency tree after removing unused API members from Figure 3.5.

MAVEN build lifecycle further enhances the tool's usability, making it a seamless and convenient solution for developers to optimize their project dependencies.

### 3.3.3 Bloat removal

DEPTRIM receives as input a debloated dependency tree, such as the ones generated by DEPCLEAN. If the provided dependency tree is not debloated, DEPTRIM determines which dependencies are bloated (*i.e.*, there is no path from the project bytecode toward any of the class members in the unused dependencies), and removes them from the original `pom.xml`. Next, DEPTRIM proceeds to remove the unused classes within non-bloated dependencies by analyzing the call graph of static bytecode calls . Any class file from the dependencies that is not present in the call graph is deemed unreachable and removed. Once all the unused class files in a dependencies are removed, DEPTRIM qualifies the dependency tree as specialized.

DEPTRIM downloads, unzips, and removes the unused compiled classes directly from the project dependencies at build time (*i.e.*, during the MAVEN `package` phase). Moreover, to facilitate reuse, DEPTRIM deploys each specialized dependency in the local MAVEN repository along with its `pom.xml` file and corresponding `MANIFEST.MF` metadata. After specializing each non-bloated dependency, DEPTRIM produces a specialized version of the project's dependency tree. For example, Figure 3.6 shows the specialized dependency tree after removing unused classes from the dependencies $d_1$, $d_2$, and $d_3$ as presented in Figure 3.5. In addition, DEPTRIM produces a variant of the `pom.xml` file that removes the bloated dependencies and points to the specialized dependencies instead of their original versions This results in a TST or a PST for the project.

The output of the DEPTRIM is a set of specialized `pom.xml` files representing

the dependencies of the project. These files encompass all the essential bytecode and resources required for sharing and reusing functionalities among the packages within the dependency tree. In particular, DEPTRIM takes care of keeping the classes in dependencies that may not be directly instantiated by the project, but are accessible from the used classes in the dependencies, with regard to the project. The details about this procedure are described in Algorithms 1 in **??**.

### 3.3.4 Debloating assessment

To assess the debloated dependency tree, DEPTRIM builds the totally specialized dependency tree (TST or PST) of the project. All specialized dependencies replace their original version in the project `pom.xml`. Then, in order to validate that the specialization did not remove necessary bytecode, DEPTRIM builds the project, *i.e.* its sources are compiled and its tests are run. If the build is a SUCCESS, DEPTRIM returns this TST.

In cases where the build with the TST fails, DEPTRIM proceeds to build the project with one specialized dependency at a time. Thus, rather than attempting to improve the soundness of the static call graph, which is proven to be challenging in Java [161], DEPTRIM performs an exhaustive search of the dependencies that are unsafe to specialize. At this step, DEPTRIM builds as many versions of the dependency tree as there are specialized dependencies, each containing a single specialized dependency. DEPTRIM attempts to build the project with each of these single specialized dependency trees. If the project build is successful, DEPTRIM marks the dependency as safe to be specialized. In case the dependency is not safe to specialize, DEPTRIM keeps the original dependency entry intact in the specialized `pom.xml` file. Finally, DEPTRIM constructs a partially specialized dependency tree (PST) with the union of all the dependencies that are safe to be specialized. Then, the project is built with this PST to verify that the build is successful. If all build steps pass, DEPTRIM returns this PST.

### 3.3.5 Key insights

We use DEPTRIM to generate specialized dependency trees for 30 notable open-source Java projects. DEPTRIM effectively analyzes 35,343 classes across 467 dependencies in these projects. For 14 projects, it generates a dependency tree where all compile its dependencies are effectively specialized. For the remaining 16 projects, DEPTRIM produces a dependency tree that includes all dependencies that can be specialized without breaking the build, while leaving the others unmodified. DEPTRIM specializes 86.6 % of the dependencies, removing 47.0 % of the unused

classes from those dependencies. The specialized dependencies are deployed locally as reusable JAR files. For each project, DEPTRIM generates a specialized version of the `pom.xml` file, replacing the original dependencies with specialized ones, ensuring that the project continues to build correctly.

We perform a novel assessment of the ratio of dependency classes compared to project classes, based on actual class usages. We compute this ratio for the 30 original studied projects and found that it is 8.7×, which is evidence of the massive impact of code reuse in the Java ecosystem. We found that it is possible to decrease this ratio of dependency classes to project classes through dependency specialization with DEPTRIM, from 8.7× to 4.4×. This result confirms the relevance of our approach in substantially reducing the share of third-party classes in Java projects.

**Summary of Contribution #2**

We advance the state-of-the-art for dependency tree reduction through the implementation of a specialization technique that tailors individual dependencies to the specific requirements of a project. We implement an automated tool, DEPTRIM, that analyses third-party dependencies of a MAVEN project to remove the unused classes. DEPTRIM repackages the dependencies to create a specialized version of the dependency tree at build time. We use DEPTRIM to successfully specialize the dependency tree of 14 projects in its entirety, and 16 partially, reducing the number of third-party classes by 47.0 %. We found that our specialization technique enables a reduction in the ratio of project classes to dependency classes by a factor of two.

☞ This contribution is presented in Research Paper VI [6].

## 3.4 Contribution #3: Debloating With Respect to Clients

Our third contribution goes one step further than any previous work on software debloating and investigates how debloating Java libraries impacts the clients of these libraries. We propose coverage-based debloating, a novel technique to debloat projects based on coverage information collected at runtime. We implemented this technique in a tool called JDBL, which precisely captures what parts of a project and its dependencies are used when running with a specific

workload. The goal is to determine the ability of dynamic analysis via coverage at capturing the behaviors that are relevant for the clients of the debloated libraries.

### 3.4.1 Novel concepts

In this contribution, MAVEN projects are referred to as libraries, and the project that reuses the library are called clients. We introduce a set of novel concepts necessary for debloating libraries *w.r.t.* clients as follows:

*Input Space.* The input space of a compiled MAVEN project is the set of all valid inputs for its public Application Programming Interface (API) that can be executed by a client.

MAVEN projects provide API members, abstracting implementation details to facilitate external reuse. Libraries generally provide public API members for external reuse. However, there exist other dynamic reuse mechanisms that can be utilized by Java clients (*e.g.*, through reflection, dynamic proxies, or the use of unsafe APIs). An effective way to determine which API members are reused is trough the execution of a workload.

*Project Workload.* A workload is a set of valid inputs belonging to the input space of a compiled MAVEN project.

Workloads play a crucial role in software debloating tasks that involve performing dynamic analysis. For instance, workloads are employed to identify unique execution paths in software applications, similar to those performed for profiling and observability tasks. These techniques focus on utilizing monitoring tools to analyze the application's response to various workloads at run-time, ultimately contributing to a more efficient and streamlined software system. In this context, by examining the application's response to different workloads, it is possible to generate execution traces.

*Execution Trace.* An execution trace is a sequence of calls between bytecode instructions in a compiled MAVEN project, obtained as a result of executing the project with a valid workload.

Given a valid workload for a project, one can obtain dynamic information about the program's behavior by collecting execution traces. We consider a trace as a sequence of calls, at the level of classes and methods, in compiled Java classes. These traces include the bytecode of the project itself, as well as the classes and methods in third-party libraries.

*Coverage-Based Debloating.* Given a project and an execution trace collected when running a specific workload on the project, coverage-based debloating consists of removing the bytecode constructs that are not covered when running the workload. Coverage-based debloating takes a project and workload as input

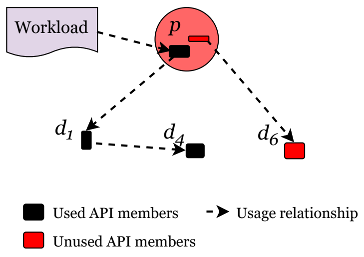



Figure 3.7: Used and unused API members in the dependency tree of Figure 3.6. Note that the usage status is *w.r.t.* the supplied workload.

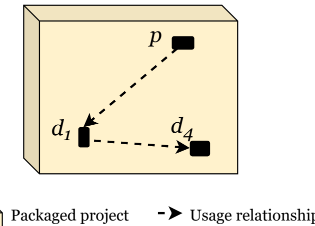



Figure 3.8: Debloated project from Figure 3.7. Only the used API members of the project and its dependencies are packaged.

and produces a valid compiled Java project as output. The generated debloated project is executable and has the same behavior as the original, modulo the workload.

### 3.4.2 Bloat detection

JDBL collects a set of coverage reports that capture the set of dependencies, classes, and methods actually used during the execution of the Java project. The coverage collection phase receives two inputs: a compilable set of Java sources, and a workload, *i.e.*, a collection of entry-points and resources necessary to execute the compiled sources. The workload can be a set of test cases or a reproducible production workload. The coverage collection phase outputs the original, unmodified, bytecode and a set of coverage reports that account for the minimal set of classes and methods required to execute the workload. The collection of accurate and complete coverage is essential for coverage-based debloating

### 3.4.3 Bloat removal

The goal of the bytecode removal phase is to eliminate the methods, classes, and dependencies that are not used when running the project with the workload. This procedure is based on the coverage information collected during the coverage collection phase. The unused bytecode instructions are removed in two passes. First, the unused class files and dependencies are directly removed from the `classpath` of the project. Then, the procedure analyzes the bytecode of the classes that are covered. When it encounters a method that is not covered, the body of the method is replaced to throw an `UsupportedOperationException`. We

choose to throw an exception instead of removing the entire method to avoid JVM validation errors caused by the nonexistence of methods that are implementations of interfaces and abstract classes.

Capturing the complete coverage of the classes that are necessary for executing a workload is critical for bloated code removal. Failure to achieve this could result in a debloated project that either fails to compile or, even worse, causes runtime errors when client projects use debloated libraries. To collect precise coverage information, we harness the diversity of code coverage tool implementations [162] and the dynamic logging capabilities of the JVM. We process and aggregate the coverage reports from JaCoCo, JCov, Yajta, and the JVM class loader. A class is deemed covered if it is reported as used by at least one of these tools, ensuring a comprehensive assessment of required classes for successful debloating. The details about this procedure are described in Algorithms 1 in **??**.

### 3.4.4 Debloating assessment

We analyze the impact of debloating Java libraries on their clients. This analysis is relevant since we focus on debloating open-source libraries, which are primarily designed for reuse in client applications. Moreover, this particular analysis offers additional insights into the validity of the coverage-based debloating technique and the effectiveness of JDBL. To validate the debloating from the clients' perspective, we conduct a two-layered assessment: a syntactic evaluation a semantic evaluation of the clients. By performing these analysis, we can guarantee that the debloated libraries preserve their functionality and compatibility, thus assessing the validity of our debloating technique.

For syntactic assessment, we verify that the clients still compile when the original library is replaced by its debloated version. We check that JDBL does not remove classes or methods in libraries that are necessary for the compilation of their client. As illustrated in Figure 3.9, we first check that the client uses the library statically in the source code. To do so, we statically analyze the source code of the clients. If there is at least one element from the library present in the source code of a client, then we consider the library as statically used by the client. If the library is used, we inject the debloated library and build the client again. If the client successfully compiles, we conclude that JDBL debloated the library while preserving the useful parts of the code that are required for compilation.

A debloated library stored on disk is of little use compared to a debloated library that provides the behavior expected by its clients. Therefore, we also need to determine if JDBL preserves the functionalities that are necessary for the clients. As illustrated in Figure 3.9, we first execute the test suite of the client with the

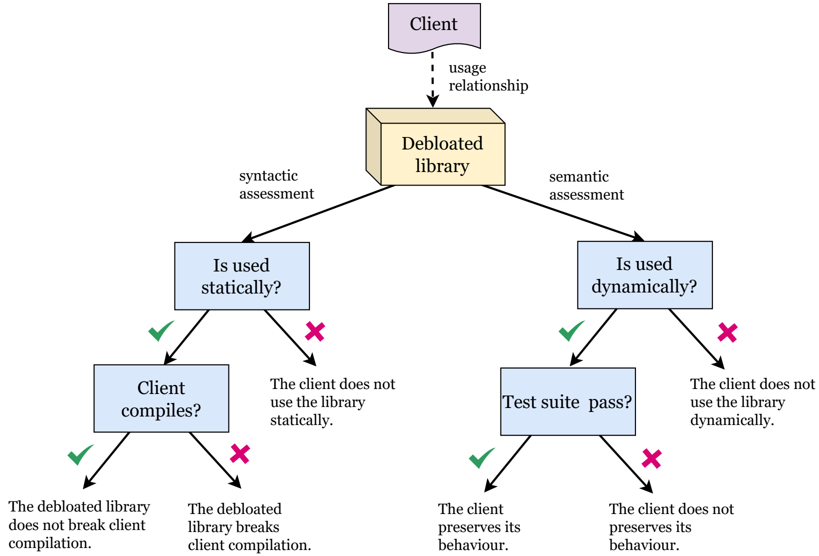



Figure 3.9: Experimental procedure to assess the impact of debloating a library on the clients that use a subset of its functionalities.

original version of the library. We check that the library is covered by at least one test of the client. If this is true, we replace the library with the debloated version and execute the test suite again. If the test suite behaves the same as with the original library, we conclude that JDBL is able to preserve the functionalities that are relevant for the clients.

Building a sound dataset of clients that execute the libraries is challenging. To ensure the validity of this protocol, we perform additional checks on the clients. All the clients have to use at least one of the debloated libraries. We only consider the clients that either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the library (static or dynamic usage). The clients that statically use the library serve as the study subjects for the syntactic assessment. The clients that have at least a test that reaches the debloated library serve as the study subjects for the semantic assessment.

### 3.4.5 Key insights

We perform the largest empirical validation of Java debloating in the literature involving 354 libraries and 1,354 clients that use these libraries. We evaluate JDBL based on an original experimental protocol that assesses the impact of coverage-based debloating on the libraries behavior, their size, as well as on their clients.

Our results indicate that JDBL can reduce 68.3 % of the bytecode size and that 211 (69.9%) debloated libraries still compile and preserve their original behaviour according to the tests.

We evaluate the usefulness of debloated libraries with respect to their client applications. Our findings reveal that 81.5 % of the clients can successfully compile and execute their test suites when replacing the corresponding dependency with a debloated version of the library. These results demonstrate that the combination of multiple coverage tools is effective in accurately capturing the code utilized at runtime, ultimately showcasing the practicality of debloated libraries for client applications.

**Summary of Contribution #3**

We propose a novel coverage-based debloating technique for Java applications. This technique addresses one key challenge of debloating techniques based on dynamic analysis: gathering precise and comprehensive coverage information that comprises the minimal set of classes and methods required to execute a program under a given workload. We conducted the most extensive empirical validation of the applicability of a software debloating technique in the literature, involving 354 libraries and 1,354 client applications. Our results provide evidence of the massive presence of code bloat in those libraries and the usefulness of our techniques to mitigate this phenomenon.

☞ This contribution is presented in Research Paper IV [4].

## 3.5 Contribution #4: Reproducible Research

Reproducible research stands as a vital cornerstone of the scientific endeavor. It plays an essential role in ensuring the validity and reliability of the research findings. Given its importance, our fourth contribution focuses on the tools and datasets that are part of the contributions of this thesis. These resources are of utmost importance as they enable other researchers to reproduce the findings and conclusions of our studies, validate the results, and build upon our work in future research endeavors. By providing open access to the datasets and tools used, we aim to promote transparency, accountability, and reproducibility for the best of science.

### 3.5.1 Software tools

Contributions **C1**, **C2**, and **C3** in this thesis encompass a software tool engineered to implement their respective debloating techniques In the following, discuss the technical challenges associated with each tool, emphasizing their roles in fostering reproducible research and advancing the field of software debloating in Java.

#### DEPCLEAN

DEPCLEAN is implemented in Java as a Maven plugin that extends the `maven-dependency-analyzer` [163], which is actively maintained by the Maven team and officially supported by the Apache Software Foundation. For the construction of the dependency tree, DEPCLEAN relies on the `copy-dependencies` and `tree` goals of the `maven-dependency-plugin`. Internally, DEPCLEAN relies on the ASM [164] library to visit all the class files of the compiled projects in order to register bytecode calls towards classes, methods, fields, and annotations among MAVEN artifacts and their dependencies. For example, it captures all the dynamic invocations created from class literals by parsing the bytecodes in the constant pool of the classes. DEPCLEAN defines a customized parser that reads entries in the constant pool of the class files directly, in case it contains special references that ASM does not support. This allows the plugin to statically capture reflection calls that are based on string literals and concatenations. Compared to `maven-dependency-analyzer`, DEPCLEAN adds the unique features of detecting transitive and inherited bloated dependencies, and producing a debloated version of the `pom.xml` file.

DEPCLEAN is open-source and reusable from Maven Central. DEPCLEAN is a well-established project that adheres to sound engineering principles such CI/CD, static analysis to ensure high code quality, and rigorous unit and integration testing. it has been used to remove bloated dependencies in both open-source and close-source projects, as well as for research purposes [40, 50, 165, 3]. As per January 2023, DEPCLEAN has 3.2 K lines of Java code, 394 commits, 12 contributors, and 155 stars [166] on GitHub. We have done 9 releases to integrate feedback from users and evolve with the new features of Java and MAVEN (*e.g.*, to achieve compatibility with Java records and other MAVEN plugins). Its source code is available at `https://github.com/castor-software/depclean`.

#### DEPTRIM

DEPTRIM is implemented in Java as a MAVEN plugin that can be integrated into a project as part of the build pipeline, or be executed directly from the command

line. This design facilitates its integration as part of the projects' CI/CD pipeline, leading to specialized binaries for deployment. At its core, DepTrim reuses the state-of-the-art static analysis of DepClean, located in the `depclean-core` module. DepTrim adds unique features to this core static Java analyzer by modifying the bytecode within dependencies based on usage information gathered at compilation time, which is different from the complete removal of unused dependencies performed by DepClean. It uses the ASM Java bytecode analysis library to build a static call graph of class files of the compiled projects and their dependencies. The call graph registers usage towards classes, methods, fields, and annotations. For the deployment of the specialized dependencies, DepTrim relies on the `deploy-file` goal of the official `maven-deploy-plugin` from the Apache Software Foundation. For dependency analysis and manipulation, DepTrim relies on the `maven-dependency-plugin`. DepTrim provides dedicated parameters to target or exclude specific dependencies for specialization, using their identifier and scope.

DepTrim is open-source and reusable from Maven Central. As per January 2023, DepTrim has 1.1 K lines of code Java code, 119 commits, and 3 contributors. Its source code is publicly available at https://github.com/castor-software/deptrim.

**JDBL**

The core implementation of JDBL consists in the orchestration of mature code coverage tools and bytecode transformation techniques. The coverage-based debloating algorithm is integrated into the different Maven building phases. JDBL gathers direct and transitive dependencies by using the official `maven-dependency-plugin` with the `copy-dependencies` goal. This allows JDBL to manipulate the project's `classpath` in order to extend code coverage tools at the level of dependencies. As with DepClean and DepTrim, we rely on ASM [164] a lightweight, and mature Java bytecode manipulation and analysis framework for the bytecode analysis, the detection of bloated classes, and the whole bytecode removal phase. The instrumentation of methods and the insertion of probes for usage collection are performed by integrating JaCoCo, JCov, Yajta, and the JVM class loader within the Maven build pipeline.

JDBL is implemented as a multi-module Maven project with a total of 5.0 K lines of code written in Java. JDBL is designed to debloat single-module Maven projects. It can be used as a Maven plugin that executes during the Maven `package` phase. Thus, JDBL is designed with usability in mind: it can be easily invoked within the Maven build life-cycle and executed automatically, no ad-

Table 3.1: Reproducible datasets for each of the appended research papers.

| RESEARCH PAPER | DATASET URL ON GITHUB |
|---|---|
| I | `https://github.com/cesarsotovalero/msr-2019` |
| II | `https://github.com/castor-software/depclean-experiments` |
| III | `https://github.com/castor-software/longitudinal-bloat` |
| IV | `https://github.com/castor-software/jdbl-experiments` |
| V | `https://github.com/chains-project/ethereum-ssc` |
| VI | `https://github.com/castor-software/deptrim-experiments` |

ditional configuration or further intervention from the user is needed. To use JDBL, developers only need to add the MAVEN plugin within the build tags of the `pom.xml` file. The source code of JDBL is publicly available on GitHub, with binaries published in Maven Central. More information on JDBL is available at `https://github.com/castor-software/jdbl`.

### 3.5.2 Reproducible datasets

All research papers in this thesis include a reproducible dataset specifically designed for transparent and reliable research. Table 3.1 shows the URL on GitHub of the companion dataset for each research paper. The datasets comprise a diverse range of technologies employed for data collection, analysis, and manipulation (*e.g.*. Shell scripts, Java artifacts, R and Python notebooks, Docker containers, CSV files, and JSON files). These datasets allow other researchers to independently verify the results obtained. It also enables the development of new methods and techniques that can be applied to the same dataset.

In addition to the datasets that come with each research paper, the author of this thesis contributed to making available two additional datasets in the Data Showcase track of the *Proceedings of the IEEE/ACM International Conference on Mining Software Repositories*:

- The Maven Dependency Graph: a Temporal Graph–Based Representation of Maven Central [8].
- DUETS: A Dataset of Reproducible Pairs of Java Library–Clients [13].

These datasets play a valuable role in promoting reproducible research in the field of Java dependency analysis. The technical challenges and benefits of both datasets for the contribution of this thesis are discussed below.

### MDG

The Maven Dependency Graph (MDG) is a graph-oriented open-source dataset that characterizes the artifacts present in Maven Central and their associated dependency relationships. It represents a snapshot of the Maven Central Repository from September 6, 2018. The MDG is implemented as a Neo4j graph database and contains a total of 2.4 M artifacts and 9.7 M dependency relationships among them. The MDG aims at enabling the Software Engineering community to conduct large-scale empirical studies on Maven Central. The dataset is accessible on Zenodo at `https://zenodo.org/record/1489120`.

The author of this thesis contributed to the creation of this dataset, including engaging in discussions leading to its technical implementation and development. The dataset has found utility in the author's Research Papers I [1] and II [2]. Furthermore, the dataset has been effectively reused by other researchers [14, 167, 168, 169].

### DUETS

The DUETS dataset consists of a collection of single-module Java libraries, which build can be successfully reproduced with MAVEN (*i.e.*, all the test pass and a compiled artifact is produced as a result of the build), and Java clients that use those libraries. DUETS includes 94 different libraries, with a total of 395 versions, as well as 2,874 clients. The construction of this dataset involved filtering 147 K Java projects and analyzing 34 K `pom.xml` files in order to identify relevant libraries and clients that reuse version of these libraries. We take a special care to build a dataset for which we ensure that both the library and the clients have a passing test suite. The dataset is accessible on Zenodo at `https://zenodo.org/record/4723387`.

The contributions in this thesis involve executing software tools on compilable and testable software projects, which we provide with DUETS. We use the DUETS dataset for the evaluation of debloating techniques that rely on both static and dynamic analysis. The dataset has found utility in the author's Research Papers III [3], IV [4], and VI [6]. Furthermore, the dataset has been effectively reused by other researchers seeking to explore the effects of API changes on clients of various libraries [170, 171].

**Summary of Contribution #4**

We contribute three new open-source research tools to the field of debloating Java dependencies: DEPCLEAN, DEPTRIM, and JDBL. Each research paper contributes experimental data and makes the results open. By sharing our datasets and making this information widely accessible, we aim to facilitate collaboration and knowledge sharing within the scientific community. Furthermore, we contribute two large Data Showcase datasets of Java dependencies. Moreover, our contributions include two extensive Data Showcase datasets of Java dependencies, which are essential for researchers and practitioners seeking to explore various aspects of software engineering. These datasets have been meticulously curated and pre-processed to ensure their quality and usability, and we hope that they will be valuable resources for the community for years to come. By following these reproducible research principles, we aim to foster collaboration and trust in the scientific community and to advance the field of software debloating.

☞ This contribution is present in Research Papers I [1], II [2], III [3], IV [4], V [5], and VI [6].

## 3.6 Summary

In this chapter, we presented and discussed the contributions of this thesis. First, we elucidated the terminology and concepts of dependency management in the MAVEN ecosystem. Further, we described the technical challenges pertaining to debloating which were targeted in each of our contributions, namely bloat detection, bloat removal, and debloat assessment.

The first contribution focuses on removing bloated dependencies. We found that $75\,\%$ of the dependency relationships in Maven Central are bloated, and that developers are willing to remove bloated dependencies: we removed $140$ bloated dependencies via merged pull requests in mature Java projects. The second contribution focuses on specializing the remaining used dependencies in the dependency tree of Java projects. We focus on reducing the share of third-party classes across the dependencies. Our technique removes $47.0\,\%$ of classes in $30$ projects, reducing the project classes to dependency classes ratio from $8.7\times$ to $4.4\times$. The third contribution is centered around the process of debloating Java libraries by removing features that are actually not used at runtime by their clients. We found that $81.5\,\%$ of the clients were able to successfully compile and

execute their test suite using the debloated library. The fourth contribution focuses on the technical challenges addressed by the three new open-source research tools that contributed to the field of debloating in this thesis and describe the two large datasets of Java dependencies employed in our research studies. We made our research tools and results openly accessible and reproducible, aiming to foster collaboration in the scientific community and advance the field of software debloating.

# Chapter 4

# Conclusions and Future Work

*"Lättare sagt än gjort."*

— svenskt ordspråk

GIVEN the ever-increasing complexity of software systems, the research field of software debloating is still in its early stages of development, with many challenges and opportunities for further investigations. In this chapter, we summarize the results of the three key technical contributions presented in this thesis: removing bloated dependencies, specializing used dependencies, and debloating *w.r.t.* clients. Moreover, we offer an author's reflection on the particular challenges encountered when conducting research in the field of empirical software engineering. Finally, we discuss promising avenues for future studies and highlight the current challenges that should be overcome in order to facilitate the progress and adoption of software debloating techniques.

## 4.1 Key Experimental Results

In this thesis, we have focused on the design and implementation of software debloating techniques in the context of Java dependencies. We propose various techniques to address the following research problems: 1) the increasing practice of software reuse leading to the emergence of bloated dependencies in the Java ecosystem; 2) the existence of a large amount of bloated code in used dependencies; and 3) the lack of knowledge regarding the impact of debloating libraries for their clients. Our technical contributions are organized into three parts to target these three problems.

First, we focus on addressing the problem of dependency bloat in the MAVEN ecosystem We create the concept of "bloated dependencies" and propose an approach to detect and remove these dependencies. We implement this approach in

a practical software tool called DEPCLEAN [2]. We use DEPCLEAN to empirically study the pervasiveness of dependency bloat in the MAVEN ecosystem. Our results reveal that 2.7 % of directly declared dependencies, 15.4 % of inherited dependencies, and 57 % of transitive dependencies are bloated. Our longitudinal analysis of bloated dependencies shows that the usage status of such dependencies do not change over time [3], and that developers are willing to remove bloated dependencies when notified, as evidenced by the removal of 140 bloated dependencies in 30 open-source projects. Beyond academic recognition, DEPCLEAN has received positive feedback from developers for its ability to detect bloated dependencies in a variety of real-world projects. Overall, our experimental results highlight the importance of analyzing, maintaining, and testing configuration files and other software artifacts related to the management of third-party dependencies (*e.g.*, `pom.xml` files).

Second, we focus on the dependencies that are partially used by MAVEN projects. We propose a novel technique called "dependency specialization" to reduce the amount of third-party code in Java projects based on their actual usage [6]. We implement this dependency specialization technique in a tool called DEPTRIM, which automatically identifies the necessary subset of functionalities for each dependency and removes the rest, resulting in repackaged specialized dependencies. We use DEPTRIM to evaluate the effectiveness of our technique on 30 mature Java projects. Our results show that DEPTRIM successfully specializes 86.6 % of the dependencies in the projects without affecting its build, while dividing by two the amount of third-party code. Overall, our findings suggest that the specialization of dependencies is an effective approach to significantly reduce the share of third-party code in Java projects.

Third, we focus on investigating how debloating Java libraries impacts the clients of these libraries. We propose a novel technique for debloating, which we call "coverage-based debloating", that leverages code coverage information collected at runtime to detect and remove code bloat [4]. We implement this approach in a software tool called JDBL which relies on a combination of state-of-the-art Java bytecode coverage tools to precisely capture what parts of a project and its dependencies are used when running with a specific workload. With this information, JDBL automatically removes the parts that are not covered, in order to generate a debloated version of the project. We use JDBL to debloat 211 Java libraries in order to determine the ability of this technique at capturing the behaviors that are relevant for the clients of the debloated libraries The debloated versions are syntactically correct and preserve their original behavior according to the workload. We evaluate thi debloating approach on client projects that

either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the libraries that we debloat. Our results show that 81.5 % of the clients, with at least one test that uses the library, successfully compile and pass their test suite when the original library is replaced by its debloated version. This result constitutes the first empirical demonstration that debloating can preserve essential functionalities to successfully compile the clients of debloated libraries.

## 4.2 Reflections on Empirical Software Engineering Research

Empirical software engineering is a fascinating research field that encompasses the collection, analysis, and interpretation of data to improve software development practices [76]. The inherent complexities of software development, coupled with the challenges of collecting and analyzing large amounts of human and computer-generated data, make empirical software engineering research a challenging field. Throughout our contributions, we have embraced these challenges and have striven to address and overcome each of them as they arose.

One of the primary challenges has been finding useful datasets of software artifacts for our empirical experiments on debloating [172]. Collecting data of software development projects for this purpose is a daunting task, as it requires access to various software artifacts such as source code, build configuration files, and third-party dependencies [173]. Additionally, researchers must ensure that the data has been ethically collected, and is accurate, complete, and relevant to their research questions [174]. For instance, we investigated to what extent the number of bloated dependencies increases over time in software projects. To collect relevant data, we need to analyze a large number of open-source repositories of Java projects that are representative of the dependency management process in the MAVEN ecosystem and analyze their dependency trees over time at different releases. We encountered this task challenging as many repositories are out of date [175] and some dependencies cannot be resolved (*e.g.*, such as those dependencies that are hosted in private repositories and become inaccessible to the research community). However, we hope that leveraging new tools, such as bots to automate pull requests [176] will encourage developers to update dependencies and maintain their projects in an up-to-date state [177]. To further promote reproducibility in our research and support the broader software engineering community, we have invested significant effort in curating high-quality datasets of software artifacts that are readily available for other researchers to use. In this same spirit, the software engineering community has been actively promoting reproducible

research by offering publicly accessible datasets via the *Data Showcase* track at the *International Conference on Mining Software Repositories* (MSR). This initiative aims to encourage the sharing of high-quality datasets for software engineering research purposes. We are proud to have contributed to this effort throughout this thesis.

Another challenge of empirical software engineering research is finding sound metrics to evaluate the proposed tools and techniques [178, 179, 180]. When conducting our debloating experiments, we had to identify metrics that are valid and reliable for measuring the effectiveness of our proposed debloating techniques. For instance, in the case of our empirical evaluation of the debloating results of JDBL, our experiments focus on measuring the amount of code bloat removed in the debloated libraries at three different code granularity levels: methods, classes, and dependencies. However, we notice that most previous works in software debloating do not consider the code removed in third-party dependencies. Therefore, we had to assume that counting the number of completely removed third-party dependencies is a reasonable choice in this case. Overall, finding appropriate metrics in software engineering can be challenging, as some metrics may not exist previously and for those that already exist, it could be difficult to accurately use them in the context of some specific experiment. We hope that our original metrics will become beneficial to the research community exploring software debloating techniques.

One more challenge we have encountered is establishing a fair and realistic comparison of our techniques with other existing tools in the field. Research tools are often not available or the research experiments conducted are not reproducible [181]. For example, we encountered difficulties in finding available software debloating tools, as some are closed-source or no longer accessible. Upon contacting the authors of some existing tools, we faced challenges in executing them correctly due to specific configuration requirements. Additionally, certain experiments are designed for specific research environments, which complicates the process of comparing them in diverse contexts. In this regard, the use of Docker containers has been widely recognized as an effective way to promote reproducibility in scientific research [182]. Docker provides a self-contained environment that can be easily shared and replicated across different computing platforms. In order to contribute to this ongoing effort and foster a culture of reproducibility within the research community, we have made our software tools (DepClean, DepTrim, and JDBL) publicly available and reusable, providing an opportunity for other researchers to easily build upon our work and perform fair comparisons in future studies.

Last but not least, we have learned after working on tens of thousands of

open-source projects that it is hard to build and execute software in general [183]. This can be a challenging and time-consuming process, especially for large projects containing millions of lines of code and thousands of dependencies. For instance, while conducting our experiments on the software supply chain of the Ethereum Java clients Besu and Teku [5], we embraced the opportunities presented by their significant engineering complexity to further enhance our understanding of complex software systems (*e.g.*, at that moment, Besu was composed of 41 internal modules, containing 355 unique third-party dependencies provided by 165 distinct supplying organizations). Through our experience, we notice that studying projects with well-defined CI/CD pipelines can greatly simplify the building process, thereby saving time and effort for researchers that would otherwise be spent on manual configuration and integration. Moreover, sometimes when we were building and executing the software multiple times to collect sufficient data we found nondeterministic behaviors (*e.g.*, flaky tests [184], Heisenbugs [185], or non-atomic operations [186]). We believe that the existence of those engineering challenges when building and executing real-world software represents fundamental opportunities that contribute to the vibrant and dynamic nature of empirical software engineering research.

In summary, empirical software engineering research provides answers to the fundamental questions about the practice of software development. It is a thriving research field that holds promise for advancing our understanding of software development practices and improving the quality of software products [187]. Throughout our research journey, we have successfully tackled various challenges, including gathering valuable datasets, identifying suitable metrics, comparing our work *w.r.t.* other research tools, and building and executing software projects from public repositories on GitHub. These challenges, which are commonly encountered by researchers in the field, have served as opportunities for us to enhance the quality of our research and draw more impactful conclusions. As such, it is imperative that our community remain aware of these existing challenges and continue working to mitigate them through more careful planning and execution of their research projects, ultimately promoting reproducible science. We believe that research on empirical software engineering will remain a vital and enduring research field for years to come.

## 4.3 Future Work

Software debloating is an important area of research that has the potential to significantly improve the performance and reliability of software applications. Our

research has shown that there exist open challenges in improving the effectiveness of debloating. In this section, we discuss potential research directions on top of our contributions.

### 4.3.1 Neural debloating

The overall research goal of software debloating is to facilitate the adoption and integration of automatic software debloating techniques in the industry to improve software. An interesting direction for future work in this field is to explore the use of advanced Machine Learning methods to enhance the effectiveness of debloating. Promising seminal efforts in this direction have already been made employing Reinforcement Learning [99]. We consider promising the use of Deep Learning algorithms to learn patterns of code execution in order to detect and predict the emergence of code bloat. By leveraging the capabilities of these algorithms, debloating techniques can potentially achieve a higher degree of precision and promptness in identifying and removing code that is not necessary for the software's functionality.

One possible research direction towards incorporating advanced Machine Learning methods into software debloating would be to use Convolutional Neural Networks (CNN) and Neural Machine Translation (NMT) networks to facilitate feature extraction and representation of code execution patterns. These neural network architectures have proven to be effective in various software engineering tasks, including code generation from textual program descriptions [188] and automatic program repair [189]. Additionally, reinforcement learning algorithms, such as Q-learning or Deep Q-Networks (DQN), could be employed to train agents capable of making optimal decisions during the debloating process [190]. We believe that the combination of cutting-edge Machine Learning techniques holds immense potential to revolutionize software debloating, ultimately leading to leaner, more efficient, and secure software systems that can benefit the entire software engineering community.

The preservation of software functionality after the debloating process is a complex challenge that lies at the heart of software debloating [49]. This challenge is particularly daunting when attempting to identify and remove code that appears to be unused but is actually necessary for the proper functioning of the application. By leveraging advanced Machine Learning techniques, researches can potentially improve the accuracy of identifying truly necessary code, thereby preserving the intended behavior of the debloated artifacts. On the other hand, current debloating approaches rely on static analysis techniques, which face the intractable problem of accurately determining whether a given piece of code is actually

necessary for the correct execution of the software application. Moreover, some debloating techniques may inadvertently introduce new bugs or vulnerabilities, which necessitates a thorough evaluation of the debloating process. By harnessing the capabilities of Machine Learning, we hope that innovative techniques will be developed in order to accurately identify and preserve the necessary behavior of the application, ultimately addressing this critical area of research in the field of software debloating.

To evaluate such a technique, an experiment could be designed in which a dataset of software projects with known code bloat issues is collected. The new neural debloating approach would be applied to these projects, and the results be compared against traditional debloating methods (such as the code analysis techniques contributed in this thesis), as well as with the results obtained using the reinforcement learning approach by Heo *et al.* [99]. Evaluation metrics could include the amount of code bloat removed, the accuracy of the debloating decisions, and the impact on software functionality, as assessed by successfully passing the test suite. The ultimate goal is to apply and evaluate these debloating techniques in real-world production environments. This experiment would provide valuable insights into the effectiveness of advanced Machine Learning methods for software debloating and help establish the potential of these techniques in addressing uncovered future issues associated with the existence of code bloat.

### 4.3.2 Debloating across the whole software stack

Exploring debloating software across the entire software stack is a vital area for future research, as it can significantly improve the efficiency and security of software systems [36]. A promising direction involves focusing on software components within the Java Development Kit (JDK), which serves as a foundational part of numerous Java-based applications. Despite its importance, the JDK contains several features that are rarely used and therefore add unnecessary code bloat to the running applications. For instance, the CORBA (Common Object Request Broker Architecture) module, which facilitates communication between objects in a distributed system, is currently included in many JDK distributions even though most modern applications have transitioned to alternative technologies like RESTful web services or gRPC for distributed computing. In the case of DepClean, it imports the entire package `java.util.zip` from the JDK, yet it only uses classes `ZipEntry` and `ZipFile` for performing JAR file manipulations, and the other classes from this package, such as classes `Deflater` and `Inflater` for general purpose compression constitute bloat for DepClean. Although the Java community has made substantial efforts in providing tools like `jdeps` to help

identify which packages are actually used by an application, there is still a lack of fully automatic tools to effectively debloat Java software. To address this issue, future research efforts could focus on identifying and removing these unused features from the JDK, thereby reducing the overall size of the software stack and improving its performance.

Debloating an entire software stack, such as the JVM, JDK, and the OS layer running on top of modern containers, is a complex yet crucial endeavor because it involves carefully analyzing, maintaining, and testing not only the application code but also its dependencies and the underlying runtime environment. To accomplish this, a holistic approach is required, which considers debloating at every layer of the stack. One of the main challenges for future research on full-stack debloating is that dependencies and features often interact in non-trivial ways, making it difficult to determine which components can be safely removed without affecting the overall functionality. To tackle this challenge, researchers could develop sophisticated debloating techniques that combine static and dynamic analysis, along with Machine Learning, to identify and remove bloat at different levels (*e.g.*, through the analysis of system calls). For instance, a debloating approach could begin by analyzing the JDK and JVM layers, identifying rarely used or obsolete modules and components. Following this, the debloating process could be extended to the application and container layers, focusing on the dependencies and features specific to the frameworks used, *e.g.* Spring Boot or Quarkus. Throughout the process, the future debloating techniques will ensure the preservation of software functionality by carefully evaluating the potential impact of each code removal on the overall system's behavior.

The results of our studies stress the need to engineer, *i.e.*, analyze, maintain, and test dependency configuration files to avoid software bloat at a higher level of the software stack. Debloating modern frameworks that contain many bloated dependencies, such as the aforementioned Spring Boot and Quarkus, is another important area for future research. These frameworks are designed to simplify the development process by providing pre-built features and dependencies that can be easily integrated into applications. However, this convenience comes at the cost of bloated dependencies and unnecessary features that can slow down application performance and increase the risk of security vulnerabilities. To address this issue, future research could focus on developing more efficient and streamlined versions of these frameworks that remove unnecessary dependencies and features, while still maintaining the core functionality that developers appreciate. By doing so, tailored frameworks can help to reduce the overall bloat of the software stack and improve the efficiency and security of software in production environments.

## 4.4 Summary

In this section, we presented the key results for each of our technical contributions. We discussed how our debloating approaches help to cope with the increasing complexity of software systems. Additionally, we reflected on the challenges encountered while conducting empirical software engineering research, offering valuable insights on the opportunities for future work. As we continue to identify promising research directions for further studies in this field, it is essential to confront and overcome the existing challenges in order to promote the development and adoption of effective software debloating techniques, ultimately contributing to developing better software systems.

# References


[1] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The Emergence of Software Diversity in Maven Central", in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 333–343. DOI: 10.1109/MSR.2019.00059.

[2] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem", *Springer Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021. DOI: 10.1007/s10664-020-09914-8.

[3] C. Soto-Valero, T. Durieux, and B. 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 (ESEC/FSE)*, 2021, pp. 1021–1031. DOI: 10.1145/3468264.3468589.

[4] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, "Coverage-Based Debloating for Java Bytecode", *ACM Transactions on Software Engineering and Methodology*, pp. 1–34, 2022. DOI: 10.1145/3546948.

[5] C. Soto-Valero, M. Monperrus, and B. Baudry, "The Multibillion Dollar Software Supply Chain of Ethereum", *IEEE Computer*, vol. 55, no. 10, pp. 26–34, 2022. DOI: 10.1109/MC.2022.3175542.

[6] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies", *In arXiv*, pp. 1–17, 2023. DOI: 10.48550/ARXIV.2302.08370.

[7] C. Soto-Valero, J. Bourcier, and B. Baudry, "Detection and Analysis of Behavioral T-Patterns in Debugging Activities", in *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 110–113. DOI: 10.1145/3196398.3196452.

[8] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: a Temporal Graph-Based Representation of Maven Central", in *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 344–348. DOI: 10.1109/MSR.2019.00060.

[9] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "The Strengths and Behavioral Quirks of Java Bytecode Decompilers", in *Proceedings of the IEEE 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2019, pp. 92–102. DOI: 10.1109/scam.2019.00019.

[10] C. Soto-Valero and M. Pic, "Assessing the Causal Impact of the 3-Point per Victory Scoring System in the Competitive Balance of LaLiga", *International Journal of Computer Science in Sport*, vol. 18, no. 3, pp. 69–88, 2019. DOI: 10.2478/ijcss-2019-0018.

[11] N. Harrand, C. Soto-Valero, M. Monperrus, and B. Baudry, "Java Decompiler Diversity and Its Application to Meta-Decompilation", *Journal of Systems and Software*, vol. 168, p. 110 645, 2020. DOI: 10.1016/j.jss.2020.110645.

[12] G. Halvardsson, J. Peterson, C. Soto-Valero, and B. Baudry, "Interpretation of Swedish Sign Language Using Convolutional Neural Networks and Transfer Learning", *Springer SN Computer Science*, vol. 2, no. 3, p. 207, 2021. DOI: 10.1007/s42979-021-00612-w.

[13] T. Durieux, C. Soto-Valero, and B. Baudry, "DUETS: A Dataset of Reproducible Pairs of Java Library–Clients", in *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 545–549. DOI: 10.1109/MSR52588.2021.00071.

[14] N. Harrand, A. Benelallam, C. Soto-Valero, F. Bettega, O. Barais, and B. Baudry, "API Beauty Is in the Eye of the Clients: 2.2 Million Maven Dependencies Reveal the Spectrum of Client–API Usages", *Journal of Systems and Software*, vol. 184, p. 111 134, 2022. DOI: 10.1016/j.jss.2021.111134.

[15] M. Balliu *et al.*, "Challenges of Producing Software Bill Of Materials for Java", *In arXiv*, pp. 1–10, 2023. DOI: 10.48550/arXiv.2303.11102.

[16] J. Ron, Soto-Valero, L. Zhang, B. Baudry, and M. Monperrus, "Highly Available Blockchain Nodes With N-Version Design", *In arXiv*, pp. 1–12, 2023. DOI: 10.48550/arXiv.2303.14438.

[17] C. W. Krueger, "Software reuse", *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[18] V. R. Basili, L. C. Briand, and W. L. Melo, "How reuse influences productivity in object-oriented systems", *Communications of the ACM*, vol. 39, no. 10, pp. 104–116, 1996.

[19] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability", in *Proceedings. 26th International Conference on Software Engineering*, IEEE, 2004, pp. 282–291.

[20] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems", *Empirical Software Engineering*, vol. 24, pp. 381–416, 2019.

[21] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? An empirical study on the impact of security advisories on library migration", *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[22] F. Mancinelli *et al.*, "Managing the complexity of large free and open source package-based software distributions", in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, IEEE, 2006, pp. 199–208.

[23] F. Hou and S. Jansen, "A systematic literature review on trust in the software ecosystem", *Empirical Software Engineering*, vol. 28, no. 1, p. 8, 2023.

[24] C. Lima and A. Hora, "What are the characteristics of popular APIs? A large-scale study on Java, Android, and 165 libraries", *Software Quality Journal*, vol. 28, no. 2, pp. 425–458, 2020.

[25] A. S. Foundation, *Apache Maven*, Available at `https://maven.apache.org`, Jan. 2023.

[26] GitHub, *npm: A JavaScript package manager*, Available at `https://www.npmjs.com/package/npm`, Jan. 2023.

[27] GitHub, *pip: The PyPA recommended tool for installing Python packages*, Available at `https://pypi.org/project/pip`, Jan. 2023.

[28] Apache Software Foundation, *The Maven Central Repository*, Available at `https://mvnrepository.com/repos/central`, Jan. 2023.

[29] *Snyk state of open source security 2022*, `https://snyk.io/reports/open-source-security/`, Accessed: 2023-01-11.

[30] T. Gustavsson, "Managing the open source dependency", *Computer*, vol. 53, no. 2, pp. 83–87, 2020.

[31] R. Cox, "Surviving software dependencies", *Communications of the ACM*, vol. 62, no. 9, pp. 36–43, 2019.

[32] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang, "Escaping dependency hell: Finding build dependency errors with the unified dependency graph", in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 463–474.

[33] P. Salza, F. Palomba, D. Di Nucci, A. De Lucia, and F. Ferrucci, "Third-party libraries in mobile apps: When, how, and why developers update them", *Empirical Software Engineering*, vol. 25, pp. 2341–2377, 2020.

[34] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the apache community upgrades dependencies: an evolutionary study", *Empirical Software Engineering*, vol. 20, pp. 1275–1317, 2015.

[35] Y. Wu, Y. Manabe, T. Kanda, D. M. German, and K. Inoue, "Analysis of license inconsistency in large collections of open source projects", *Empirical Software Engineering*, vol. 22, pp. 1194–1222, 2017.

[36] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments", in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017, pp. 65–70.

[37] S. Butler *et al.*, "Maintaining interoperability in open source software: A case study of the Apache PDFBox project", *Journal of Systems and Software*, vol. 159, p. 110 452, 2020.

[38] G. J. Holzmann, "Code Inflation.", *IEEE Software*, vol. 32, no. 2, pp. 10–13, 2015.

[39] F. Massacci and I. Pashchenko, "Technical leverage in a software ecosystem: Development opportunities and security risks", in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 1386–1397.

[40] S. E. Ponta, W. Fischer, H. Plate, and A. Sabetta, "The Used, the Bloated, and the Vulnerable: Reducing the Attack Surface of an Industrial Application", in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2021, pp. 555–558.

[41] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications", in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 421–426.

[42] L. Gelle, H. Saidi, and A. Gehani, "Wholly!: a build system for the modern software stack", in *Formal Methods for Industrial Critical Systems: 23rd International Conference, FMICS 2018, Maynooth, Ireland, September 3-4, 2018, Proceedings 23*, Springer, 2018, pp. 242–257.

[43] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is static analysis able to identify unnecessary source code?", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–23, 2020.

[44] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, recycle to de-bloat software", in *ECOOP 2011–Object-Oriented Programming: 25th European Conference, Lancaster, Uk, July 25-29, 2011 Proceedings 25*, Springer, 2011, pp. 408–432.

[45] M. D. Brown and S. Pande, "Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating", in *CSET@ USENIX Security Symposium*, 2019.

[46] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: debloating dynamic server-side JavaScript applications", *Empirical Software Engineering*, vol. 27, no. 7, p. 161, 2022.

[47] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study", in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 507–518.

[48] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "JShrink: In-Depth Investigation into Debloating Modern Java Applications", in *Proc. ESEC/FSE*, 2020, pp. 135–146.

[49] Q. Xin, Q. Zhang, and A. Orso, "Studying and understanding the tradeoffs between generality and reduction in software debloating", in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[50] C.-C. Chuang, L. Cruz, R. van Dalen, V. Mikovski, and A. van Deursen, "Removing dependencies from large software projects: are you really sure?", in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2022, pp. 105–115.

[51] X. Tërnava, M. Acher, L. Lesoil, A. Blouin, and J.-M. Jézéquel, "Scratching the Surface of./configure: Learning the Effects of Compile-Time Options on Binary Size and Gadgets", in *Reuse and Software Quality: 20th International Conference on Software and Systems Reuse, ICSR 2022, Montpellier, France, June 15–17, 2022, Proceedings*, Springer, 2022, pp. 41–58.

[52] D. Spinellis, "Reflection as a mechanism for software integrity verification", *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 1, pp. 51–62, 2000.

[53] A. A. Ahmad *et al.*, "Trimmer: An automated system for configuration-based software debloating", *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3485–3505, 2021.

[54] A. Quach and A. Prakash, "Bloat factors and binary specialization", in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019, pp. 31–38.

[55] C. G. Kalhauge and J. Palsberg, "Binary reduction of dependency graphs", in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 556–566.

[56] N. Wirth, "A plea for lean software", *Computer*, vol. 28, no. 2, pp. 64–68, 1995.

[57] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: Cross-project code reuse in github", in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, IEEE, 2017, pp. 291–301.

[58] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects", in *Top Productivity through Software Reuse: 12th International Conference on Software Reuse, ICSR 2011, Pohang, South Korea, June 13-17, 2011. Proceedings 12*, Springer, 2011, pp. 207–222.

[59] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 4, pp. 1–44, 2013.

[60] N. Harutyunyan, "Managing your open source supply chain-why and how?", *Computer*, vol. 53, no. 6, pp. 77–81, 2020.

[61] S. Eder, H. Femmer, B. Hauptmann, and M. Junker, "Which features do my users (not) use?", in *2014 IEEE International Conference on Software Maintenance and Evolution*, IEEE, 2014, pp. 446–450.

[62] Y. Wang *et al.*, "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*, 2018, pp. 319–330.

[63] Y. Jiang, D. Wu, and P. Liu, “Jred: Program customization and bloatware mitigation based on static analysis”, in *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, IEEE, vol. 1, 2016, pp. 12–21.

[64] Y. Jiang, C. Zhang, D. Wu, and P. Liu, “Feature-based software customization: Preliminary analysis, formalization, and methods”, in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, IEEE, 2016, pp. 122–131.

[65] M. Lamothe, Y.-G. Guéhéneuc, and W. Shang, “A systematic review of API evolution literature”, *ACM Computing Surveys (CSUR)*, vol. 54, no. 8, pp. 1–36, 2021.

[66] C. D. Roover, R. Lämmel, and E. Pek, “Multi-dimensional Exploration of API Usage”, in *21st International Conference on Program Comprehension*, ser. ICPC, 2013, pp. 152–161. DOI: 10.1109/ICPC.2013.6613843.

[67] Z. Guo *et al.*, “How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–56, 2021.

[68] F. P. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering”, *Computer*, vol. 20, no. 4, pp. 1019, Apr. 1987, ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532. [Online]. Available: https://doi.org/10.1109/MC.1987.1663532.

[69] P.-H. Kamp, “The Most Expensive One-Byte Mistake”, *Commun. ACM*, vol. 54, no. 9, pp. 4244, 2011, ISSN: 0001-0782. DOI: 10.1145/1995376.1995391. [Online]. Available: https://doi.org/10.1145/1995376.1995391.

[70] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, “Piranha: Reducing feature flag debt at Uber”, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 221–230.

[71] B. A. Azad, P. Laperdrix, and N. Nikiforakis, “Less is More: Quantifying the Security Benefits of Debloating Web Applications”, in *USENIX Security Symposium*, 2019, pp. 1697–1714.

[72] E. Raymond, “The cathedral and the bazaar”, *Knowledge, Technology & Policy*, vol. 12, no. 3, pp. 23–49, 1999.

[73] T. B. Callo Arias, P. van der Spek, and P. Avgeriou, “A practice-driven systematic review of dependency analysis solutions”, *Empirical Software Engineering*, vol. 16, pp. 544–586, 2011.

[74] H. Zhong and H. Mei, “An empirical study on API usages”, *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2017.

[75] A. Gkortzis, D. Feitosa, and D. Spinellis, “Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities”, *Journal of Systems and Software*, vol. 172, p. 110 653, 2021.

[76] O. Shmueli and B. Ronen, “Excessive software development: Practices and penalties”, *International Journal of Project Management*, vol. 35, no. 1, pp. 13–27, 2017.

[77] S. Bhattacharya, K. Rajamani, K Gopinath, and M. Gupta, “The interplay of software bloat, hardware energy proportionality and system bottlenecks”, in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, 2011, pp. 1–5.

[78] Y. Tang *et al.*, “Xdebloat: Towards automated feature-oriented app debloating”, *IEEE Transactions on Software Engineering*, vol. 48, no. 11, pp. 4501–4520, 2021.

[79] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, “TRIMMER: application specialization for code debloating”, in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 329–339.

[80] H. C. Vázquez, A. Bergel, S. Vidal, J. D. Pace, and C. Marcos, “Slimming javascript applications: An approach for removing unused functions from javascript libraries”, *Information and software technology*, vol. 107, pp. 18–29, 2019.

[81] T. M. Ahmed, W. Shang, and A. E. Hassan, “An empirical study of the copy and paste behavior during development”, in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, IEEE, 2015, pp. 99–110.

[82] H.-C. Kuo, J. Chen, S. Mohan, and T. Xu, “Set the configuration for the heart of the os: On the practicality of operating system kernel debloating”, *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 1, pp. 1–27, 2020.

[83] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "Does lean imply green? a study of the power performance implications of Java runtime bloat", *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1, pp. 259–270, 2012.

[84] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, "Software bloat and wasted joules: Is modularity a hurdle to green software?", *Computer*, vol. 44, no. 09, pp. 97–101, 2011.

[85] C. Qian, H. Hu, M. Alharthi, S. P. H. Chung, T. Kim, and W. Lee, "RAZOR: A Framework for Post-deployment Software Debloating", in *USENIX Security Symposium*, 2019, pp. 1733–1750.

[86] Y. Chen, T. Lan, and G. Venkataramani, "Damgate: Dynamic adaptive multi-feature gating in program binaries", in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, 2017, pp. 23–29.

[87] G. A. Campbell, "Cognitive complexity: An overview and evaluation", in *Proceedings of the 2018 international conference on technical debt*, 2018, pp. 57–58.

[88] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading", in *27th {USENIX} Security Symposium*, 2018, pp. 869–886.

[89] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for Java", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 24, no. 6, pp. 625–666, 2002.

[90] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 476–486.

[91] R. Williams, T. Ren, L. De Carli, L. Lu, and G. Smith, "Guided feature identification and removal for resource-constrained firmware", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 2, pp. 1–25, 2021.

[92] M. D. Brown and S. Pande, "Carve: Practical security-focused software debloating using simple feature set mappings", in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, 2019, pp. 1–7.

[93] J. Landsborough, S. Harding, and S. Fugate, "Removing the kitchen sink from software", in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, 2015, pp. 833–838.

[94] A. Ruprecht, B. Heinloth, and D. Lohmann, "Automatic feature selection in large-scale system-software product lines", *ACM SIGPLAN Notices*, vol. 50, no. 3, pp. 39–48, 2014.

[95] A. Ziegler, J. Geus, B. Heinloth, T. Hönig, and D. Lohmann, "Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–23, 2019.

[96] G. Malecha, A. Gehani, and N. Shankar, "Automated software winnowing", in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, pp. 1504–1511.

[97] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, "BinTrimmer: Towards static binary debloating through abstract interpretation", in *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16*, Springer, 2019, pp. 482–501.

[98] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su, "Perses: Syntax-guided program reduction", in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 361–371.

[99] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning", in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 380–394.

[100] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating", in *Proceedings of the 12th European Workshop on Systems Security*, 2019, pp. 1–6.

[101] Y. Chen, S. Sun, T. Lan, and G. Venkataramani, "Toss: Tailoring online server systems through binary feature customization", in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation*, 2018, pp. 1–7.

[102] P. Biswas, N. Burow, and M. Payer, "Code specialization through dynamic feature observation", in *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, 2021, pp. 257–268.

[103] J. Wu *et al.*, "LIGHTBLUE: Automatic profile-aware debloating of bluetooth stacks", in *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.

[104] I. Agadakos *et al.*, "Large-scale debloating of binary shared libraries", *Digital Threats: Research and Practice*, vol. 1, no. 4, pp. 1–28, 2020.

[105] H. Zhang, M. Ren, Y. Lei, and J. Ming, "One size does not fit all: security hardening of mips embedded systems via static binary debloating for shared libraries", in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 255–270.

[106] P. Pashakhanloo *et al.*, "Pacjam: Securing dependencies continuously via package-oriented debloating", in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 903–916.

[107] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Program debloating via stochastic optimization", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, 2020, pp. 65–68.

[108] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries", in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 70–83.

[109] C. Porter, G. Mururu, P. Barua, and S. Pande, "Blankit library debloating: Getting what you want instead of cutting what you dont", in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 164–180.

[110] M. Ghaffarinia and K. W. Hamlen, "Binary control-flow trimming", in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1009–1022.

[111] J. Christensen, I. M. Anghel, R. Taglang, M. Chiroiu, and R. Sion, "DECAF: Automatic, adaptive de-bloating and hardening of COTS firmware", in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 1713–1730.

[112] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat", in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 268–278.

[113] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications", in *Proceedings of the 2013 international symposium on memory management*, 2013, pp. 119–130.

[114] G. Xu and A. Rountev, “Detecting inefficiently-used containers to avoid bloat”, in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 160–173.

[115] S. Bhattacharya, K. Gopinath, and M. G. Nanda, “Combining concern input with program analysis for bloat detection”, *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 745–764, 2013.

[116] Q. Xin, F. Behrang, M. Fazzini, and A. Orso, “Identifying features of android apps from execution traces”, in *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, IEEE, 2019, pp. 35–39.

[117] G. Wagner, A. Gal, and M. Franz, “Slimming a Java virtual machine by way of cold code removal and optimistic partial program loading”, *Science of Computer Programming*, vol. 76, no. 11, pp. 1037–1053, 2011.

[118] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, “RedDroid: Android application redundancy customization based on static analysis”, in *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, IEEE, 2018, pp. 189–199.

[119] Z. El-Rewini and Y. Aafer, “Dissecting Residual APIs in Custom Android ROMs”, in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1598–1611.

[120] O. Starov, P. Laperdrix, A. Kapravelos, and N. Nikiforakis, “Unnecessarily Identifiable: Quantifying the fingerprintability of browser extensions due to bloat”, in *The World Wide Web Conference*, 2019, pp. 3244–3250.

[121] F. d. A. Farzat, M. d. O. Barros, and G. H. Travassos, “Evolving JavaScript code to reduce load time”, *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1544–1558, 2019.

[122] R. Morales, R. Saborido, and Y.-G. Guéhéneuc, “Momit: Porting a javascript interpreter on a quarter coin”, *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2771–2785, 2020.

[123] C. Qian, H. Koo, C. Oh, T. Kim, and W. Lee, “Slimium: debloating the chromium browser with feature subsetting”, in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 461–476.

[124] I. Koishybayev and A. Kapravelos, “Mininode: Reducing the Attack Surface of Node. js Applications”, in *RAID*, 2020, pp. 121–134.

[125] R. Ye, L. Liu, S. Hu, F. Zhu, J. Yang, and F. Wang, "JSLIM: Reducing the known vulnerabilities of Javascript application by debloating", in *Emerging Information Security and Applications: Second International Symposium, EISA 2021, Copenhagen, Denmark, November 12-13, 2021, Revised Selected Papers*, Springer, 2022, pp. 128–143.

[126] C. Oh, S. Lee, C. Qian, H. Koo, and W. Lee, "Deview: Confining progressive web applications by debloating web apis", in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 881–895.

[127] M. Hague, A. W. Lin, and C.-H. L. Ong, "Detecting redundant CSS rules in HTML5 applications: a tree rewriting approach", in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015, pp. 1–19.

[128] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at google)", in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 724–734.

[129] D. Qiu, B. Li, and H. Leung, "Understanding the API usage in Java", *Information and software technology*, vol. 73, pp. 81–100, 2016.

[130] H. V. Pham, P. M. Vu, T. T. Nguyen, *et al.*, "Learning API Usages From Bytecode: A Statistical Approach", in *Proceedings of the 38th International Conference on Software Engineering*, ACM, 2016, pp. 416–427.

[131] J. Hejderup, "In Dependencies We Trust: How Vulnerable Are Dependencies in Software Modules?", Ph.D. dissertation, Delft University of Technology, 2015.

[132] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: From package-based to call-based dependency networks", *Empirical Software Engineering*, vol. 27, no. 5, p. 102, 2022.

[133] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage Analysis of Open-source Java Projects", in *Proceedings of the 2011 ACM Symposium on Applied Computing*, ser. SAC '11, TaiChung, Taiwan: ACM, 2011, pp. 1317–1324. DOI: 10.1145/1982185.1982471.

[134] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek, "An Exploratory Study on Reuse at Google", in *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, ser. SERIP, Hyderabad, India: ACM, 2014, pp. 14–23, ISBN: 978-1-4503-2859-3. DOI: 10.1145/2593850.2593854.

[135] B. A. Myers and J. Stylos, “Improving API Usability”, *Communications of the ACM*, vol. 59, no. 6, pp. 62–69, 2016.

[136] W. C. Lim, “Effects of Reuse on Quality, Productivity, and Economics”, *IEEE Software*, vol. 11, no. 5, pp. 23–30, 1994, ISSN: 0740-7459. DOI: 10.1109/52.311048.

[137] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[138] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric, “Build system with lazy retrieval for Java projects”, in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 643–654.

[139] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, “Removing False Code Dependencies to Speedup Software Build Processes”, in *Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON, Toronto, Ontario, Canada: IBM Press, 2003, pp. 343–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=961322.961375.

[140] D. A. Wheeler, “Preventing heartbleed”, *Computer*, vol. 47, no. 08, pp. 80–83, 2014.

[141] H. Elahi, G. Wang, and X. Li, “Smartphone bloatware: an overlooked privacy problem”, in *Security, Privacy, and Anonymity in Computation, Communication, and Storage: 10th International Conference, SpaCCS 2017, Guangzhou, China, December 12-15, 2017, Proceedings 10*, Springer, 2017, pp. 169–185.

[142] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, “Vuln4real: A methodology for counting actually vulnerable dependencies”, *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2020.

[143] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, “An empirical study on the maintenance of source code clones”, *Empirical Software Engineering*, vol. 15, pp. 1–34, 2010.

[144] D. Caivano, P. Cassieri, S. Romano, and G. Scanniello, “An Exploratory Study on Dead Methods in Open-source Java Desktop Applications”, in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–11.

[145] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design”, *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[146] R. Haas, R. Niedermayr, T. Röhm, and S. Apel, "Recommending Unnecessary Source Code Based on Static Analysis", *ICSE'19 Companion*, vol. 178, 2019.

[147] K. Ali, X. Lai, Z. Luo, O. Lhoták, J. Dolby, and F. Tip, "A study of call graph construction for JVM-hosted languages", *IEEE transactions on software engineering*, vol. 47, no. 12, pp. 2644–2666, 2019.

[148] G. Antal, P. Hegeds, Z. Herczeg, G. Lóki, and R. Ferenc, "Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools", *IEEE Access*, vol. 11, pp. 25 266–25 284, 2023. DOI: 10.1109/ACCESS.2023.3255984.

[149] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir, "On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation", in *Programming Languages and Systems: 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2–6, 2018, Proceedings 16*, Springer, 2018, pp. 69–88.

[150] B. Livshits *et al.*, "In defense of soundiness: A manifesto", *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[151] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019.

[152] M. Hilton, J. Bell, and D. Marinov, "A large-scale study of test coverage evolution", in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 53–63.

[153] U. P. Schultz, J. L. Lawall, and C. Consel, "Automatic program specialization for java", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 4, pp. 452–499, 2003.

[154] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The Dynamics of Software Composition Analysis", *Poster at the 34th ACM/IEEE International Conference on Automated Software Engineering*, ASE 2019, 2019.

[155] S. Liang and G. Bracha, "Dynamic Class Loading in the Java Virtual Machine", in *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 98, Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, pp. 3644, ISBN: 1581130058. DOI: 10.1145/286936.286945. [Online]. Available: https://doi-org.focus.lib.kth.se/10.1145/286936.286945.

[156] L. Chen, F. Hassan, X. Wang, and L. Zhang, “Taming behavioral backward incompatibilities via cross-project testing and analysis”, in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.

[157] M. Alhanahnah, R. Jain, V. Rastogi, S. Jha, and T. Reps, “Lightweight, multi-stage, compiler-assisted application specialization”, in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, IEEE, 2022, pp. 251–269.

[158] B. Baudry, S. Allier, and M. Monperrus, “Tailored source code transformations to synthesize computationally diverse program variants”, in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 149–159.

[159] Apache Maven, *Introduction to the Dependency Mechanism*, Available at `https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html`, Jan. 2023.

[160] A. Gkortzis, D. Feitosa, and D. Spinellis, “A double-edged sword? Software reuse and potential security vulnerabilities”, in *Reuse in the Big Data Era: 18th International Conference on Software and Systems Reuse, ICSR 2019, Cincinnati, OH, USA, June 26–28, 2019, Proceedings 18*, Springer, 2019, pp. 187–203.

[161] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, “On the recall of static call graph construction in practice”, in *Proc. of ICSE*, 2020, pp. 1049–1060.

[162] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, “Code coverage differences of java bytecode and source code instrumentation tools”, *Software Quality Journal*, vol. 27, pp. 79–123, 2019.

[163] Apache Maven, *Maven Dependency Analyzer Plugin*, Available at `http://maven.apache.org/shared/maven-dependency-analyzer`, Jan. 2023.

[164] OW2 Consortium, *ASM Java bytecode manipulation and analysis framework*, Available at `https://asm.ow2.io`, Jan. 2023.

[165] C. C. Chuang, “How to remove dependencies from large software projects with confidence”, 2022.

[166] H. Borges and M. T. Valente, “Whats in a github star? understanding repository starring practices in a social coding platform”, *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[167] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, “Breaking bad? Semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study”, *Empirical Software Engineering*, vol. 27, no. 3, p. 61, 2022.

[168] J. Düsing and B. Hermann, “Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories”, *Digital Threats: Research and Practice*, vol. 3, no. 4, pp. 1–25, 2022.

[169] D. Jaime, J. El Haddad, and P. Poizat, “A preliminary study of rhythm and speed in the maven ecosystem”, in *21st Belgium-Netherlands Software Evolution Workshop*, 2022.

[170] S. Venkatanarayanan, J. Dietrich, C. Anslow, and P. Lam, “VizAPI: Visualizing Interactions between Java Libraries and Clients”, in *2022 Working Conference on Software Visualization (VISSOFT)*, IEEE, 2022, pp. 172–176.

[171] L. Martins, H. Costa, and I. Machado, “On the diffusion of test smells and their relationship with test code quality of Java projects”, *Journal of Software: Evolution and Process*, pp. 1–20, 2023.

[172] T. Xie, J. Pei, and A. E. Hassan, “Mining software engineering data”, in *29th International Conference on Software Engineering (ICSE’07 Companion)*, IEEE, 2007, pp. 172–173.

[173] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, “Crossrec: Supporting software developers by recommending third-party libraries”, *Journal of Systems and Software*, vol. 161, p. 110 460, 2020.

[174] N. E. Gold and J. Krinke, “Ethics in the mining of software repositories”, *Empirical Software Engineering*, vol. 27, no. 1, p. 17, 2022.

[175] A. Decan, T. Mens, and E. Constantinou, “On the evolution of technical lag in the npm package dependency network”, in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, IEEE, 2018, pp. 404–414.

[176] L. Erlenhov, F. G. d. O. Neto, and P. Leitner, “An Empirical Study of Bots in Software Development: Characteristics and Challenges from a Practitioners Perspective”, in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 445455, ISBN: 9781450370431. DOI: 10.1145/3368089.3409680.

[177] S. Mirhosseini and C. Parnin, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?”, in *2017 32nd IEEE/ACM international conference on automated software engineering (ASE)*, IEEE, 2017, pp. 84–94.

[178] H. F. Li and W. K. Cheung, “An empirical study of software metrics”, *IEEE Transactions on Software Engineering*, no. 6, pp. 697–708, 1987.

[179] D. Coleman, D. Ash, B. Lowther, and P. Oman, “Using metrics to evaluate software system maintainability”, *Computer*, vol. 27, no. 8, pp. 44–49, 1994.

[180] N. E. Fenton and M. Neil, “Software metrics: roadmap”, in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 357–370.

[181] Y.-G. Guéhéneuc and F. Khomh, “Empirical software engineering”, *Handbook of Software Engineering*, pp. 285–320, 2019.

[182] C. Boettiger, “An introduction to Docker for reproducible research”, *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[183] F. Hassan, S. Mostafa, E. S. Lam, and X. Wang, “Automatic building of java projects in software repositories: A study on feasibility and challenges”, in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2017, pp. 38–47.

[184] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, “An empirical analysis of flaky tests”, in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 643–653.

[185] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, “Finding and Reproducing Heisenbugs in Concurrent Programs.”, in *OSDI*, vol. 8, 2008.

[186] C. Artho, K. Havelund, and A. Biere, “High-level data races”, *Software Testing, Verification and Reliability*, vol. 13, no. 4, pp. 207–227, 2003.

[187] V. Basili and L. Briand, “Reflections on the Empirical Software Engineering journal”, *Empirical Software Engineering*, vol. 27, pp. 1–4, 2022.

[188] C. Lyu, R. Wang, H. Zhang, H. Zhang, and S. Hu, “Embedding API dependency graph for neural code generation”, *Empirical Software Engineering*, vol. 26, pp. 1–51, 2021.

[189] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair”, *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[190] J. A. Prado Lima, W. D. Mendonça, S. R. Vergilio, and W. K. Assunção, "Cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software", *Empirical Software Engineering*, vol. 27, no. 6, p. 133, 2022.

# Part II

# Appended Research Papers

# Paper I

2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

# The Emergence of Software Diversity in Maven Central

César Soto-Valero[†], Amine Benelallam[*], Nicolas Harrand[†], Olivier Barais[*], and Benoit Baudry[†]
[†]KTH Royal Institute of Technology, Stockholm, Sweden
Email: {cesarsv, harrand, baudry}@kth.se
[*]Univ Rennes, Inria, CNRS, IRISA, Rennes, France
Email: amine.benelallam@inria.fr, barais@irisa.fr

***Abstract*—Maven artifacts are immutable: an artifact that is uploaded on Maven Central cannot be removed nor modified. The only way for developers to upgrade their library is to release a new version. Consequently, Maven Central accumulates all the versions of all the libraries that are published there, and applications that declare a dependency towards a library can pick any version. In this work, we hypothesize that the immutability of Maven artifacts and the ability to choose any version naturally support the emergence of software diversity within Maven Central. We analyze 1,487,956 artifacts that represent all the versions of 73,653 libraries. We observe that more than 30% of libraries have multiple versions that are actively used by latest artifacts. In the case of popular libraries, more than 50% of their versions are used. We also observe that more than 17% of libraries have several versions that are significantly more used than the other versions. Our results indicate that the immutability of artifacts in Maven Central does support a sustained level of diversity among versions of libraries in the repository.**

***Index Terms*—Maven Central, Software Diversity, Library Versions, Evolution, Open-Source Software**

## I. Introduction

Maven Central is the most popular repository to distribute and reuse JVM-based artifacts (i.e., reusable software packages implemented in Java, Clojure, Scala or other languages that can compile to Java bytecode). By September 6, 2018, Maven Central contains over $2.8M$ artifacts and serves over $100M$ downloads every week [1]. The Maven dependency management system, which is able to resolve transitive dependencies automatically, has been key to this success: it relieves developers from the complexity of manual management of their dependencies. Uploading artifacts into Maven Central is the most effective way for open source projects to remain permanently accessible to their users. In this way, every build tool able to download Java libraries can fetch from a world of libraries and dependencies in a single and authoritative place.

In this work, we analyze software artifacts from the perspective of one essential characteristic enforced by Maven Central: immutability[1]. All artifacts (code packages, documentation, dependency declarations, etc.) that are uploaded on Maven Central are immutable: they cannot be rewritten nor deleted. This is a critical design choice that has a significant influence on the way the Maven Central repository is utilized. We hypothesize that this design decision is a great opportunity to prevent dependency monoculture [2] and increase the diversity [3] among software dependencies.

Previous works have analyzed Maven artifacts from the perspective of the risks induced by immutability. First, the redundancy in multiple versions can introduce conflicts among dependencies, e.g., trying to load the same class several times. This risk has been extensively analyzed by Wang and colleagues [4]. Second, the projects that depend on a library need to explicitly update their dependency descriptions in order to benefit from the update. This represents a risk since these projects can eventually rely on outdated dependencies [5] that can contain security issues [6] or API breaking changes [7].

We take a fresh look at the presence of multiple versions of the same library in Maven Central, and consider it as an opportunity. We analyze how the ability to choose any library version for software reuse supports the emergence of software diversity in the repository and how this diversity of versions fuels the success of popular libraries. We consider this emergent diversity of reused versions as an opportunity since it participates in mitigating the risks of software monoculture [8]. Overcoming this type of monoculture is essential to build resilient and robust software systems [3], [9], [10].

To conduct this empirical study, we rely on an existing dataset, the Maven Dependency Graph [1], which captures a snapshot of Maven Central as of September 6, 2018. This dataset comes in the form of a temporal graph with metadata of $2.4M$ artifacts belonging to $223K$ libraries, with more than $9M$ direct dependency relationships between them. In order to enable reasoning not only at the artifact level but also at the library level, we extend this dataset with another abstraction layer capturing dependencies at the library level.

We measure activity, popularity and timeliness of a subset of $73,653$ libraries with multiple versions, which represents $61.81\%$ of the total number of artifacts in Maven Central. We empirically investigate whether the diversity of library versions is a valuable design choice. Our contributions are as follows:

- a quantitative analysis of the diversity of usage and popularity of library versions;
- evidence of the presence of large quantities of artifacts that participate in the emergence of diversity;
- open science with replication code and scripts available online.

[1]Sonatype community support: https://issues.sonatype.org/browse/OSSRH-39131

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00059

## II. Background and Definitions

In this section, we describe the dataset of Maven artifacts that constitutes the raw material for our work, as well as its extended library-level abstraction.

### A. The Maven Dependency Graph

To conduct this empirical study, we rely on the Maven Dependency Graph (MDG), a dataset that captures all of the artifacts deployed on the Maven Central repository as of September 6, 2018 [1]. The MDG includes $2,407,335$ artifacts. Each artifact is uniquely identified with a triplet (*'groupId:artifactId:version'*). The *groupId* identifier is a way of grouping different Maven artifacts, for instance by library vendor. The *artifactId* identifier refers to the library name. Finally, the *version* identifies each library release uniquely. For example, the triplet *'org.neo4j:neo4j-io:3.4.7'* identifies the version 3.4.7 of an input/output abstraction layer for the Neo4j graph database. The MDG also includes $9,715,669$ dependency relationships as declared in the Project Object Model (`pom.xml`) file of each artifact.

In this work, we focus on *libraries*, i.e., the sets of artifacts that share the same tuple '*groupId:artifactId*' but have different versions. The MDG includes $223,478$ of such libraries, but the concept of library is not rigorously captured in the graph. Consequently, we extend the artifact nodes of the MDG with labels referring to their corresponding library. We call `LIBS` the set of all libraries in Maven Central. We introduce an ordering function denoted $<_v$ that leverages the standard version numbering policy described by the Apache Software Foundation[2] in order to compare the different versions of artifacts belonging to the same library. For instance, 1.2.0 $<_v$ 2.0.0. We also define a temporal ordering function denoted by $<_t$ to compare the release dates of different artifacts. For example, '12-09-2011' $<_t$ '30-03-2015'. In the remainder of the paper, we refer to artifacts as *library versions* or simply *versions*. We define the MDG as follows:

**Definition 1.** ***Maven Dependency Graph.*** *The MDG is a vertex-labelled graph, where vertices represent Maven library versions, and edges describe dependency relationships or precedence relationships. We use a labeling function over vertices to group versions by library. We define the MDG as $\mathcal{G} = (\mathcal{V}, \mathcal{D}, \mathcal{N}, \mathcal{L}, \mathcal{R})$, where,*

- *the set of vertices $\mathcal{V}$ represents the library versions present in Maven Central*
- *the set of directed edges $\mathcal{D}$ represents dependency relationships between library versions*
- *the set of directed edges $\mathcal{N}$ represents versions precedence relationships, where the version of the source node is strictly lower than the version of the target node w.r.t. $<_v$*
- *the surjective labelling function $\mathcal{L}$ returns the corresponding library of a given library version $v \in \mathcal{V}$, defined as $\mathcal{L} : \mathcal{V} \rightarrow$ `LIBS`*
- *the temporal function $\mathcal{R}$ refers to the date at which a library version $v \in \mathcal{V}$ was deployed, defined as $\mathcal{R} : \mathcal{V} \rightarrow T$, where $T$ is a $<_t$-ordered discrete time domain*

[2]https://cwiki.apache.org/confluence/display/MAVEN/Version+number+policy

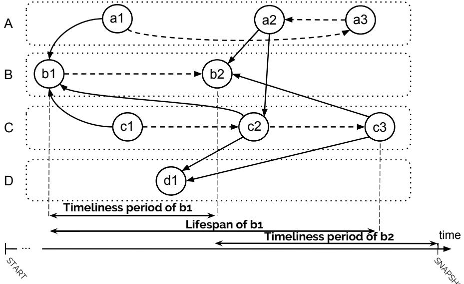



Fig. 1. Example of relationships between library versions in the Maven Dependency Graph.

In the MDG, $T$ is bounded to ['15-05-2002', '06-09-2018'], where the lower bound refers to the date when the first library was deployed on Maven Central. In the rest of the paper, we refer to the lower and upper bounds respectively as `START` and `SNAPSHOT`, and we use days as the time granularity.

Figure 1 illustrates the different nodes and relationships within a simplified graph $\mathcal{G}$ composed of four libraries (A,B,C,D) and nine library versions $\{a_1, a_2, a_3, b_1, b_2, c_1, c_2, c_3, d_1\}$. The regular edges represent dependency relationships. For example, the first version of A ($a_1$) depends on the first version of B ($b_1$), and the second version of A ($a_2$) depends on the second version of B ($b_2$) and C ($c_2$). The dashed edges represent precedence relationships, and all vertices that are related through such edges constitute the different versions of a library. In Figure 1, we place nodes in a temporal order, from left to right, corresponding to the deployment date, thus the node $b_1$ is the firstly deployed, while the node $c_3$ is the most recently deployed.

The temporal order of releases does not imply a similar versioning order for a given library. In some cases, library instances with lower version number may be released after library versions with a greater version number, e.g., in case of a library version downgrade or maintenance of several major library versions. In Figure 1, we can see that $a_2 <_t a_3$ and $a_3 <_v a_2$. Note, this is a common practice adopted by very popular libraries such as Apache CXF[3], and Mule[4] [11].

**Definition 2.** ***Additional notations.*** *For further references in the MDG, we introduce the following notations:*

- *$next(v)$: the next release of a given library version $v$ w.r.t. the ordering function $<_v$*
- *$next_{all}(v)$: transitive closure on the next releases of a library version $v$*

[3]https://cxf.apache.org

[4]https://www.mulesoft.com

- *latest: the library version* $v$ *such that* $\nexists\, next(v)$
- *LATESTS: the set of all latest library versions in a dependency graph* $\mathcal{G}$
- $deps(v)$*:* $\mathcal{V} \rightarrow \mathcal{V}^n$*, with* $n \in \mathbb{N}$*: the set of direct dependencies of a given library version* $v \in \mathcal{V}$
- $deps_{tree}(v)$*: the whole dependency tree of* $v$
- $users(v)$*:* $\mathcal{V} \rightarrow \mathcal{V}^n$*, with* $n \in \mathbb{N}$*: the set of library versions declaring a dependency towards* $v$
- $users_{all}(v)$*: all the transitive users of* $v$

For example, in Figure 1, $deps(a_2) = \{b_2, c_2\}$, $deps_{tree}(a_2) = \{b_2, c_2, d_1\}$, $users(d_1) = \{c_2, c_3\}$ and $users_{all}(d_1) = \{c_2, c_3, a_2\}$.

### B. The Maven Library's Dependency Graph

In order to be able to reason about not only versions but also libraries, we elevate the abstraction of the MDG to the library level. Figure 2 shows the elevated graph corresponding to the dependency graph $\mathcal{G}$ in Figure 1. We construct a weighted graph, $\mathcal{G}_{\mathcal{L}}$, where nodes correspond to libraries (LIBS) in $\mathcal{G}$. We create an outgoing edge between two libraries $l_1$ and $l_2$ if there is at least a version of $l_1$ that uses a library version of $l_2$. We denote by $D(l)$ the set of direct library dependencies of a given library $l$. For example, $D(A) = \{B, C\}$. Finally, the weight of the outgoing edges from $l_1$ to $l_2$ corresponds to the number of versions of $l_1$ that use a version of $l_2$. We define the Maven Library's Dependency Graph (MDG$_L$) as follows:

**Definition 3.** ***Maven Library's Dependency Graph.*** *The MDG$_L$ is a edge-weighted graph, where vertices represent Maven libraries, and edges' weight describes the number of dependency relationships between their versions. We define the MDG$_L$ as* $\mathcal{G}_{\mathcal{L}}$ *= (LIBS,* $\mathcal{E}$*,* $\mathcal{W}$*), where,*

- *the set of vertices LIBS represents the libraries present in Maven Central*
- *the set of edges* $\mathcal{E}$ *represents the dependency relationships between libraries*
- *the weighing function* $\mathcal{W}$ *represents the weight of a given edge, defined as* $\mathcal{W} : \mathcal{E} \rightarrow \mathbb{N}$

*For further references in the MDG$_L$, we introduce the following notations:*

- *the set of direct library dependencies* $D$ *of a given library, defined as* $D : \texttt{LIBS} \rightarrow \texttt{LIBS}^n$
- *the weighing function* $\overleftarrow{W}$ *returns the sum of the weights of incoming edges, defined as* $\overleftarrow{W} : \texttt{LIBS} \rightarrow \mathbb{N}$
- *the weighing function* $\overrightarrow{W}$ *returns the sum of the weights of outgoing edges, defined as* $\overrightarrow{W} : \texttt{LIBS} \rightarrow \mathbb{N}$

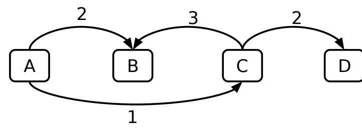



Fig. 2. The elevated Maven Library's Dependency Graph from Figure 1.

## III. Study Design

This work is articulated around five research questions. In this section, we introduce these questions as well as the metrics that we collect to answer them. We also describe the representative subset of artifacts that we study throughout the paper.

### A. Research Questions

***RQ1****: To what extent are the different library versions actively used?*

Because Maven artifacts are immutable, all the versions of a given library that have been released in Maven Central are always present in the repository. Meanwhile, previous studies have shown that users of a given version do not systematically update their dependency when a new version is released [5], [12], [13]. Consequently, we hypothesize that, at some point in time, multiple versions of a library are actively used. In this research question, we investigate how many versions are currently used, how many have been used but are not anymore and how many versions have never been used.

***RQ2****: How are the actively used versions distributed along the history of a library?*

The full history of versions of a library released on Maven Central is always available. Consequently, users can decide to depend on any of the versions. In this research question, we analyze where, in the history of versions, are located the versions that are actively used.

***RQ3****: Among the actively used versions of a library, is there one or several versions that are significantly more popular than the others?*

Library users are free to decide which version to depend on and for how much time. In the long term, these users' decisions determine what are the most popular libraries and versions in the entire software ecosystem [5], [14]. This research question investigates to what extent these decisions lead to the emergence of one or more versions that receive a greater number of usages compared to the other versions.

***RQ4****: Does the number of actively used versions relate to the popularity of a library?*

We observe that for most libraries, more than one version is actively used at a given point in time. The library developers have no control over this since they cannot remove versions from Maven Central, nor force their users to update their dependencies. Meanwhile, it might be good for a library to maintain several versions that fit different usages. In this question, we investigate how the existence of multiple active versions relates to the overall popularity of a library.

***RQ5****: How timely are the different library versions in Maven Central?*

With each new release, project maintainers make an effort to improve the quality of their libraries (e.g., by fixing bugs, adding new functionalities or increasing performance). These changes are expected to be directly reflected in the number of

users that update their dependencies to the new available release, and also in the number of new usages of the library [15]. This research question aims to get insights into how timely is the release of new versions. In particular, we investigate how much attraction gets a library version while it was the latest, compared to the older versions during the same period of time.

### *B. Metrics*

To characterize the *activity status* of libraries and versions in terms of their usages by other latest library versions, we introduce the notions of *active*, *passive*, and *dormant* libraries and versions. Moreover, we introduce the *lifespan* of library versions to get insights on the duration of their activity period. These notions and measures are intended to answer RQ1 and RQ2.

**Metric 1.** ***Activity status.*** *A* ***passive library version*** *$v$ is a version that has been used in the past, but is no longer used, even transitively, by any latest library version ($v \in$ $\mathtt{LATESTS}$). Formally, this metric is described as a boolean function $isPsv : \mathcal{V} \to \{true, false\}$, where,*

$$isPsv(v) = \begin{cases} false & v \in \bigcup\limits_{i \in \mathtt{LATESTS}} \{deps_{tree}(i)\} \\ true, & \text{otherwise} \end{cases}$$

*An* ***active library version*** *$v$ is a version where $isPsv(v) = false$. A* ***dormant library version*** *is an extreme case of a passive library version that occurs when the version has never been used by existing libraries (i.e., $users(v) = \emptyset$) in Maven Central.*

*At the library level, an* ***active library*** *is a library that has at least one active version, whereas a* ***passive library*** *is a library that has all its versions passive. A* ***dormant library*** *is an extreme case of passive library that occurs when all its versions are dormant.*

**Metric 2.** ***Lifespan.*** *The lifespan of a library version $v$ is the time range during which it was/is being used. We define this period as the time range between the release date of $v$ and the timestamp at which it becomes passive. In case $v$ is active, this period starts at the release date of the artifact until the day the $\mathtt{SNAPSHOT}$ was captured. Dormant library versions do not have a lifespan at all. We denote this metric by $ls(v)$ =$[startLs_v, endLs_v]$. Then, the interval's upper bound can be formally described as follows:*

$$endAct_v = \begin{cases} \mathtt{SNAPSHOT}, & \neg isPsv(v) \\ last, & isPsv(v) \end{cases}$$

*where,* $last = \max \bigcup\limits_{i \in users_{all}(v)} \{\mathcal{R}(next(i))\}$.

To study the *popularity* of library versions in Maven Central, and hence answer RQ3 and RQ4, we introduce a metric of popularity which measures the transitive influence and connectivity of a library version in the MDG. We rely on the standard PageRank algorithm [16], which accounts for the number of transitive usages. Intuitively, library versions with a higher PageRank are more likely to have a larger number of transitive usages. On the other hand, to measure the popularity of libraries, we use the Weighted PageRank algorithm [17] on the MDG$_L$.

**Metric 3.** ***Popularity.*** *The* ***popularity of a library version*** *$v \in \mathcal{V}$ is as follows:*

$$pop_{\mathcal{V}}(v) = (1-d) + d \sum_{i \in users(v)} pop_{\mathcal{V}}(i),$$

*where $d$ is a damping factor to reflect user behavior, which is usually set to 0.85 [18].*

*The* ***popularity of a library*** *$l \in \mathtt{LIBS}$ is as follows:*

$$pop_{\mathcal{L}}(l) = (1-d) + d \sum_{i \in U(l)} pop_v(i) \overleftarrow{c}_{(l,i)} \overrightarrow{c}_{(l,i)},$$

*where $\overleftarrow{c}$ and $\overrightarrow{c}$ are respectively:*

$$\overleftarrow{c}_{(l,i)} = \frac{\overleftarrow{W}(i)}{\sum\limits_{p \in D(l)} \overleftarrow{W}(p)}, \quad \overrightarrow{c}_{(l,i)} = \frac{\overrightarrow{W}(i)}{\sum\limits_{p \in D(l)} \overrightarrow{W}(p)}.$$

Finally, to answer RQ5, we introduce the notion of *timeliness* of library versions. This metric looks at the number of usages of every single version when it was latest and assesses if it was successful in attracting more users compared to its older versions. To this end, we compare the usages of a given version $v$ during its lifespan to the usages that the whole library has received during the period when $v$ was latest. We call this period the *timeliness period*.

**Metric 4.** ***Timeliness.*** *The* ***timeliness period****, $tp(v)$, of a library version $v$, is the time range between the release date of $v$ and the most recently released version of its library ordered by $<_t$, which is not necessarily $next(v)$. We denote this version as $mr$:*

*$tp(v) = [\mathcal{R}(v), \mathcal{R}(mr)]$,*
*where,* $mr = \min\limits_{i \in next_{all}(v)} \{\mathcal{R}(i) | \mathcal{R}(i) >_t \mathcal{R}(v)\}$.

*The* ***timeliness of a library version*** *$v$ is a function, $tim(v) : \mathcal{V} \to \mathbb{Q}^+$, where,*

$$tim(v) = \frac{|users(v)|}{|\bigcup\limits_{i \in \mathcal{V}} \{i | \mathcal{R}(i) \in tp(v) \wedge \mathcal{L}(v) \in \bigcup\limits_{j \in deps(i)} \{\mathcal{L}(j)\}\}|}.$$

*In case the library corresponding to $v$ was not used during the timeliness period of $v$ (the denominator is 0), then we consider $tim(v) = 0$. This also applies when $v$ is dormant. All first releases of libraries have $tim(v) = 1$ since they have no earlier releases.*

*Based on the timeliness metric, three situations can occur:*

- *$v$ is* ***timely*** *if $tim(v) = 1$: $v$ was a success during its timeliness period and users relied on it*
- *$v$ is* ***over-timely*** *if $tim(v) > 1$: $v$ has attracted users beyond its timeliness period*
- *$v$ is* ***under-timely*** *if $tim(v) < 1$: users relied on older versions during its timeliness period*

TABLE I
CATEGORIES OF LIBRARIES IN MAVEN CENTRAL ACCORDING TO THEIR RELEASING PROFILES

| Category | Criteria | #Libraries (%) | #Versions (%) |
|---|---|---|---|
| (i) | #versions = 1 | 65,557 (29.33%) | 65,557 (2.72%) |
| (ii) | One shot* | 32,825 (14.69%) | 459,445 (19.08%) |
| (iii) | #versions > 1 | 125,096 (55.98%) | 1,882,333 (78.2%) |

(*) Libraries with more than one version and that have been released in the same day.

### C. Study Subjects

During our initial exploration of the MDG, we distinguished three different categories of libraries in Maven Central: (i) libraries that have only one version (∼30%), (ii) libraries with multiple versions all released on the same day (∼15%), and (iii) libraries with multiple versions released within different time intervals (∼55%). Table I gives detailed numbers about these categories. In particular, after manual inspection we notice that a large number of libraries belonging to categories (i) and (ii) are shipped with their classpath. We suspect these projects to be using Maven only for deploying and storing their libraries in Maven Central, but not for dependency management or further maintenance tasks.

In this work, we are interested in studying libraries that have multiple versions and utilize Maven regularly to manage and update their dependencies, i.e., libraries belonging to category (iii) in Table I. Figure 3 shows the distribution of the number of versions for the libraries in this category. The minimum and maximum number of versions are respectively 2 and more than 2,000, precisely, 2,122. Meanwhile, the 1st-Q and 3rd-Q are around 5 and 200 versions respectively.

In order to conduct our empirical study on a representative dataset, we choose [1st-Q, 3rd-Q] as a range of number of versions. Therefore, this study focuses on all the libraries with between 5 and 200 versions. This accounts for 73,653 libraries and 1,487,956 versions, representing 32.96% and 61.81% of the total number of libraries and version in Maven Central at the `SNAPSHOT` time.

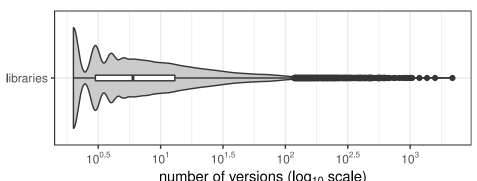



Fig. 3. Distribution of the number of library versions in Maven Central.

## IV. RESULTS

In this section, we address our research questions and present the results obtained.

### A. *RQ1*: *To what extent are the different library versions actively used?*

To answer RQ1, we study the activity status of libraries and versions in Maven Central. Table II shows the numbers and percentages of active, passive and dormant libraries and versions. We observe a low percentage of active versions (14.73%), whereas there is a predominant number of passive ones (85.27%), of which more than a half are dormant (45.16%). On the other hand, we can notice that the majority of libraries are active (95.49%), i.e., have at least one of its versions active. Meanwhile, passive libraries represent nearly 5% of the total number of libraries, of which (∼4%) are dormant.

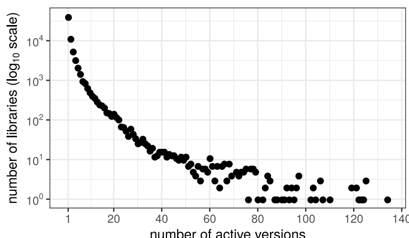



Fig. 4. Distribution of the number of active versions across active libraries

TABLE II
ACTIVITY STATUS OF LIBRARIES AND VERSIONS IN THE STUDY SUBJECTS

| Status | #Versions (%) | #Libraries (%) |
|---|---|---|
| Active | 219,184 (14.73%) | 70,337 (95.49%) |
| Passive non-dormant | 596,776 (40.11%) | 387 (0.53%) |
| Dormant | 671,996 (45.16%) | 2,929 (3.98%) |
| Total | 1,487,956 (100%) | 73,653 (100%) |

We are intrigued by the 2,929 dormant libraries. The median number of versions in this family of libraries is 9 with a maximum of 150 versions. We noticed that most of them are in-house utility libraries, intended for custom logging or testing, e.g., '*com.twitter:util-benchmark_2.11.0*'. Other libraries are archetypes[5], e.g., '*io.fabric8.archetypes:karaf-cxf-rest-archetype*'. These libraries are not intended to be used in production. Their custom nature makes them used rather internally, or by the library maintainers themselves.

In Table II, we also observe that a low proportion of versions are active 219,184 (14.73%), yet they are distributed across a very high number of libraries, 70,337 (95.49%), making these libraries active. Figure 4 summarizes the distribution of active versions in active libraries. We observe that more than a half of active libraries, 40,233 in total, have only one active version. The remainder, 30,104 libraries, have more than one active version. For some libraries, such as '*org.hibernate:hibernate-core*', more than 100 versions are currently active. However,

[5]https://maven.apache.org/guides/introduction/introduction-to-archetypes

the number of libraries with more than $100$ active versions represents less than $2\%$ of the total. More interestingly, we notice that $17\%$ of the libraries have active versions belonging to more than one different major releases (e.g., 2.X.X). For instance, the library '*activemq:activemq*' has two active major versions: 3.X.X and 4.X.X, whereas '*com.spotify:docker-client*' has seven active versions: from 2.X.X up to 8.X.X.

Figure 5 shows the lifespan distribution of active and passive versions. To avoid the bias introduced by the SNAPSHOT time constraint, we consider only non-latest active versions of libraries ($v \notin \mathtt{LATESTS}$). As we can see from the figure, the lifespan of passive versions is approximately distributed between $8$ and $80$ days (1st-Q and 3rd-Q), whereas, this range is larger for active versions: between $351$ and $1,626$ days. This conveys that versions that are active for more than $80$ days are likely to remain active for a longer period. Subsequently, these libraries are likely to be popular and widely used. Finally, we notice that the median number of days a version spends after its creation before being used for the first time is $14$, with a mean of $57.61$. This suggests that versions that have been dormant for less than $57$ days are likely to become active; beyond this time period, they are likely to remain dormant.

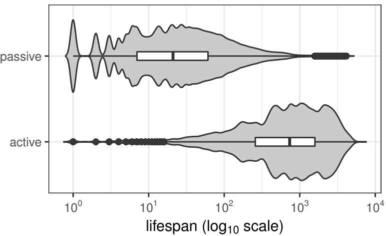



Fig. 5. Distributions of the lifespan (in days) of passive and active versions

**Findings from RQ1:** More than $40\%$ of libraries in Maven Central have strictly more than one active version, while almost $4\%$ of the libraries have never been used. This hints on an inclusive, immutable repository that can support the emergence of a diversity of library usages.

### B. *RQ2: How are the actively used versions distributed along the history of a library?*

According to Metric 1, active libraries have at least one active version. In this research question, we focus on understanding how these active versions are distributed across the different library releases.

Figure 6 shows the positional distribution of all the active versions in the libraries. Since libraries can have different number of versions, we use a normalized relative index lying between $[0,1]$, where $0$ and $1$ represent the indexes of the first and last versions of the library, respectively. First of all, we observe that active versions are scattered across different positional indices. While $68.4\%$ of active library versions are almost evenly distributed across the non-latest releases, a significant number of active versions, precisely $69,146$ ($31.6\%$), are latest versions. This result is inline with the current policies of dependency management systems, which recommend upgrading to latest dependencies.

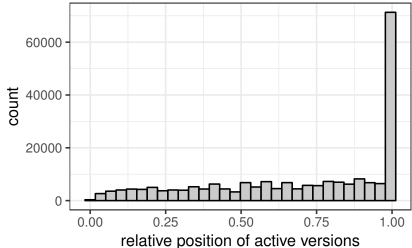



Fig. 6. Positional distribution of active versions (#bins = 30).

Digging further, we investigate the transitional distribution of active and passive versions. To do this, we transform each library $l \in \mathtt{LIBS}$ into a vector, $\mathcal{S}_l$, capturing the passive/active status corresponding to all of its versions. Our objective is to analyze the occurrence of common transitional patterns between active and passive versions.

Let $\mathcal{S}_l$ be a vector representing the activity status of library versions ordered by $<_v$ (i.e. ordered by version number). The status corresponding to a version $v$ is P if $isPSV(v)$ is $TRUE$ and A otherwise. For example, the library *com.google.guava:guava-jdk5* has a total of five versions, i.e., $\mathcal{S}_{guava-jdk5}$ = [A,A,P,P,A]. Considering that we are particularly interested in transitional patterns, the consecutive versions with the same status can be compressed to a single status, e.g., the previous example is represented as [A,P,A].

TABLE III
THE TOP-7 MOST COMMON TRANSITIONAL PATTERNS

| Pattern | Frequency | Example |
|---|---|---|
| [P,A] | $43,549$ | *commons-codec:commons-codec* |
| [P,A,P,A] | $10,219$ | *org.apache.commons:commons-lang3* |
| [P,A,P,A,P,A] | $3,478$ | *org.jboss.logging:jboss-logging* |
| [A,P, A] | $2,761$ | *com.google.guava:guava-jdk5* |
| [P,A,P,A,P,A,P,A] | $1,592$ | *org.joda:joda-convert* |
| [A,P,A,P,A] | $1,343$ | *com.google.inject:guice* |
| [P,A,P] | $613$ | *org.springframework:spring-webflow* |

We obtained a total of $94$ different transitional patterns. Table III shows the frequency of appearance of the seven most common of them. As expected, the $92\%$ of the patterns are finishing by an A. The most frequent pattern is [P,A], i.e., old versions are passive and the latest ones are active. Yet, the remaining patterns represent more than $40\%$ of the libraries. The rest of libraries follow a pattern where some old versions are also active. In extreme cases, the latest version of the library is passive (patterns finishing with a P). In such cases, we observe that most of their clients use an older version with the same major version number. We

speculate that this behavior is due to the clients' belief that the version they use is rather stable. Similar findings have been reported by Kula et al. [12]. We also observe that $5.5\%$ of the libraries have their earliest version active. It is interesting to note that many of them are very popular libraries, e.g., '*org.hamcrest:hamcrest-core*' and '*org.apache.ant:ant*'.

**Findings from RQ2:** $31.6\%$ of active versions are latest and the remaining $68.4\%$ of active versions are evenly distributed across the libraries' history. When the clients do not use the latest version, they often depend on earlier versions belonging to the same major release of the library.

*C.* ***RQ3****: Among the actively used versions of a library, is there one or several versions that are significantly more popular than the others?*

In this research question, we investigate the diversity in the popularity of library versions. We assess the popularity of a library versions using Metric 3. In particular, we are interested in identifying significantly popular versions and analyzing the positional distribution of these versions. For this aim, we use the Tukey's outlier detection method [19] to identify versions with a popularity score that is far greater than the remaining versions of the library.

We distinguish between three different classes of libraries: (i) libraries that do not have a significantly most popular version $(55,148)$, (ii) libraries with one significantly popular version $(9,622)$, and (iii) libraries with more than one significantly popular version $(8,883)$. The first class (i) represents libraries with versions that have a similar number of usages. The classes (ii) and (iii) represent libraries with one or more versions that have attracted more users compared to the rest of their versions. A large number of the users of significantly popular versions are different versions of the same library. These are library providers that may have remained loyal to one version despite the release of newer versions. To our surprise, almost all the significantly popular versions are active, only $86$ out of $143,334$ are passive. For instance, '*com.amazonaws:aws-java-sdk:1.11.409*' is significantly popular and passive.

Figure 7 shows illustrative examples, Apache IO, JUnit, and XML APIs, each one corresponding to one of these three classes. The horizontal dashed line in each frame represents the outlier's threshold of the library. All the versions that lie above this line are considered significantly popular. As shown in the figure, although the version $2.4$ of Apache IO is quite old, it is still the most popular release of this library in Maven Central. In the case of JUnit, it has two significantly popular versions: $4.11$ and $4.12$. On the other hand, the library XML APIs does not have any significantly popular version (i.e., the popularity of its versions remains steady across time).

In order to measure the positional distribution of popular versions, we focus on libraries that have at least one significantly popular version. We determine the relative position of such versions with respect to the number of versions of the

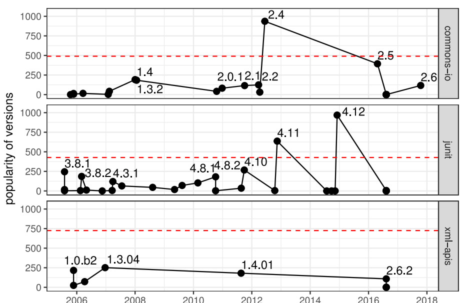



Fig. 7. Evolution of the popularity of versions ($pop_{\mathcal{V}}(v)$ metric) corresponding to the libraries Apache Commons IO, JUnit and XML APIs.

library. As for the positional distribution of active version, we also normalize the relative position between $[1, 0]$. The histogram in Figure 8 shows the distribution of the positions of the most popular versions across libraries. We observe that less than $10\%$ of libraries have their latest version as the most popular. This is expected since the average lifespan of latest versions is lower than the average of non-latest versions. Interestingly, we found that the remaining highly popular versions are almost equally distributed, between $2\%$ and $5\%$, in the remaining positions.

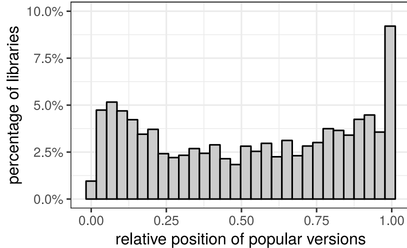



Fig. 8. Histogram of the positional distribution of significantly popular library versions (#bins = 30).

This result indicates that the most popular libraries in Maven Central are distributed across all the different library releases. It is notable that for almost $85\%$ of libraries the most used version is not the latest. Thus, older versions are still being heavily used by other libraries, with the exception of the first version which is rarely the most popular.

**Findings from RQ3:** $17\%$ of the libraries have more than one significantly popular version distributed across different releases, each of which creates a niche fitting a group of users. This indicates that library developers successfully address the needs of diverse populations of users.

*D. **RQ4**: Does the number of actively used versions relate to the popularity of a library?*

We have seen so far that many libraries in Maven Central have multiple active versions, of which more than one can be significantly more popular than the others. Now, we investigate whether the activity status of versions has a direct effect on the popularity of their corresponding library. For this, we calculate the percentages of active and passive versions of each library and compare them with respect to the overall popularity of the library.

Figure 9 shows the smoothing function corresponding to the relation between the popularity of libraries and their percentages of active versions. There is a significant positive correlation between both variables (Spearman's rank correlation test: $\rho = 0.87$, p-value $< 0.01$). In particular, we observe that libraries that have more than $50\%$ of active versions are more likely to be very popular, as popular libraries with many versions attract more clients for their versions.

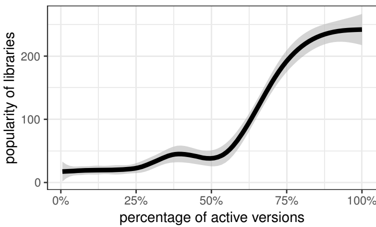



Fig. 9. Fitting curve (GAM model) of the percentage of active versions w.r.t. the popularity of libraries ($pop_{\mathcal{L}}(l)$ metric). The shaded area around the fitting curve represents the $95\%$ confidence interval.

Table IV shows the seven most popular libraries ranked in decreasing order of popularity, as well as their percentage of active and significantly popular versions. As we can see, in all the cases a significant proportion of their versions are active. This indicates that many versions of these libraries continue being actively used, contributing to the popularity of the library and adding dependency diversity among all the clients. In three cases out of seven, there are more than two versions that are significantly more popular than the others. Finally, we also notice that these popular libraries serve general purposes, which allow them to fit well for various types of usages.

**Findings from RQ4:** Popular libraries in Maven Central have most of their versions active and serve general purposes. Moreover, the popularity of a library can be estimated by the number of its active versions. The more active versions a library has, the more likely it is to be popular, and vice-versa.

*E. **RQ5**: How timely are the different library versions in Maven Central?*

This research question focuses on the temporal dimension of the dataset. We analyze whether the diversity of popular and active versions that we observe today is a phenomenon that sustained in the past history of the libraries. We look at every single library version $v$ separately and investigate whether, during the time period when $v$ was the latest, it gained the expected attraction among its older peers. We compare the number of usages that a version $v$ gets during its lifespan period against the number of usages that the whole library received during the timeliness period of $v$. For this comparison, we rely on the timeliness function described in Metric 4. This metric can be considered as an internal popularity metric that assesses the popularity of a version among its peers.

TABLE IV
THE TOP-7 MOST POPULAR LIBRARIES IN OUR STUDY SUBJECTS AND THEIR NUMBER OF ACTIVE AND SIGNIFICANTLY POPULAR VERSIONS

| Library | Domain | #Active (%) | #Popular (%) |
|---|---|---|---|
| *google.code.findbugs:jsr305* | Utility | 10 (90%) | 1 (9.01%) |
| *org.slf4j:slf4j-api* | Logging | 63 (86.3%) | 3 (4.1%) |
| *log4j:log4j* | Logging | 18 (94.7%) | 1 (5.2%) |
| *com.google.guava:guava* | Utility | 71 (79.7%) | 1 (1.2%) |
| *junit:junit* | Testing | 27 (96.5%) | 2 (7.1%) |
| *org.hamcrest:hamcrest-core* | Testing | 5 (100%) | 1 (20%) |
| *commons-logging:logging* | Logging | 15 (88.3%) | 2 (11.8%) |

Overall, for all our study subjects, $70.6\%$ of library versions are under-timely (including dormant versions), while $19.8\%$ are timely, and the remaining $9.6\%$ are over-timely. Figure 10 shows the distribution of the three timeliness classes for active and passive versions. We observe that roughly $45\%$ of passive library versions were under-timely. These are versions that did not attract users for their library throughout their timeliness period. Meanwhile, almost $55\%$ are timely. These are library versions that were not only active at some point, but also widely used. This gives substantial evidence that the diversity that we observe today has existed in the past in Maven Central. On the other hand, we observe that $55.3\%$ of active versions are under-timely. These are versions that are not widely popular among their peers, yet active. The average lifespan of these versions is $\sim$777 days, which suggests that although they are under-timely, they are likely to remain active for a long period of time; whereas, the remaining active versions are evenly distributed among timely and over-timely.

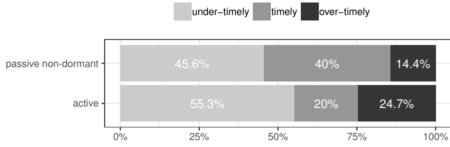



Fig. 10. Proportions of timeliness classes for passive and active versions.

In order to analyze the distribution of the timeliness classes at the library level, we calculate the proportions of under-timely, timely and over-timely versions in each library. Figure 11 shows a ternary diagram [20] representing the distribution of the three timeliness classes across the study subjects. In the figure, each point represents a library. In

general, we observe a high dispersion in the space of libraries, meaning that there are representative cases for almost all of the different proportions of classes. The paired correlation tests between the proportions of each of the classes and the popularity of their corresponding library reveal that none of the correlations are statistically significant (p-value > $0.05$ according to the Spearman's test). Therefore, the proportions of the timeliness of the versions of a library are not directly related with the popularity of the library.

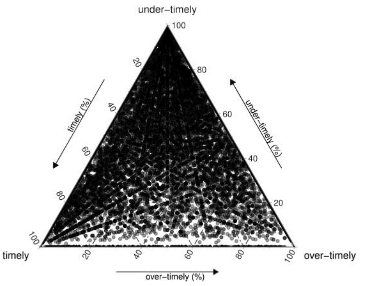



Fig. 11. Distribution of libraries w.r.t. their percentages of over-timely, timely and under-timely versions. The dispersion of points inside the triangle indicates that the proportions of classes are well distributed across the libraries.

**Findings from RQ5:** The diversity in the usage and popularity of versions has consistently sustained during the history of Maven Central. We observe that $\sim$10% of all the library versions attract new users during their timeliness period and remain active even after the next version has been released. Meanwhile, there is no correlation between the popularity of a library and the timeliness of its versions.

## V. Discussion

In this section, we discuss the implications of our findings about the emergence of software diversity in Maven Central, as well as some threats to the validity of our results.

### *A. Supporting the Emergence of Software Diversity*

This study focuses on the diversity of usages of libraries and versions in Maven Central. We have observed empirically how the immutability of versions, which is a characteristic enforced by design in Maven Central, supports the natural emergence of software diversity [3]. This diversity takes multiple forms and has various effects:

- all active libraries have strictly more than one active version, and the 42.7% of them have more than two active versions;
- 17% of the libraries have two or more versions that are significantly more popular than the others, which indicates a very rich diversity in usages of the latest library releases and may imply that the latest library versions deployed on Maven Central use different versions of a similar library;
- the most popular libraries are also the ones that have the largest proportion of active versions;
- the existence of multiple used versions that overlap in time is a common phenomenon in the history of all libraries.

We interpret these multiple forms of diversity in usage and popularity of libraries as follows: a repository that offers the opportunity for users to choose their dependencies, naturally supports the emergence of diversity among these dependencies. In other words, this massive emergent diversity is not only due to users who forget to update their dependencies. Many users decide very explicitly to depend on one or the other version of a library because it perfectly fits their needs. Consequently, this kind of diversity emerges in a fully decentralized and unsupervised manner.

Our study also highlights some important challenges for a repository that supports diversification. First, there is a cost for the maintainers of Maven Central. We have observed that, although most libraries are actively used ($95.49\%$), only $14.73\%$ of the Maven artifacts are used. We have also noticed that some companies use Maven Central to store artifacts that nobody else uses ($45.1\%$ of versions are dormant). Consequently, keeping all versions induces an overhead in hardware and software resources. Second, there is a cost for the developers of popular libraries who need to maintain several versions of their library to serve different clients. Third, there is a risk that users decide to keep a dependency towards a vulnerable or flawed version.

The trade-off between healthy levels of diversity in a system (here, the Maven Central ecosystem) and the challenges of redundancy and noise is necessary and very natural. Biological studies insist on the importance of keeping less fit or even unexpressed genes as genetic material that is necessary in order to adapt to unpredictable environmental changes [21], [22]. Our study reveals that the immutability of Maven artifacts provides the material for libraries to eventually fit the needs of various users, which eventually results in the emergence of diverse popular and timely versions. In the same way that biological systems do, library maintainers can accommodate the overhead of manual updates and conflict management in order to contribute to the sustainability of the massively large pool of software diversity that exists in Maven Central.

### *B. Threats to Validity*

We report about internal, external, and reliability threats to the validity of our study.

*a) Internal validity:* The internal threat relates to the metrics employed, especially those to compare the popularity of libraries and versions. In this work, we characterize popularity in terms of number of usages and quantify it based on well-known graph-based metrics [23]. Thus, we assume that a widely reused library is a popular one, and we consider only the relationships described in Maven Central, which do not take into account usages from private projects. The jOOQ library is one example among others. Because it is dual-license, many OSS libraries avoid to depend on it, but other

closed-source software are still using it and there is no way to quantify their number. However, as suggested in previous studies, software popularity can be measured in a variety of ways, depending on different factors such as social or technical aspects [24]. Another concern relates to the fact that conventions on semantic versioning are not really taken well into account by library maintainers [25]. Still, we believe that at the scale of the dataset employed in this study, our metrics are a fair approximation of the state of practice in Maven Central.

*b) External validity:* Our results might not generalize to other software repositories beyond the Maven Central ecosystem (e.g., npm, RubyGems or CRAN). It should also be noticed that Maven Central does not perform any real vetting of the people that deploy artifacts or on the quality of such artifacts. Thus, the integrity and origin of most of our study subjects therein is not known or verifiable. Moreover, this work takes into account version ordering as well as temporal ordering relationships, which we believe are sufficient to give a plausible representation of the way that libraries are updated as well as their evolution trends.

*c) Reliability validity:* Our results are reproducible, the dataset used in this study is publicly available online[6]. Moreover, we provide all necessary code[7] to replicate our analysis, including Cypher queries and R notebooks.

## VI. Related Work

This paper is related to a long line of previous works about mining software repositories and analysis of dependency management systems. In this section we discuss the related work along the following aspects.

*a) Structure and updating behavior:* Over the past years, several research papers have highlighted the benefits of leveraging graph-based representations and ecologycal principles to analyze the architecture of large-scale software systems [26]–[29]. Raemaekers et al. [30] investigated the adherence to semantic versioning principles in Maven Central as well as the update trends of popular libraries. They found that the presence of breaking changes has little influence on the actual delay between the availability of a library and the use of the newer version. Kula et al. [12] study the latency in trusting the latest release of a library and propose four types of dependency adoptions according to the dependency declaration time. De Castilho et al. [31] use the Maven Central repository for automatically selecting and acquiring tools and resources to build efficient NLP processing pipelines. Their analysis relied partially on Maven build files to collect library dependencies in industrial systems. However, as far as we know, none of the existing works have studied the repercussion of the artifacts' immutability at the scale of the entire Maven Central repository.

*b) Analysis of evolution trends:* The evolution of software repositories is a popular and widely-researched topic in the area of empirical software engineering. Recently, Decan et al. [32] perform a comparison of the similarities and differences between seven large dependency management systems based on the packages gathered and archived in the *libraries.io* dataset. They observe that dependency networks tend to grow over time and that a small number of libraries have a high impact on the transitive dependencies of the network. Kikas et al. [33] study the fragility of dependency networks of JavaScript, Ruby, and Rust and report on the overall evolutionary trends and differences of such ecosystems. Abdalkareem et al. [34] investigate about the reasons that motivate developers to use trivial packages on the npm ecosystem. Raemaekers et al. [35] construct a Maven dataset to track the changes on individual methods, classes, and packages of multiple library versions. Our work expands the existing knowledge in the area by showing how software repositories can contribute to prevent dependency monoculture by making available a more diverse set of library versions for software reuse.

*c) Security and vulnerability risks:* Researchers have investigated and compared dependency issues across many packaging ecosystems. Suwa et al. [11] investigate the occurrence of rollbacks during the update of libraries in Java projects. Their results confirm previous studies that show that library migrations have no clear patterns and in many cases, the latest available version of a library is not always the most used [36], [37]. Mitropoulos et al. [38] present a dataset composed of bugs reports for a total of $17,505$ Maven projects. They use FindBugs to detect numerous types of bugs and also to store specific metadata together with the FindBugs results. Zapata et al. [39] compare how library maintainers react to vulnerable dependencies based on whether or not they use the affected functionality in their client projects. Our work considers security and vulnerability risks in software repositories from a novel perspective, i.e., by taking into account the benefits and drawbacks that come with the emergence of software diversity.

## VII. Conclusion

In this paper, we performed an empirical study on the diversity of libraries and versions in the Maven Central repository. We studied the activity, popularity and timeliness of $1,487,956$ artifacts that represent all the versions of $73,653$ libraries. We defined various graph-based metrics based on the dependencies among Maven artifacts that are captured in the Maven Dependency Graph [1]. We found that $\sim$40% of libraries have two or more versions that are actively used, while almost $4\%$ never had any user in Maven Central. We also found that more than $90\%$ of the most popular versions are not the latest releases, and that both active and significantly popular versions are distributed across the history of library versions. In summary, we presented quantitative empirical evidence about how the immutability of artifacts in Maven Central supports the emergence of natural software diversity, which is fundamental to prevent dependency monoculture during software reuse. Our next step is to investigate how we can amplify this natural emergence of software diversity through dependency transformations at the source code level.

[6]https://doi.org/10.5281/zenodo.1489120

[7]https://github.com/castor-software/oss-graph-metrics/tree/master/maven-central-diversity

ACKNOWLEDGMENTS

This work has been partially supported by the EU Project STAMP ICT-16-10 No.731529, by the Wallenberg Autonomous Systems and Software Program (WASP), by the TrustFull project financed by the Swedish Foundation for Strategic Research and by the OSS-Orange-Inria project.

REFERENCES

[1] A. Benelallam, N. Harrand, C. Soto-Valero, B. Baudry, and O. Barais, "The Maven Dependency Graph: a Temporal Graph-based Representation of Maven Central," in *16th International Conference on Mining Software Repositories (MSR)*, (Montreal, Canada), IEEE/ACM, 2019.

[2] M. Stamp, "Risks of monoculture," *Commun. ACM*, vol. 47, pp. 120–, Mar. 2004.

[3] B. Baudry and M. Monperrus, "The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond," *ACM Computing Survey*, vol. 48, no. 1, pp. 16:1–16:26, 2015.

[4] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the Dependency Conflicts in my Project Matter?," in *26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 319–330, 2018.

[5] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?," *Empirical Software Engineering*, vol. 23, pp. 384–417, Feb 2018.

[6] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable Open Source Dependencies: Counting Those That Matter," in *12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, (New York, NY, USA), pp. 42:1–42:10, ACM/IEEE, 2018.

[7] K. Jezek, J. Dietrich, and P. Brada, "How Java APIs Break – An Empirical Study," *Information and Software Technology*, vol. 65, pp. 129–146, 2015.

[8] J. H. Lala and F. B. Schneider, "It monoculture security risks and defenses," *IEEE Security & Privacy*, vol. 7, no. 1, pp. 12–13, 2009.

[9] I. Gashi, P. Popov, and L. Strigini, "Fault tolerance via diversity for off-the-shelf products: A study with sql database servers," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, pp. 280–294, Oct 2007.

[10] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds: Exploiting the intrinsic redundancy of web applications," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, pp. 16:1–16:42, May 2015.

[11] H. Suwa, A. Ihara, R. G. Kula, D. Fujibayashi, and K. Matsumoto, "An Analysis of Library Rollbacks: A Case Study of Java Libraries," in *24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pp. 63–70, Dec 2017.

[12] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest maven release," in *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, (Montreal, Canada), pp. 520–524, 2015.

[13] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, pp. 1275–1317, Oct 2015.

[14] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining Trends of Library Usage," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, IWPSE-Evol '09, (New York, NY, USA), pp. 57–62, ACM, 2009.

[15] R. G. Kula, D. M. German, T. Ishio, A. Ouni, and K. Inoue, "An Exploratory Study on Library Aging by Monitoring Client Usage in a Software Ecosystem," in *24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 407–411, 2017.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank Citation Ranking: Bringing Order to the Web," Technical Report 1999-66, Stanford InfoLab, November 1999.

[17] W. Xing and A. Ghorbani, "Weighted PageRank Algorithm," in *Proceedings of the Second Annual Conference on Communication Networks and Services Research (CNSR)*, pp. 305–314, May 2004.

[18] P. Boldi, M. Santini, and S. Vigna, "PageRank As a Function of the Damping Factor," in *14th International Conference on World Wide Web (WWW)*, (New York, NY, USA), pp. 557–566, ACM, 2005.

[19] J. Tukey, "Exploratory data analysis," 1977.

[20] N. E. Hamilton and M. Ferry, "ggtern: Ternary diagrams using ggplot2," *Journal of Statistical Software*, vol. 87, no. CN 3, pp. 1–17, 2018.

[21] R. Frankham, "Genetics and extinction," *Biological Conservation*, vol. 126, no. 2, pp. 131–140, 2005.

[22] A. A. Sawant, R. Robbes, and A. Bacchelli, "On the reaction to deprecation of clients of 4 + 1 popular java apis and the jdk," *Empirical Software Engineering*, vol. 23, pp. 2158–2197, Aug 2018.

[23] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components based on Use Relations," *IEEE Transactions on Software Engineering*, vol. 31, pp. 213–225, March 2005.

[24] A. Zerouali, T. Mens, G. Robles, and J. Gonzalez-Barahona, "On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2019.

[25] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the maven repository," *Journal of Systems and Software*, vol. 129, pp. 140 – 158, 2017.

[26] R. Ramler, G. Buchgeher, C. Klammer, M. Pfeiffer, C. Salomon, H. Thaller, and L. Linsbauer, "Benefits and drawbacks of representing and analyzing source code and software engineering artifacts with graph databases," in *""* (D. Winkler, S. Biffl, and J. Bergsmann, eds.), pp. 125–148, Springer, 2019.

[27] F. Mancinelli, J. Boender, R. D. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions," in *21st International Conference on Automated Software Engineering (ASE)*, pp. 199–208, IEEE/ACM, Sep. 2006.

[28] T. Mens, M. Claes, and P. Grosjean, "ECOS: Ecological Studies of Open Source Software Ecosystems," in *Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 403–406, Feb 2014.

[29] T. Mens and P. Grosjean, "The ecology of software ecosystems," *Computer*, vol. 48, pp. 85–87, Oct 2015.

[30] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning versus Breaking Changes: A Study of the Maven Repository," in *14th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 215–224, Sept 2014.

[31] R. E. de Castilho and I. Gurevych, "A broad-coverage collection of portable nlp components for building shareable analysis pipelines," in *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT*, pp. 1–11, 2014.

[32] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, pp. 1–36, 2018.

[33] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," in *14th International Conference on Mining Software Repositories (MSR)*, (Buenos Aires, Argentina), pp. 102–112, IEEE/ACM, May 2017.

[34] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why Do Developers Use Trivial Packages? An Empirical Case Study on npm," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, (New York, NY, USA), pp. 385–395, ACM, 2017.

[35] S. Raemaekers, A. van Deursen, and J. Visser, "The Maven Repository Dataset of Metrics, Changes, and Dependencies," in *10th IEEE Working Conference on Mining Software Repositories (MSR)*, (San Francisco, CA, USA), pp. 221–224, IEEE, ACM, 2013.

[36] C. Teyton, J.-R. Falleri, M. Palyart, and X. Blanc, "A Study of Library Migrations in Java," *Journal of Software: Evolution and Process*, vol. 26, no. 11, pp. 1030–1052, 2014.

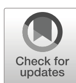
[37] C. Teyton, J.-R. Falleri, and X. Blanc, "Mining Library Migration Graphs," in *19th Working Conference on Reverse Engineering (WCRE)*, (Kingston, Canada), pp. 289–298, October 2012.

[38] D. Mitropoulos, V. Karakoidas, P. Louridas, G. Gousios, and D. Spinellis, "The Bug Catalog of the Maven Ecosystem," in *11th Working Conference on Mining Software Repositories (MSR)*, (New York, NY, USA), pp. 372–375, ACM, 2014.

[39] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards Smoother Library Migrations: A Look at Vulnerable Dependency Migrations at Function Level for npm JavaScript Packages," in *34th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 559–563, IEEE, 2018.

# Paper II

Empirical Software Engineering (2021) 26: 45
https://doi.org/10.1007/s10664-020-09914-8




# A comprehensive study of bloated dependencies in the Maven ecosystem

César Soto-Valero[1] · Nicolas Harrand[1] · Martin Monperrus[1] · Benoit Baudry[1]

Accepted: 23 September 2020 / Published online: 25 March 2021
© The Author(s) 2021

**Abstract**

Build automation tools and package managers have a profound influence on software development. They facilitate the reuse of third-party libraries, support a clear separation between the application's code and its external dependencies, and automate several software development tasks. However, the wide adoption of these tools introduces new challenges related to dependency management. In this paper, we propose an original study of one such challenge: the emergence of bloated dependencies. Bloated dependencies are libraries that are packaged with the application's compiled code but that are actually not necessary to build and run the application. They artificially grow the size of the built binary and increase maintenance effort. We propose DepClean, a tool to determine the presence of bloated dependencies in Maven artifacts. We analyze 9,639 Java artifacts hosted on Maven Central, which include a total of 723,444 dependency relationships. Our key result is as follows: 2.7% of the dependencies directly declared are bloated, 15.4% of the inherited dependencies are bloated, and 57% of the transitive dependencies of the studied artifacts are bloated. In other words, it is feasible to reduce the number of dependencies of Maven artifacts to 1/4 of its current count. Our qualitative assessment with 30 notable open-source projects indicates that developers pay attention to their dependencies when they are notified of the problem. They are willing to remove bloated dependencies: 21/26 answered pull requests were accepted and merged by developers, removing 140 dependencies in total: 75 direct and 65 transitive.

**Keywords** Dependency management · Software reuse · Debloating · Program analysis

## 1 Introduction

Software reuse, a long time advocated software engineering practice (Naur and Randell 1969; Krueger 1992), has boomed in the last years thanks to the widespread adoption of build automation and package managers (Cox 2019; Soto-Valero et al. 2019). Package managers provide both a large pool of reusable packages, a.k.a. libraries, and systematic ways to

Communicated by: Gabriele Bavota

✉ César Soto-Valero
cesarsv@kth.se

[1] KTH Royal Institute of Technology, Stockholm, Sweden

declare what are the packages on which an application depends. Examples of such package management systems include Maven for Java, npm for Node.js, or Cargo for Rust. Build tools automatically fetch all the packages that are needed to compile, test, and deploy an application.

Package managers boost software reuse by creating a clear separation between the application and its third-party dependencies. Meanwhile, they introduce new challenges for the developers of software applications, who now need to manage those third-party dependencies (Cox 2019) to avoid entering into the so-called "dependency hell". These challenges relate to ensuring high quality dependencies (Salza et al. 2019), keeping the dependencies up-to-date (Bavota et al. 2015), or making sure that heterogeneous licenses are compatible (Wu et al. 2017).

Our work focuses on one specific challenge of dependency management: the existence of *bloated dependencies*. This refers to packages that are declared as dependencies for an application, but that are actually not necessary to build or run it. The major issue with bloated dependencies is that the final deployed binary file includes more code than necessary: an artificially large binary is an issue when the application is sent over the network (e.g., web applications) or it is deployed on small devices (e.g., embedded systems). Bloated dependencies could also embed vulnerable code that can be exploited, while being actually useless for the application (Gkortzis et al. 2019). Overall, bloated dependencies needlessly increase the difficulty of managing and evolving software applications.

We propose a novel, unique, and large scale analysis of bloated dependencies. So far, research on bloated dependencies has been only touched with a study of copy-pasted dependency declarations by McIntosh et al. (2014), and a study of unused dependencies in the Rust ecosystem by Hejderup et al. (2018). Our previous work gives preliminary results on this topic in the context of Java (Harrand et al. 2019). Yet, there is no systematic analysis of the presence of bloated dependencies nor about the importance of this problem for developers in the Java ecosystem.

Our work focuses on Maven, the most popular package manager and automatic build system for Java and languages that compile to the JVM. In Maven, developers declare dependencies in a specific file, called the POM file. In order to analyze thousands of artifacts on Maven Central, the largest repository of Java artifacts, manual analysis is not a feasible solution. To overcome this problem, we have developed DepClean, a tool that performs an automatic analysis of dependency usage in Maven projects. Given an application and its POM file, DepClean collects the complete dependency tree (the list of dependencies declared in the POM, as well as the transitive dependencies) and analyzes the bytecode of the artifact and all its dependencies to determine the presence of bloated dependencies. Finally, DepClean generates a variant of the POM in which bloated dependencies are removed.

Armed with DepClean, we structured our analysis of bloated dependencies in two parts. First, we automatically analysed 9,639 artifacts and their 723,444 dependencies. We found that 75.1% of these dependencies are bloated. We identify transitive dependencies and the complexities of dependency management in multi-module projects as the primary causes of bloat. Second, we performed a user study involving 30 artifacts, for which the code is available as open-source on GitHub and which are actively maintained. For each project, we used DepClean to generate a POM file without bloated dependencies and submitted the changes as a pull request to the project. Notably, our work yielded 21 merged pull requests by open-source developers and 140 bloated dependencies were removed.

To summarize, this paper makes the following contributions:

- A comprehensive study of bloated dependencies in the context of the Maven package manager. We are the first to quantify the magnitude of bloat on a large scale (9,639 Maven artifacts) showing that 75.1% of dependencies are bloated.
- A tool called DepClean to automatically analyze and remove bloated dependencies in Java applications packaged with Maven. DepClean can be used in future research on package management as well as by practitioners.
- A qualitative assessment of the opinion of developers regarding bloated dependencies. Through the submission of pull requests to notable open-source projects, we show that developers care about removing dependency bloat: 21/26 of answered pull requests have been merged, removing 140 bloated dependencies.

The remainder of this paper is structured as follows. Section 2 introduces the key concepts about dependency management with Maven and presents an illustrative example. Section 3 introduces the new terminology and describes the implementation of DepClean. Section 4 presents the research questions that drive our study, as well as the methodology followed. Section 5 covers our experimental results for each research question. Sections 6 and 7 provide a comprehensive discussion of the results obtained and present the threats to the validity of our study. Section 8 concludes this paper and provides future research directions.

## 2 Background

We provide an overview of the Maven package management system and of the essential terminology. We illustrate these concepts with a concrete example.

### 2.1 Maven Dependency Management Terminology

Maven is a popular package manager and build automation tool for Java projects and other languages that compile to the JVM (e.g., Scala, Kotlin, Groovy, Clojure, or JRuby). Maven relies on a specific build file, known as the POM (acronym for "Project Object Model"), where developers specify information about the project, its dependencies and its build process. POM files can inherit from a base POM, known as the Maven parent POM. The inheritance and declaration of dependencies is a design decision of developers.

**Maven Project** A Maven project includes source code files and build files. It can be a *single-module*, or a *multi-module* project. The former has a single POM file, which includes all the dependencies and build instructions to produce a single artifact (JAR file). The latter allows to separately build multiple artifacts in a certain order through a so-called aggregator POM. In multi-module projects, developers can define a parent POM that specifies the dependencies used by all the modules.

**Maven Artifact** We refer to *artifacts* as compiled Maven projects that have been deployed to a binary code repository. A Maven artifact is typically a JAR file that is uniquely identified with a triplet $(G{:}A{:}V)$, $G$, the *groupId*, identifies the organization that develops the artifact, $A$, the *artifactId*, is the name of the artifact, and $V$ corresponds to its *version*. Maven Central is the most popular public repository to host Maven artifacts.

**Dependency Resolution** Maven resolves dependencies in two steps: (1) based on the POM file(s), it determines the set of required dependencies, and (2) it fetches dependencies that are not present locally from external repositories such as Maven Central. Maven constructs a *dependency tree*, that captures all dependencies and their relationships. Given one artifact, we distinguish between three types of Maven dependencies: *direct* dependencies that are explicitly declared in the POM file; *inherited* dependencies, which are declared in the parent POM; and *transitive* dependencies obtained from the transitive closure of direct and inherited dependencies. Dependency version management is a key feature of the dependency resolution, which Maven handles with a specific *dependency mediation algorithm*.[1]

## 2.2 A Brief Journey in the Dependencies of the JXLS Library

We illustrate the concepts introduced previously with one concrete example: JXLS,[2] an open source Java library for generating Excel reports. It is implemented as a multi-module Maven project with a parent POM in `jxls-project`, and three modules: `jxls`, `jxls-examples`, and `jxls-poi`.

Listing 1 shows an excerpt of the POM file of the `jxls-poi` module, version 1.0.15. It declares `jxls-project` as its parent POM (lines 1 – 5) and a direct dependency towards the `poi` Apache library (lines 10–14). Figure 1 depicts an excerpt of its Maven dependency tree (we do not show testing dependencies here, such as JUnit, to make the figure more readable). Nodes in blue, pink, and yellow represent direct, inherited, and transitive dependencies, respectively, for the `jxls-poi` artifact (as reported by the `dependency:tree` Maven plugin).

**Listing 1** Excerpt of the POM file corresponding to the module `jxls-poi` of the multi-module Maven project JXLS

```
<parent>
   <groupId>org.jxls</groupId>
   <artifactId>jxls-project</artifactId>
<version>2.6.0</version>
</parent>
<artifactId>jxls-poi</artifactId>
<packaging>jar</packaging>
<version>1.0.15</version>
<dependencies>
   <dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi</artifactId>
      <version>3.17</version>
   </dependency>
   ...
</dependencies>
```

The library `jcl-over-slf4j` declares a dependency towards `slf4j-api`, version 1.7.12, which is omitted by Maven since it is already added from the `jxls-project` parent POM. On the other hand, JXLS declares dependencies to version 1.7.26 of `jcl-over-slf4j` and `slf4j-api`, but the lower version 1.7.12 was

[1] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.
[2] http://jxls.sourceforge.net

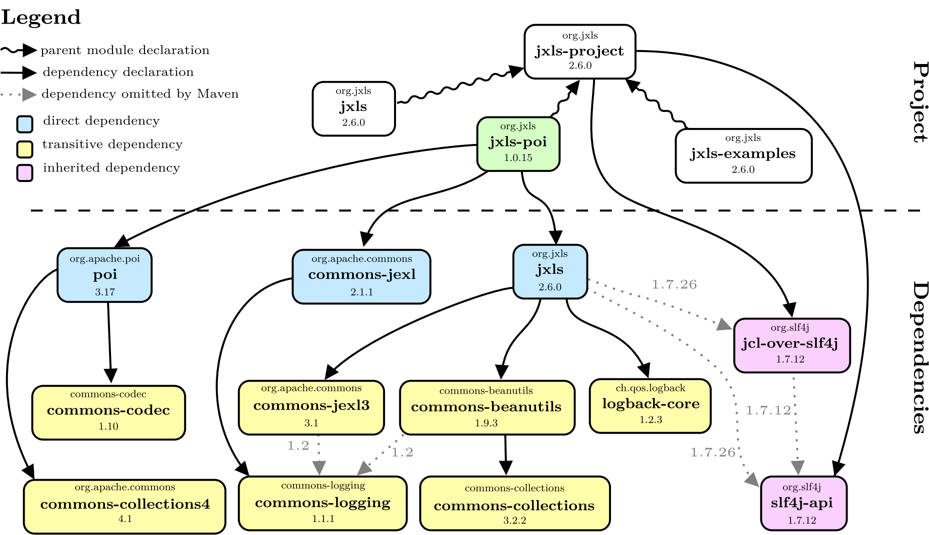



**Fig. 1** Excerpt of the dependency tree of the multi-module Maven project JXLS (dependencies used for testing are not shown for the sake of simplicity)

chosen over it since it is nearer to the root in the dependency tree and, by default, Maven resolves version conflicts with a nearest-wins strategy. Once Maven finishes the dependency resolution, the classpath of `jxls-poi` includes the following artifacts: `poi`, `commons-codec`, `commons-collections4`, `commons-jexl`, `commons-logging`, `jxls`, `commons-jexl3`, `commons-beanutils`, `commons-collections`, `logback-core`, `jcl-over-slf4j`, and `slf4j-api`. The goal of our work is to determine if all the artifacts in the classpath of Maven projects such as `jxls-poi` are actually needed to build and run those projects.

## 3 Bloated Dependencies

In this section, we introduce the idea of bloated dependency, which is the fundamental concept presented and studied in the rest of this paper. We describe our methodology to study bloated dependencies, as well as our tools to automatically detect and remove them from Maven artifacts.

Dependencies among Maven artifacts form a graph, according to the information declared in their POMs. This graph has been introduced in our previous work about the Maven Dependency Graph (MDG) (Benelallam et al. 2019). The MDG is defined as follows:

**Definition 1** (Maven Dependency Graph) The MDG is a vertex-labelled graph, where vertices are Maven artifacts (uniquely identified by their $G{:}A{:}V$ coordinates), and edges represent dependency relationships among them. Formally, the MDG is defined as $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where:

- $\mathcal{V}$ is the set of artifacts in the Maven Central repository
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ represent the set of directed edges that determine dependency relationships between each artifact $v \in \mathcal{V}$ and its dependencies

## 3.1 Novel Concepts

Each artifact in the MDG has its own Maven Dependency Tree (MDT), which is the transitive closure of all the dependencies needed to build the artifact, as resolved by Maven.

**Definition 2** (Maven Dependency Tree) The MDT of an artifact $v \in \mathcal{V}$ is a directed acyclic graph of artifacts, with $v$ as the root node, and a set of edges $\mathcal{E}$ representing dependency relationships between them.

In this work, we introduce the novel concept of *bloated dependency* as follows:

**Definition 3** (Bloated Dependency) An artifact $p$ is said to have a bloated dependency relationship $\varepsilon_b \in \mathcal{E}$ if there is a path in its MDT, between $p$ and any dependency $d$ of $p$, such that none of the elements in the API of $d$ are used, directly or indirectly, by $p$.

To reason about the bloated dependencies of an artifact, we introduce a new data structure, called the Dependency Usage Tree (DUT) as follows.

**Definition 4** (Dependency Usage Tree) The DUT of an artifact $a$, defined as $\text{DUT}_a = (\mathcal{V}, \mathcal{E}, \nabla)$, is a tree, whose nodes are the same as the Maven Dependency for $a$ and which edges are all of the $(a, a_i)$, for all nodes $a_i \in \text{DUT}_a$. A labeling function $\nabla$ assigns each edge one of the following six dependency usage types: $\nabla : \mathcal{E} \rightarrow$ {ud, ui, ut, bd, bi, bt} such that:

$$\nabla(\langle p, d \rangle) = \begin{cases} \text{ud}, & \text{if } d \text{ is used and it is directly declared by } p \\ \text{ui}, & \text{if } d \text{ is used and it is inherited from a parent of } p \\ \text{ut}, & \text{if } d \text{ is used and it is resolved transitively by } p \\ \text{bd}, & \text{if } d \text{ is bloated and it is directly declared by } p \\ \text{bi}, & \text{if } d \text{ is bloated and it is inherited from a parent of } p \\ \text{bt}, & \text{if } d \text{ is bloated and it is resolved transitively by } p \end{cases}$$

It is to be noted that, in the case of transitive dependencies, $\nabla$ assigns the bt label to the relationship $(a, a_i)$ if and only if two conditions hold: 1) $a$ does not use any member of $a_i$, and 2) none of the artifacts in the tree need to use $a_i$ to fulfill the requirements of $a$.

Given a Maven artifact $a$ we build both the $\text{MDT}_a$ and the $\text{DUT}_a$. Both trees include exactly the same set of nodes, but the edges are different. In the $\text{MDT}_a$, an edge $(a_1, a_2)$ exists when the POM of $a_1$ declares a dependency towards $a_2$. In the $\text{DUT}_a$, all edges start from $a$, and an edge $(a, a_1)$ means that $a_1$ is an artifact in the $\text{MDT}_a$. In the case $a_1$ is not

a direct dependency of $a$, then the edge $(a, a_1)$ does not exist in the $\mathrm{MDT}_a$, yet we need to model it, since it is the relation $(a, a_1)$ that can be bloated or used.

## 3.2 Example

Figure 2 illustrates the dependency usage tree for the example presented in Fig. 1. Analyzing the bytecode of `jxls-poi`, we find no references to any API member of the direct dependencies `commons-jexl` (explicitly declared in the POM) and `sl4j` (inherited from its parent POM). Therefore, these dependency relationships are labelled as bloated-direct (bd) and bloated-inherited (bi) dependencies, respectively.

Now let us consider the dependency to `commons-codec`. In the MDT of `jxls-poi` (cf. Fig. 1), we observe that `commons-codec` is a transitive dependency of `jxls-poi`, through `poi`. From the perspective of bloat, what we want to know is the following: is `commons-codec` necessary to build and run `jxls-poi`? Therefore, we are interested in the relationship between `jxls-poi` and `commons-codec`, which we model in the DUT of `jxls-poi` (cf. Fig. 2). To answer the question of usage we need two analyses. First, an analysis of the bytecode of `jxls-poi` reveals that it does not use `commons-codec` directly. Second, we observe that `jxls-poi` uses some members of `poi`, and that these members of `poi` do not use `commons-codec`. So, we conclude that the relationship between `jxls-poi` and `commons-codec` is bloated, and the corresponding edge is labelled bt. It is important to note that a bloated transitive dependency relationship between `jxls-poi` and `commons-codec` does not mean that `commons-codec` is bloated for `poi`, but only for the subpart of `poi` that is necessary for `jxls-poi`. Table 1 summarizes the labelling of all the dependency relationships of `jxls-poi`.

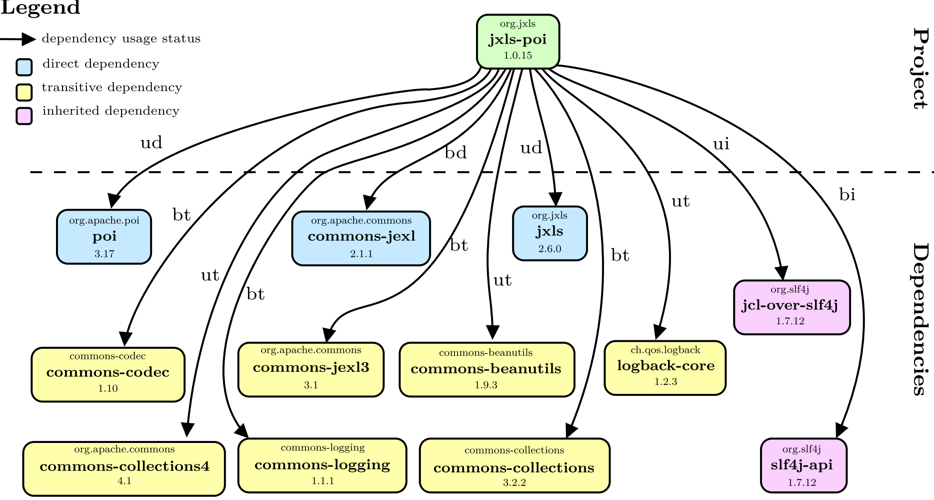



**Fig. 2** Dependency Usage Tree (DUT) for the example presented in Fig. 1. Edges are labelled according to Definition 4 to reflect the usage status between `jxls-poi` and each one of its dependencies

**Table 1** Contingency table of the different types of dependency relationships studied in this work for the example presented in Fig. 2

| | Used | Bloated |
|---|---|---|
| Direct | poi, jxls | commons-jexl |
| Inherited | jcl-over-sl4j | sl4j-api |
| Transitive | commons-beanutils,logback-core, commons-collections4 | commons-logging, commons-collections, commons-codec, commons-jexl3 |

### 3.3 DepClean: A Tool for Detecting and Removing Bloated Dependencies

For our study, we design and implement a specific tool called DepClean. An overview of DepClean is shown in Fig. 3, it works as follows. It receives as inputs a built Maven project and a repository of artifacts, then it extracts the dependency tree of

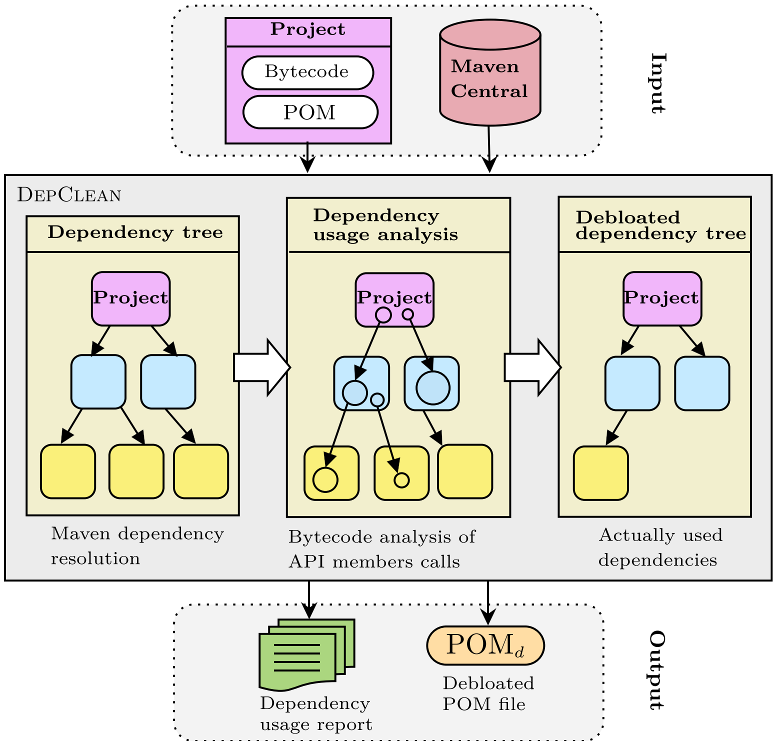



**Fig. 3** Overview of the tool DepClean to detect and remove bloated dependencies in Maven projects. Rounded squares represent artifacts, circles inside the artifacts are API members, arrows between API members represents bytecode calls between artifacts, arrows between artifacts represent dependency relationships between them

the projects and constructs a DUT to identify the set of dependencies that are actually used by the project. DEPCLEAN has two outputs: (1) it returns a report with the usage status of all types of dependencies, and (2) it produces an alternative version of the POM file ($POM_d$) with all the bloated dependencies removed (i.e., the XML node of the bloated dependency is removed). DEPCLEAN does not perform any modifications to the source code, bytecode, or configurations files in the project under consideration.

Algorithm 1 details the main procedure of DEPCLEAN. The algorithm receives as input a Maven artifact $p$ that includes a set of dependencies in its dependency tree, denoted as $DT$, and returns a report of the usage status of its dependencies and a debloated version of its POM. Notice that DEPCLEAN computes two transitive closures: over the Maven dependency tree (line 2) and over the call graph of API members (line 3).

**Algorithm 1** Main procedure to detect and remove bloated dependencies.

```
Input: A Maven artifact p.
Output: A report of the bloated dependencies of p and a clean POM file f of p
        without bloated dependencies.
1  f ← copyPOM(p);
2  DT ← getDependencyTree(p);
3  UD ← getUsedDependencies(p, DT); // refer to Algorithm 2
4  foreach d ∈ DT do
5      if d ∈ UD then
6          continue; // do nothing, the dependency is actually
               used by p
7      end
8      if isDeclared(d, f) then
9          report d as bloated-direct and remove(d, f);
10     else
11         report d as bloated-inherited or bloated-transitive, and exclude(d, f);
12     end
13 end
14 return f;
```

The algorithm first copies the POM file of $p$, resolving all its direct and transitive dependencies locally, and obtaining the dependency tree (lines 1 and 2). If $p$ is a module of a multi-module project, then all the dependencies declared in its parent POM are included as direct dependencies of $p$. Then, the algorithm proceeds to construct a set with the dependencies that are actually used by $p$ (line 3).

Algorithm 2 explains the bytecode analysis. The detection component statically analyzes the bytecode of $p$ and all its dependencies to check which API members are being referenced by the artifact, either directly or indirectly. Notice that it behaves differently if the included artifact is a direct, inherited, or a transitive dependency. If none of the API members of a dependency $d \in DT$ are called, even indirectly via transitive dependencies, then $d$ is considered to be bloated, and we proceed to remove it.

**Algorithm 2** Procedure to obtain all the dependencies used, directly or indirectly, by a Maven artifact.

```
  Input: A Maven artifact p and its dependency tree DT.
  Output: A set of dependencies UD actually used by p.
1 UD ← ∅;
2 foreach d ∈ DT do
3   if (p, d) is a direct or inherited dependency then
4     part_d ← extractMembers(p, d); // extract the subpart of d
          used by p
5     if part_d ≠ ∅ then
6       add(d, UD);
7     else
8       continue;
9     end
10  else
11    find the path l = [p, ..., d] connecting p and d in DT;
12    a ← p;
13    foreach b ∈ l_{2,...,n} do
14      part_b ← extractMembers(a, b);
15      if part_b = ∅ then
16        break;
17      else
18        a ← part_b;
19      end
20      if b = d then
21        add(d, UD);
22      end
23    end
24  end
25 end
```

In Maven, we remove bloated dependencies in two different ways: (1) if the bloated dependency is explicitly declared in the POM, then we remove its declaration clause directly (line 9 in Algorithm 1), or (2) if the bloated dependency is inherited from a parent POM or induced transitively, then we excluded it in the POM (line 11 in Algorithm 1). This exclusion consists in adding an `<exclusion>` clause inside a direct dependency declaration, with the *groupId* and *artifactId* of the transitive dependency to be excluded. Excluded dependencies are not added to the classpath of the artifact by way of the dependency in which the exclusion was declared.

DepClean is implemented in Java as a Maven plugin that extends the `maven-dependency-analyzer`[3] tool, which is actively maintained by the Maven team and officially supported by the Apache Software Foundation. For the construction of the dependency tree, DepClean relies on the `copy-dependencies` and `tree` goals of the `maven-dependency-plugin`. Internally, DepClean relies on the ASM[4] library

[3]http://maven.apache.org/shared/maven-dependency-analyzer

[4]https://asm.ow2.io

to visit all the `.class` files of the compiled projects in order to register bytecode calls towards classes, methods, fields, and annotations among Maven artifacts and their dependencies. For example, it captures all the dynamic invocations created from class literals by parsing the bytecodes in the constant pool of the classes. DEPCLEAN defines a customized parser that reads entries in the constant pool of the `.class` files directly, in case it contains special references that ASM does not support. This allows the plugin to statically capture reflection calls that are based on string literals and concatenations. Compared to `maven-dependency-analyzer`, DEPCLEAN adds the unique features of detecting transitive and inherited bloated dependencies, and to produce a debloated version of the POM file. DEPCLEAN is open-source and reusable from Maven Central, the source code is available at https://github.com/castor-software/depclean.

## 4 Experimental Methodology

In this section, we present the research questions that articulate our study. We also describe the experimental protocols used to select and analyze Maven artifacts for an assessment of the impact of bloated dependencies in this ecosystem.

### 4.1 Research Questions

Our investigation of bloated dependencies in the Maven ecosystem is composed of four research questions grouped in two parts. In the first part, we perform a large scale quantitative study to answer the following research questions:

- **RQ1**: *How frequently do bloated dependencies occur?* With this research question, we aim at quantifying the amount of bloated dependencies among 9,639 Maven artifacts. We measure direct, inherited and transitive dependencies to provide an in-depth assessment of the dependency bloat in the Maven ecosystem.
- **RQ2**: *How do the reuse practices affect bloated dependencies?* In this research question, we analyze bloated dependencies with respect to two distinctive aspects of reuse in the Maven ecosystem: the additional complexity of the Maven dependency tree caused by transitive dependencies, and the choice of a multi-module architecture.

The second part of our study focuses on 30 notable Maven projects and presents the qualitative feedback about how developers react to bloated dependencies, and to the solutions provided by DEPCLEAN. It is guided by the following research questions:

- **RQ3**: *To what extent are developers willing to remove bloated-direct dependencies?* Direct dependencies are those that are explicitly declared in the POM. Hence, those dependencies are easy to remove since it only requires the modification of a POM that developers can easily change. In this research question, we use DEPCLEAN to detect and fix bloated-direct dependencies. Then, we communicate the results to the developers. We report on their feedback.
- **RQ4**: *To what extent are developers willing to exclude bloated-transitive dependencies?* Transitive dependencies are those not explicitly declared in the POM but induced from other dependencies. We exchange with developers about such cases. This gives unique insights about how developers react to excluding transitive dependencies from their projects.

## 4.2 Experimental Protocols

### 4.2.1 Protocol of the Quantitative Study (RQ1 & RQ2)

Figure 4 shows our process to build a dataset of Maven artifacts in order to answer RQ1 and RQ2. Steps ① and ② focus on the collection of a representative set, according to the number of direct dependencies, of Maven artifacts: we sample our study subjects from the whole MDG, then we resolve the dependencies of each study subject. In steps ③ and ④, we analyze dependency usages with DEPCLEAN and compute the set of metrics to answer RQ1 and RQ2.

**Filter Artifacts** In the first step, we leverage the Maven Dependency Graph (MDG) from previous research (Benelallam et al. 2019), a graph database that captures the complete dependency relationships between artifacts in Maven Central at a given point in time. Figure 5a shows the distribution of the number of direct dependencies of the artifacts with at least one direct dependency in the MDG. The number of direct dependencies is a representative measure that reflects the initial intentions of developers with respect to code reuse. We select a representative sample that includes 14,699 Maven artifacts (Fig. 5b). Representativeness is achieved by sampling over the probability distribution of the number of direct dependencies per artifact in the MDG, per the recommendation of Shull (2007, Chapter 8.3.1). From the sampled artifacts, we select as study subjects all the artifacts that meet the following additional criteria:

- Public API: The subjects must contain at least one `.class` file with one or more public methods, i.e., can be reused via external calls to their API.
- Diverse: The subjects all have different *groupId* and *artifactId*, i.e., they belong to different Maven projects.

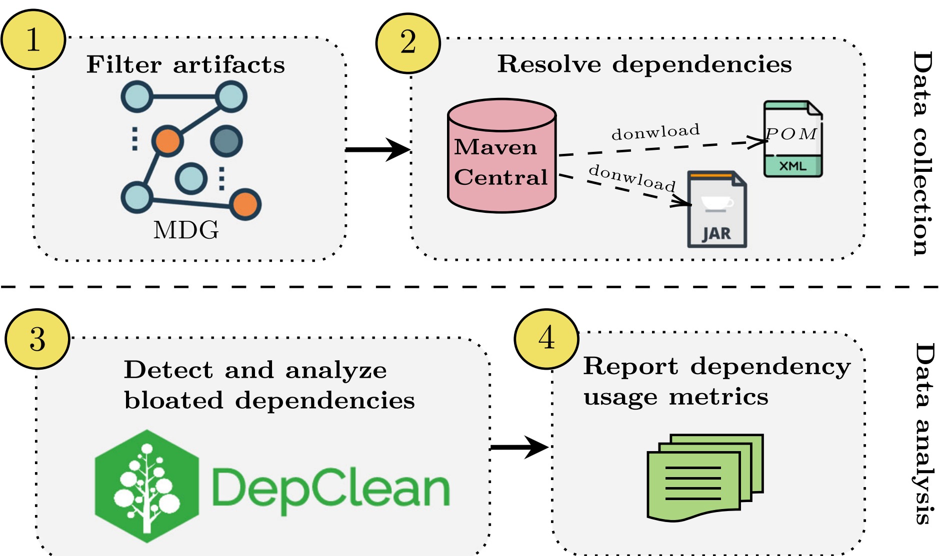



**Fig. 4** Experimental framework used to collect artifacts and analyze bloated dependencies in the Maven ecosystem

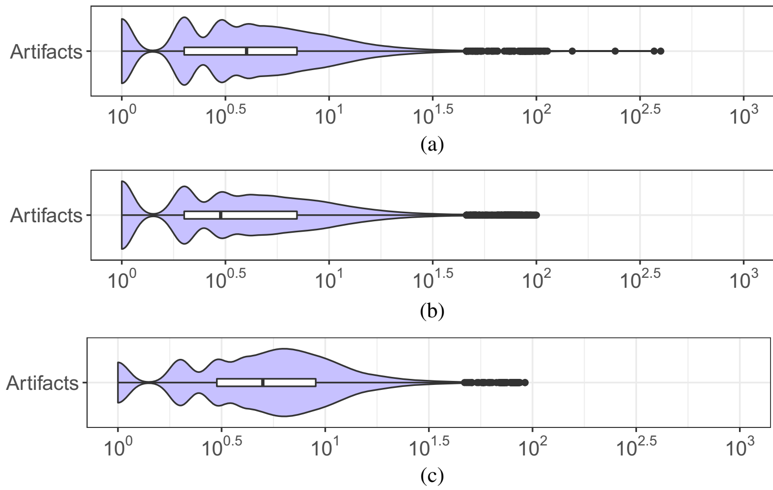



**Fig. 5** Distribution of the number of direct dependencies of the artifacts at the different stages of the data filtering process

- Reused: The subjects are used by at least one client via direct declaration.
- Complex: The subjects have at least one direct dependency with *compile* scope, i.e., we can analyze the dependency tree and the reused artifacts.
- Latest: The subjects are the latest released version at the time of the experiment (October, 2019).

After this systematic selection procedure, we obtain a dataset of 9,770 Maven artifacts. The density of artifacts with a number of direct dependencies in the range [3, 9] in our dataset (Fig. 5c) is higher than in the MDG (Fig. 5a). This is a direct consequence of our selection criteria where artifacts must have at least one direct dependency with *compile* scope. This filter removes artifacts that contain only dependencies that are not shipped in the JAR of the artifact (e.g., *test* dependencies). Therefore, the 9,770 artifacts used as study subjects are representative of the artifacts in Maven Central that include third-party dependencies in the JAR.

**Resolve Dependencies** In the second step, we download the binaries of all the selected artifacts and their POMs from Maven Central and we resolve all their direct and transitive dependencies to a local repository. To ensure the consistency of our analysis, we discard the artifacts that depend on libraries hosted in external repositories. In case of any other error when downloading some dependency, we exclude the artifact from our analysis. This occurred for a total of 131 artifacts in the dataset obtained in the first step.

Table 2 shows the descriptive statistics about the 9,639 artifacts in our final dataset for RQ1 and RQ2. The dataset includes 44,488 direct, 180,693 inherited, and 498,263 transitive dependency relationships (723,444 in total). We report about their dependencies with *compile* scope, since those dependencies are necessary to build the artifacts. Columns #C, #M, and #F give the distribution of the number of classes, methods, and fields per artifact (we count both the public and private API members). The size of artifacts varies, from small artifacts with one single class (e.g., `org.elasticsearch.client:transport:6.2.4`), to large libraries with thousands of classes (e.g., `org.apache.hive:hive-exec:3.1.0`). In total, we analyze

**Table 2** Descriptive statistics of the 9,639 Maven artifacts selected to conduct our quantitative study of bloated dependencies (RQ1 & RQ2)

| | API Members | | | Dependencies | | |
|---|---|---|---|---|---|---|
| | #C | #M | #F | #D | #I | #T |
| Min. | 1 | 1 | 0 | 1 | 0 | 0 |
| 1st-Q | 10 | 63 | 21 | 2 | 0 | 6 |
| Median | 32 | 231 | 75 | 4 | 2 | 20 |
| 3rd-Q | 111 | 891 | 279 | 7 | 18 | 59 |
| Max. | 47,241 | 435,695 | 129,441 | 148 | 453 | 1,776 |
| Total | 2,397,879 | 22,633,743 | 6,510,584 | 44,488 | 180,693 | 498,263 |

the bytecode of more than 30 millions of API members. Columns #D, #I, and #T account for the distributions of direct, inherited, and transitive dependencies, respectively. `com.bbossgroups.pdp:pdp-system:5.0.3.9` is the artifact with the largest number of declared dependencies in our dataset, with 148 dependency declarations in its POM file, while `be.atbash.json:octopus-accessors-smart:0.9.1` has the maximum number of transitive dependencies: 1,776. The distributions of direct and transitive dependencies are notably different: typically the number of transitive dependencies is an order of magnitude larger than direct dependencies, with means of 20 and 4, respectively.

**Dependency Usage Analysis** This is the first step to answer RQ1 and RQ2: collect the status of all dependency relationships for each artifact in our dataset. For each artifact, we first unpack its JAR file, as well as its dependencies. Then, for each JAR file, we analyze all the bytecode calls to API class members using DEPCLEAN. This provides a Dependency Usage Tree (DUT) for each artifact, on which each dependency relationship is labeled with one of the six categories as we illustrated in Table 1: bloated-direct (bd), bloated-inherited (bi), bloated-transitive (bt), used-direct (ud), used-inherited (ui), or used-transitive (ut).

**Collect Dependency Usage Metrics** This last step consists of collecting a set of metrics about the global presence of bloated dependencies. We define our analysis metrics with the goals of studying 1) the dependency usage relationships in the DUT (RQ1), and 2) the complexity resulting from the adoption of the multi-module Maven architecture (RQ2).

In RQ1, we analyze our dataset as a whole, looking at the usage status of dependency relationships from two perspectives:

– *Global distribution of dependency usage.* This is the normalized distribution of each category of dependency usage, over each dependency relationship of each of the 9,639 artifacts in our dataset.
– *Distribution of dependency usage type, per artifact.* For each of the six types of dependency usage, we compute the normalized ratio over the total number of dependency relationships for each artifact in our dataset.

In RQ2, we analyze how the specific reuse strategies of Maven relate to the presence of bloated dependencies. First, we use the number of transitive dependencies and the height of the dependency tree as a measure of complexity. The former measure is guided by the fact that transitive dependencies are more difficult to handle by developers; the latter measure is

guided by the idea that dependencies that are deeper in the dependency tree are more likely to be bloated. We use the following metrics:

– *Bloated dependencies w.r.t. the number of transitive dependencies.* For each artifact that has at least one transitive dependency, we determine the relation between the ratio of transitive dependencies and the ratio of bloated dependencies.
– *Bloated-transitive dependencies w.r.t. to the height of the dependency tree.* Given an artifact and its Maven dependency tree, the height of the tree is the longest path between the root and its leaves. To compute this metric, we group our artifacts according to the height of their tree. The maximum dependency tree height that we observed is 14. However, there are only 152 artifacts with a tree higher than 9. Therefore, we group all artifacts with height $\geq 9$. For each subset of artifact with the same height, we compute the size of the subset and the distribution of bloated-transitive dependencies of each artifact in the subset.

Second, we distinguish the presence of bloated dependencies between single and multi-module Maven projects, according to the following metrics:

– *Global distribution of dependency usage in a single or multi-module project.* We present two plots that measure the distribution of each type of dependency usage in the set of single and multi-module projects. It is to be noted that the plot for single-module projects does not include bloated-inherited (bi) and used-inherited (ui) dependencies since they have no inherited dependencies.
– *Distribution of dependency usage type, per artifact, in a single or multi-module project.* We present two plots that provide six distributions each: the distribution of each type of dependency usage type for artifacts that are in a single-module or multi-module project.

#### 4.2.2 Protocol of the Qualitative Study (RQ3 & RQ4)

In RQ3 and RQ4, we perform a qualitative assessment of the relevance of bloated dependencies for the developers of open-source projects. We systematically select 30 notable open-source projects hosted on GitHub to conduct this analysis. We query the GitHub API to list all the Java projects ordered by their number of stars. Then, we randomly select a set of projects that fulfil all the following criteria: (1) we can build them successfully with Maven, (2) the last commit was at the latest in October 2019, (3) they declare at least one dependency in the POM, (4) they have a description in the README about how to contribute through pull requests, and (5) they have more than 100 stars on GitHub.

Table 3 shows the selected 30 projects per those criteria, to which we submitted at least one pull request. They are listed in decreasing order according to their number of stars on GitHub. The first column shows the name of the project as declared on GitHub, followed by the name of the targeted module if the project is multi-module. Notice that in the case of `jenkins` we submitted two pull requests targeting two distinct modules: `core` and `cli`. Columns two to four describe the projects according to its category as assigned to the corresponding released artifact in Maven Central, the number of commits in the master branch in October 2019, and the number of stars at the moment of conducting this study. Columns five to seven report about the total number of direct, inherited, and transitive dependencies included in the dependency tree of each considered project.

**Table 3** Maven projects selected to conduct our qualitative study of bloated dependencies (RQ3 & RQ4)

| Project | Description | | | Dependencies | | |
|---|---|---|---|---|---|---|
| | Category | #Commits | #Stars | #D | #I | #T |
| `jenkins [core]` | Automation Server | 29,040 | 14,578 | 51 | 2 | 87 |
| `jenkins [cli]` | Automation Server | 29,040 | 14,578 | 17 | 2 | 0 |
| `mybatis-3 [mybatis]` | Relational Mapping | 3,145 | 12,196 | 23 | 0 | 51 |
| `flink [core]` | Streaming | 19,789 | 11,260 | 14 | 10 | 34 |
| `checkstyle [checkstyle]` | Code Analysis | 8,897 | 8,897 | 18 | 0 | 36 |
| `auto [common]` | Meta-programming | 1,081 | 8,331 | 8 | 0 | 24 |
| `neo4j [collections]` | Graph Database | 66,602 | 7,069 | 8 | 2 | 21 |
| `CoreNLP` | NLP | 15,544 | 6,812 | 23 | 0 | 45 |
| `moshi [moshi-kotlin]` | JSON Library | 793 | 5,731 | 14 | 0 | 21 |
| `async-http-client [http-client]` | HTTP Client | 4,034 | 5,233 | 29 | 16 | 130 |
| `error-prone [core]` | Defects Detection | 4,015 | 4,915 | 44 | 0 | 35 |
| `alluxio [core-transport]` | Database | 30,544 | 4,442 | 6 | 14 | 73 |
| `javaparser [symbol-solver-logic]` | Code Analysis | 6,110 | 2,784 | 3 | 0 | 8 |
| `undertow [benchmarks]` | Web Server | 4,687 | 2,538 | 10 | 0 | 19 |
| `wc-capture [driver-openimaj]` | Webcam | 629 | 1,618 | 3 | 0 | 84 |
| `teavm [core]` | Compiler | 2,334 | 1,354 | 9 | 0 | 9 |
| `handlebars [markdown]` | Templates | 916 | 1,102 | 6 | 0 | 13 |
| `jooby [jooby]` | Web Framework | 2,462 | 1,083 | 23 | 0 | 68 |
| `tika [parsers]` | Parsing library | 4,650 | 929 | 81 | 0 | 67 |
| `orika [eclipse-tools]` | Object Mapping | 970 | 864 | 3 | 0 | 3 |
| `spoon [core]` | Meta-programming | 2,971 | 840 | 16 | 2 | 59 |
| `accumulo [core]` | Database | 10,314 | 763 | 26 | 1 | 51 |
| `couchdb-lucene` | Text Search | 1,121 | 752 | 25 | 0 | 112 |
| `jHiccup` | Profiling | 215 | 519 | 4 | 0 | 1 |
| `subzero [server]` | Cryptocurrency | 158 | 499 | 6 | 0 | 100 |
| `vulnerability-tool [shared]` | Security | 1,051 | 324 | 6 | 4 | 2 |
| `para [core]` | Cloud Framework | 1,270 | 310 | 47 | 2 | 112 |
| `launch4j-maven-plugin` | Deployment Tool | 316 | 194 | 7 | 0 | 61 |
| `jacop` | CP Solver | 1,158 | 155 | 7 | 0 | 9 |
| `selenese-runner-java` | Interpreter | 1,688 | 117 | 23 | 0 | 148 |
| `commons-configuration` | Config library | 3,159 | 100 | 31 | 0 | 49 |

We answer RQ3 according to the following protocol: 1) we run DEPCLEAN, we build the artifact with the debloated POM file, 2) if the project builds successfully, we analyze the project to propose a relevant change to the developers per the contribution guidelines, 3) we propose a change in the POM file in the form of a pull request, and 4) we discuss the pull

```
87  -         <dependency>
88  -             <groupId>org.apache.httpcomponents</groupId>
89  -             <artifactId>httpmime</artifactId>
90  -         </dependency>
```

**Fig. 6** Example of commit removing the bloated-direct dependency `org.apache.httpcomponents:httpmime` in the project Undertow

request through GitHub. Figure 6 shows an excerpt of the diff of such a change in the POM file. We note that the submitted pull requests contain a small modification in a single file: the POM.

In the first step of the protocol, we use DEPCLEAN to obtain a report about the usage of dependencies. We analyze dependencies with both compile and test scope. Once a bloated-direct dependency is found, we remove it directly in the POM and proceed to build the project. If the project builds successfully after the removal (all the tests pass), we submit the pull request with the corresponding change. If after the removal of the dependency the build fails, then we consider the dependency as used dynamically and do not suggest removing it. In the case of multi-module projects, with bloated dependencies in several modules, we submitted a single pull request per module.

For each pull request, we analyze the Git history of the POM file to determine when the bloated dependency was declared or modified. Our objective is to collect information in order to understand how the dependencies of the projects change during their evolution. This allows us to prepare a more informative pull request message and to support our discussion with developers. We also report on the benefits of tackling these bloated dependencies by describing the differences between the original and the debloated packaged artifact of the project in terms of the size of the bundle and the complexity of its dependency tree, when the difference was significant. Each pull request includes an explanatory message. Figure 7 shows an example of the pull request message submitted to the project Undertow.[5] The message explains the motivations of the proposed change, as well as the negative impact of keeping these bloated dependencies in the project.

To answer RQ4, we follow the same pull request submission protocol as for RQ3. We use DEPCLEAN to detect bloated-transitive dependencies and submit pull requests suggesting the addition of the corresponding exclusion clauses in each project POM. Figure 8 shows an example of a pull request message submitted to the project Apache Accumulo[6], while Fig. 9 shows an excerpt of the commit proposing the exclusion of the transitive dependency `org.apache.httpcomponents:httpcore` from the direct dependency `org.apache.thrift:libthrift` in its POM.

Additional information related to the selected projects and the research methodology employed is publicly available as part of our replication package at https://github.com/castor-software/depclean-experiments.

[5] https://github.com/undertow-io/undertow

[6] https://github.com/apache/accumulo

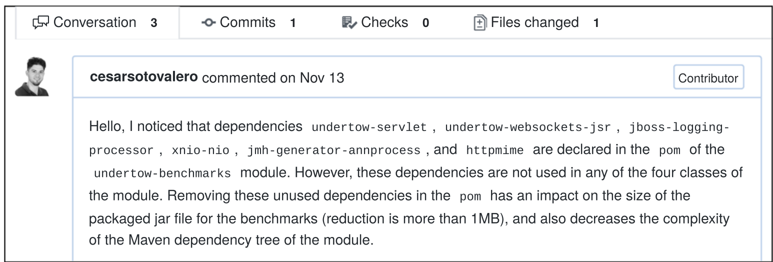



**Fig. 7** Example of message of a pull request sent to the project Undertow on GitHub

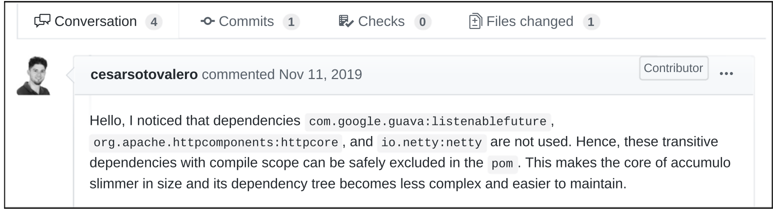



**Fig. 8** Example of message in a pull request sent to the project Apache Accumulo on GitHub

```
104         <dependency>
105           <groupId>org.apache.thrift</groupId>
106           <artifactId>libthrift</artifactId>
107 +         <exclusions>
108 +           <exclusion>
109 +             <groupId>org.apache.httpcomponents</groupId>
110 +             <artifactId>httpcore</artifactId>
111 +           </exclusion>
112 +         </exclusions>
```

**Fig. 9** Example of commit excluding the bloated-transitive dependency `org.apache.httpcomponents:httpcore` in the project Apache Accumulo

## 5 Experimental Results

We now present the results of our in-depth analysis of bloated dependencies in the Maven ecosystem.

### 5.1 RQ1: How Frequently do Bloated Dependencies Occur?

In this first research question, we investigate the status of all the dependency relationships of the 9,639 Maven artifacts under study.

Figure 10 shows the overall status of the 723,444 dependency relationships in our dataset. The x-axis represents the percentages, per usage type, of all the dependencies considered in the studied artifacts. The first observation is that the bloat phenomenon is massive: 543,610 (75.1%) of all dependencies are bloated, they are not needed to compile and run the code. This bloat is divided into three separate categories: 19,673 (2.7%) are bloated-direct dependency relationships (explicitly declared in the POMs); 111,649 (15.4%) are bloated-inherited dependency relationships from parent module(s); and 412,288 (57%) are bloated-transitive dependencies. Figure 10 shows that 75.1% of the relationships (edges in the dependency usage tree) are bloated dependencies. Note that this observation does not mean that 543,610 artifacts are unnecessary and can be removed from Maven Central. The same artifact can be present in several DUTs, i.e., reused by different artifacts, but be part of a bloated dependency relationship only in some of these DUTs, and part of a used relationship in the other DUTs.

Figure 11 shows the overall status of the dependencies with respect to the type of the dependency relationship (direct, inherited, and transitive). We observe that approximately 1/3 of direct dependencies are bloated (34.23%), whereas inherited and transitive dependencies have a higher percentage of bloat (61.79% and 82.5% of bloat, respectively). These results indicate that artifacts with inherited and transitive dependencies are more likely to have more bloated dependencies. They also confirm that transitive dependencies are the most susceptible to bloat.

In the following, we illustrate the three types of bloated dependency relationships with concrete examples.

**Bloated-Direct** We found that 2.7% of the dependencies declared in the POM file of the studied artifacts are not used at all via bytecode calls. Recall that detecting this type of bloated dependencies is good, because they are easy to remove

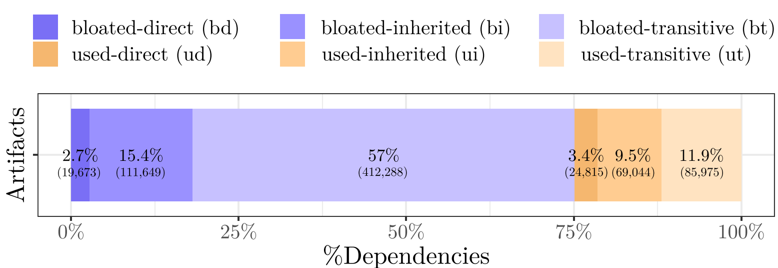



**Fig. 10** Ratio per usage status of the 723,444 dependency relationships analyzed. Raw counts are inside parentheses below each percentage

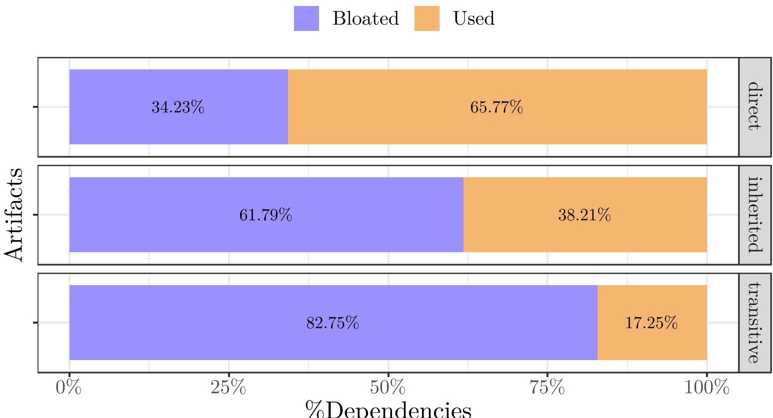



**Fig. 11** Ratio per dependency type of bloated and used dependencies of the 723,444 dependency relationships analyzed

by developers with a single change in the POM file of the project under consideration. As an example, the Apache Ignite[7] project has deployed an artifact: `org.apache.ignite:ignite-zookeeper:2.4.0`, which contains only one class in its bytecode: `TcpDiscoveryZookeeperIpFinder`, and it declares a direct dependency in the POM towards `slf4j`, a widely used Java logging library. However, if we analyze the bytecode of `ignite-zookeeper`, no call to any API member of `sl4j` exists, and therefore, it is a bloated-direct dependency. After a manual inspection of the commit history of the POM, we found that `sl4j` was extensively used across all the modules of Apache Ignite at the early stages of the project, but it was later replaced by a dedicated logger, and its declaration remained intact in the POM.

**Bloated-Inherited** In our dataset, a total of 4,963 artifacts are part of multi-module Maven projects. Each of these artifacts declares a set of dependencies in its POM file, and also inherits a set of dependencies from a parent POM. DEPCLEAN marks those inherited dependencies are either bloated-inherited or used-inherited. Our dataset includes a total of 111,649 dependency relationships labeled as bloated-inherited, which represents 15.4% of all dependencies under study and 61.8% of the total of inherited dependencies. For example, the artifact `org.apache.drill:drill-protocol:1.14.0` inherits dependencies `commons-codec` and `commons-io` from its parent POM `org.apache.drill:drill-root:1.14.0`, however, those dependencies are not used in this module, and therefore they are bloated-inherited dependencies.

**Bloated-Transitive** In our dataset, bloated-transitive dependencies represent the majority of the bloated dependency relationships: 412,288 (57%). This type of bloat is a natural consequence of the Maven dependency resolution mechanism, which automatically resolves all the dependencies whether they are explicitly declared in the POM file of the project or not. Transitive dependencies are the most common type of dependency relationships, having a direct impact on the growth of the dependency trees. This type of bloat is the most common in the Maven ecosystem. For example, the artifact

[7] https://github.com/apache/ignite

```
package org.apache.streams.filters;
import com.google.common.base.Preconditions;
public class VerbDefinitionDropFilter implements
    StreamsProcessor {
  ...
  @Override
  public List<StreamsDatum> process(StreamsDatum entry) {
    ...
    Preconditions.checkArgument(entry.getDocument() instanceof
        Activity);
    return result;
  }
  ...
}
```

**Listing 2** Code snippet of the class `VerbDefinitionDropFilter` present in the artifact `org.apache.streams:streams-filters:0.6.0`. The library `com.google.guava:guava:20.0` is included in its classpath via transitive dependency and called from the source code, but no dependency towards `guava` is declared in its POM

`org.eclipse.milo:sdk-client:0.2.1` ships the transitive dependency `gson` in its MDT, induced from its direct dependency towards `bsd-parser-core`. However, the part of `bsd-parser-core` used by `sdk-client` does not call any API member of `gson`, and therefore it is a bloated-transitive dependency.

In the following, we discuss the dependencies that are actually used. We observe that direct dependencies represent only 3.4% of the total of dependencies in our dataset. This means that the majority of the dependencies that are necessary to build Maven artifacts are not declared explicitly in the POM files of these artifacts.

It is interesting to note that 85,975 of the dependencies used by the artifacts under study are transitive dependencies. This kind of dependency usage occurs in two different scenarios: (1) the artifact uses API members of some transitive dependencies, without declaring them in its own POM file; or (2) the transitive dependency is necessary to provide a functionality to another, actually used dependency, in the dependency tree of the artifact.

```
package org.duracloud.audit.task;
import org.duracloud.common.json.JaxbJsonSerializer;
public class AuditTask extends TypedTask {
  ...
  private static JaxbJsonSerializer<Map<String, String>>
      getPropsSerializer() {
    return new JaxbJsonSerializer<>((Class<Map<String,
        String>>) (Object) new HashMap<String,
        String>().getClass());
  }
  ...
}
```

**Listing 3** Code snippet of the class `AuditTask` present in the artifact `org.duracloud:auditor:4.4.3`. The library `org.codehaus.jackson:jackson-mapper-asl:1.6.2` is used indirectly through the direct dependency `org.duracloud:common-json:4.4.3`

We now discuss an example of the first scenario based on the `org.apache.streams:streams-filters:0.6.0` artifact from the Apache

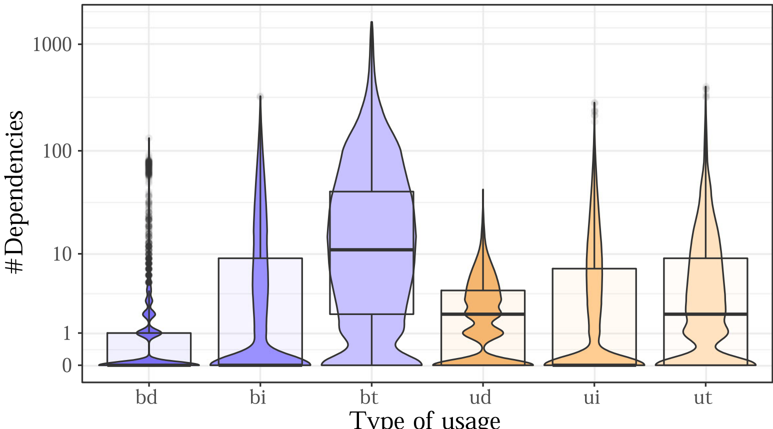



**Fig. 12** Distributions of the six types of dependency usage relationships for the studied artifacts. The thicker areas on each curve represent concentrations of artifacts per type of usage

Streams[8] project. It contains two classes: `VerbDefinitionDropFilter` and `VerbDefinitionKeepFilter`. Listing 2 shows part of the source code of the class `VerbDefinitionDropFilter`, which imports the class `PreCondition` from library `guava` (line 2) and uses its static method `checkArgument` in line 8 of method `process`. However, if we inspect the POM of `streams-filters`, we notice that there is no dependency declaration towards `guava`. It declares a dependency towards `streams-core`, which in turn depends on the `streams-utils` artifact that has a direct dependency towards `guava`. Hence, `guava` is a used-transitive dependency of `streams-filters`, called from its source code.

Let us now present an example of the second scenario. Listing 3 shows an excerpt of the class `AuditTask` included in the artifact `org.duracloud:auditor:4.4.3`, from the project DuraCloud.[9] In line 6, the method `getPropsSerializer` instantiates the `JaxbJsonSerializer` object that belongs to the direct dependency `org.duracloud:common-json:4.4.3`. This object, in turn, creates an `ObjectMapper` from the transitive dependency `jackson-mapper-asl`. Hence, `jackson-mapper-asl` is a necessary, transitive provider for `org.duracloud:auditor:4.4.3`.

Figure 12 shows the distributions of dependency usage types per artifact. The figure presents superimposed log-scaled box-plots and violin-plots of the number of dependency relationships corresponding to the six usage types studied. Box-plots indicate the standard statistics of the distribution (i.e., lower/upper inter-quartile range, max/min values, and outliers), while violin plots indicate the entire distribution of the data.

We observe that the distributions of the bloated-direct (bd) and bloated-transitive (bt) dependencies vary greatly. Bloated-direct dependencies are distributed between 0 and 1 (1st-Q and 3rd-Q), with a median of 0; whereas the second ranges between 2 and 41 (1st-Q and 3rd-Q), with a median of 11. These values are in line with the statistics presented in

[8]https://streams.apache.org

[9]https://duraspace.org

Table 2, since the number of direct and transitive dependencies in general differ approximately by one order of magnitude. Overall, from the 9,639 Maven artifacts studied, 3,472 (36%) have at least one bloated-direct dependency, while 8,305 (86.2%) have at least one bloated-transitive.

On the other hand, the inter-quartile range of bloated-direct (bd) dependencies is more compact than the used-direct (ud). In other words, the dependencies declared in the POM are mostly used. This result is expected, since developers have more control over the edition (adding/removing dependencies) of the POM file of their artifact.

The median number of used-transitive (ut) dependencies is significantly lower than the median number of bloated-transitive (bt) dependencies (2, vs. 11). This suggests that the default dependency resolution mechanism of Maven is suboptimal with respect to ensuring minimal dependency inclusion.

The number of outliers in the box-plots differs for each usage type. Notably, the bloated-direct dependencies have more outliers (in total, 25 artifacts have more than 100 bloated-direct dependencies). In particular, the artifact `com.bbossgroups.pdp:pdp-system:5.0.3.9` has the maximum number of bloated-direct dependencies: 133, out of the 147 declared in its POM. The total number of artifacts with at least one bloated-direct dependency in our dataset is 2,298, which represents 23.8% of the 9,639 studied artifacts.

**Answer to RQ1:** The analysis of the 723,444 analysed dependency relationships in our dataset reveals that 543, 610 (75%) of them are bloated. Most of the bloated dependencies are transitive 412; 288 (57%). Overall, 36% of the artifacts have at least one bloated dependency that is declared in their POM file. To our knowledge, this is the first scientific observation of this phenomenon.
**Implications:** Since developers have more control over direct dependencies, up to 17, 673 (2:7%) of dependencies can be removed directly from the POM of Maven artifacts, in order to obtain smaller binaries and reduced attack surface. RQ3 will explore the willingness of developers to do so.

## 5.2 RQ2: How do the Reuse Practices Affect Bloated Dependencies?

In this research question, we investigate how the reuse practices that lead to these distinct types of dependency relationships are related to the bloated dependencies that emerge in Maven artifacts.

Figure 13 shows the distributions, in percentages, of the direct, inherited, and transitive dependencies for the 9,639 studied artifacts. The artifacts are sorted, from left to right, in increasing order according to their ratio of direct dependencies. The y-axis indicates the ratio of each type of dependency for a given artifact. First, we observe that 4,967 artifacts belong to multi-module projects. Among these artifacts, the extreme case (far left of the plot) is `org.janusgraph:janusgraph-berkeleyje:0.4.0`, which declares only 1.4% of its dependencies in its POM, while the 48.6% of its dependencies are inherited from parent POM files, and 50% are transitive. Second, we observe that the ratio of transitive dependencies is not equally distributed. On the right side of the plot, 879 (9.1%) artifacts have no transitive dependency (they have 100% direct dependencies). Meanwhile, 5,561 (57.7%) artifacts have more than 50% transitive dependencies. The extreme

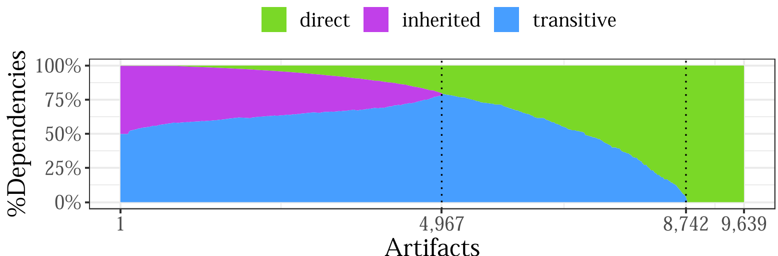



**Fig. 13** Distribution of the percentages of direct, inherited, and transitive dependencies for the 9,639 artifacts considered in this study

case is `org.apereo.cas:cas-server-core-api-validation:6.1.0`, with 77.6% transitive dependencies.

In summary, the plot in Fig. 13 offers a big picture of the distribution of the three types of dependency usage in our dataset. The inherited and transitive dependencies are a significant phenomenon in Maven: 8,742 (90.7%) artifacts in our dataset have transitive dependencies, and 51.5% of artifacts belong to multi-module projects. This observation confirms the results of the previous section, most of the bloated dependencies in our dataset are either transitive (57%) or inherited (15.4%).

### 5.2.1 Transitive Dependencies

Figure 14 plots the relation between the ratio of transitive dependencies and the ratio of bloated dependencies. Each dot represents an artifact. Dots have a higher opacity in the presence of overlaps.

The key insight in Fig. 14 is that the larger concentration of artifacts is skewed to the top right corner, indicating that artifacts with a high percentage of transitive dependencies also tend to exhibit higher percentages of bloated dependencies. Indeed, both variables are positively correlated, according to the Spearman's rank correlation test ($\rho$ = 0.65, p-value $< 0.01$).

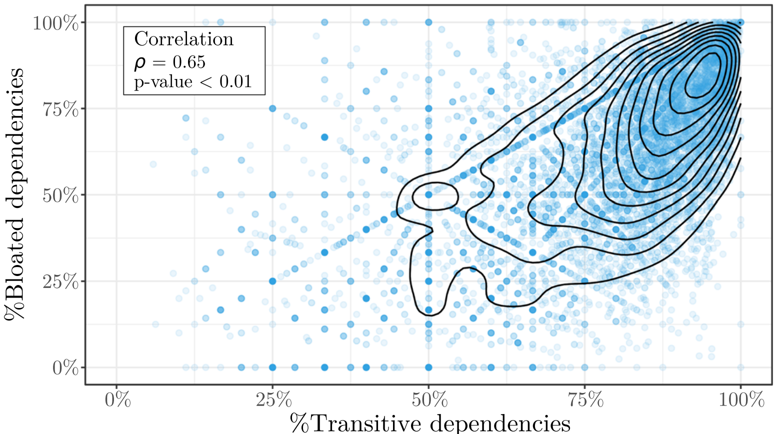



**Fig. 14** Relation between the percentages of transitive dependencies and the percentage of bloated dependencies in the 9,639 studied artifacts

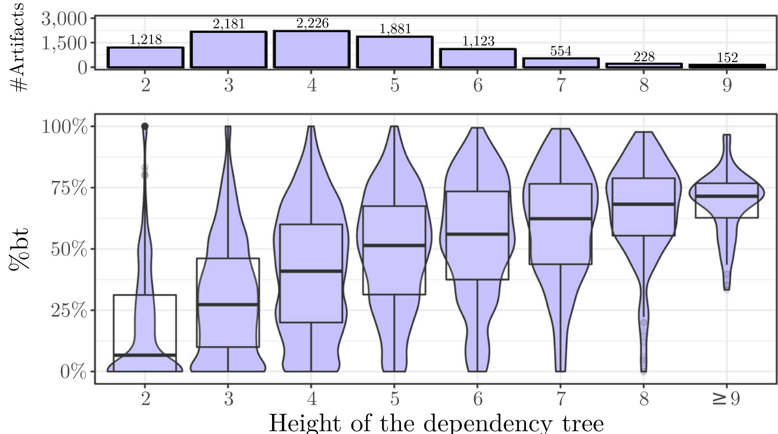



**Fig. 15** Distribution of the percentages of bloated-transitive dependencies for our study subjects with respect to the height of the dependency trees. Height values greater than 10 are aggregated. The bar plot at the top represents the number of study subjects for each height

Figure 15 shows the distribution of the ratio of transitive bloated dependencies according to the height of the dependency tree. The artifact in our dataset with the largest height is `top.wboost:common-base-spring-boot-starter:3.0.RELEASE`, with a height of 14. The bar plot on top of Fig. 15 indicates the number of artifacts that have the same height. We observe that most of the artifacts have a height of 4: 2,226 artifacts in total. Considering the number of dependencies, this suggests that the dependency trees tend to be wider than deep. This is direct consequence of the automatic dependency management by Maven: any dependency that already appears at a level closer to the root will be omitted by Maven if it is referred to at a deeper level.

Looking at the 58 artifacts with height $\geq$ 9, we notice that most of them belong to multi-module projects, and declare other modules in the same project as their direct dependencies. This is a regular practice of multi-module projects, which allows to release each module as an independent artifact. Meanwhile, this increases the complexity of dependency trees. For example, artifact `org.wso2.carbon.devicemgt:org.wso2.carbon.apimgt.handlers:3.0.192` is the extreme case of this practice in our dataset, with two direct dependencies towards other modules of the same project that in turn depend on other modules of this project. As a result, this artifact has 342 bloated-transitive and 87 bloated-inherited dependencies, a dependency tree of height 11, and is part of a multi-module project with a total of 79 modules released in Maven Central.

The plot in Fig. 15 shows a clear increasing trend of bloated-transitive dependencies as the height of the dependency tree increases. Indeed, both variables are positively correlated, according to the Spearman's rank correlation test ($\rho$ = 0.54, p-value ¡ 0.01). For artifacts with a dependency tree of height greater than 9, at least 28% of their transitive dependencies are bloated, while the median of the percentages of bloated-transitive dependencies for artifacts with height larger than 5 is more than 50%.

This finding confirms and complements the results of Fig. 14, showing that the height of the dependency tree is directly related to the occurrence of bloat. However, the height of the tree may not be the only factor that causes the bloat. For example, we hypothesize that number of transitive dependencies is another essential factor.

In order to validate this hypothesis, we perform a Spearman's rank correlation test between the number of bloated-transitive dependencies and the size of the dependency tree, i.e., the number of nodes in each tree. We found that there is a significant positive correlation between both variables ($\rho$ =0.67, p-value $< 0.01$). This confirms that the actual usage of transitive dependencies decreases with the increasing complexity of the dependency tree. This result is aligned with our previous study that suggest that most of the public API members of transitive dependencies are not used by its clients (Harrand et al. 2019).

In summary, our results point to the excess of transitive dependencies as one of the fundamental causes of the existence of bloated dependencies in the Maven ecosystem.

#### 5.2.2 Single-Module vs. Multi-Module

Let us investigate on the differences between single and multi-module architectures with respect to the presence of bloated dependencies. Figure 16 compares the distributions of bloated and used dependencies between multi-module and single-module artifacts in our dataset. We notice that, in general, multi-module artifacts have slightly more bloat than single-module, precisely 3.1% more (the percentage of bloat in single-module is 5.8% + 67.3% = 73.1% vs. 0.9% + 24.2% + 51.1% = 76.2% in multi-module). More interestingly, we observe that a majority of the inherited dependencies are bloated: 24.2% of the dependencies among multi-module project are bloated-inherited (bi), while only 15% are used-inherited (ui). This suggests that most of the dependencies inherited by Maven artifacts that belong to multi-module artifacts are not used by these modules.

We observe that the percentage of bloated-direct dependencies in multi-module artifacts is very small (0.9%) in comparison with single-module (5.8%). Meanwhile, the percentage of bloated-transitive dependencies in single-module (67.3%) is larger than in multi-module (51.1%). This is due to the "shift" of a part of direct and transitive dependencies into inherited dependencies when using a parent POM. Indeed, the "shift" from direct to inherited is the main motivation for having a parent POM: to have one single declaration of dependencies for many artifacts instead of letting each artifact manage their own dependencies.

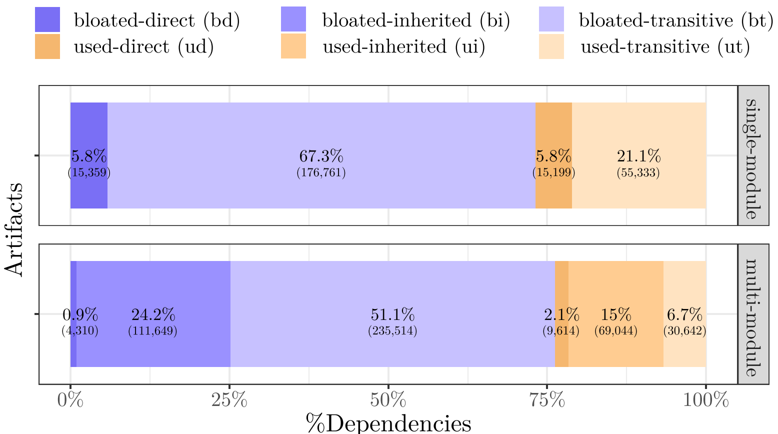



**Fig. 16** Comparison between multi-module and single-module artifacts according to the percentage status of their dependency relationships. Raw counts are inside parentheses below each percentage

This "shift" in the nature of dependencies between single and multi-module artifacts is further emphasized in Fig. 17. This plot shows superimposed log scaled box-plots and violin-plots comparing the distributions of the number of distinct dependency usage types per artifact, for single-module (top part of the figure) and multi-module (bottom part).

We observe that multi-module artifacts have less bloated-direct (1st-Q = 0, median = 0, 3rd-Q = 0) and less bloated-transitive (1st-Q = 2, median = 9, 3rd-Q = 40), compared to single-modules, as shown in Fig. 17. However, multi-module artifacts have a considerably larger number of bloated-inherited dependencies instead (1st-Q = 1, median = 5, 3rd-Q = 20). The extreme case in our dataset is the artifact `co.cask.cdap:cdap-standalone:4.3.4`, with 326 bloated-inherited dependencies in total.

In summary, the multi-module architecture in Maven projects contributes to limit redundant dependencies and facilitates the consistent versioning of dependencies in large projects. However, it introduces two challenges for developers. First, it leads to the emergence of bloated-inherited dependencies because of the friction of maintaining a common parent POM file: it is more difficult to remove dependencies from a parent POM than from an artifact's own POM. Second, it is more difficult for developers to be aware of and understand the dependencies that are inherited from the parent POM. This calls for better tooling and user interfaces to help developer grasp and analyze the inherited dependencies in multi-module projects, in order to detect bloated dependencies. To our knowledge, this type of tools is absent in the current Java dependency management ecosystem.

**Answer to RQ2:** Reuse practices that lead to complex dependency trees and multi-module architecture are correlated with the presence of bloated dependencies: the higher a dependency tree, the more bloated-transitive dependencies. In multi-module artifacts, part of the bloat associated with direct and transitive dependencies is shifted as bloated-inherited dependencies.
**Implications:** Developers should carefully consider reusing artifacts with several dependencies because they introduce bloat. They should also contemplate the risks of having bloated dependencies when considering adopting a multi-module architecture.

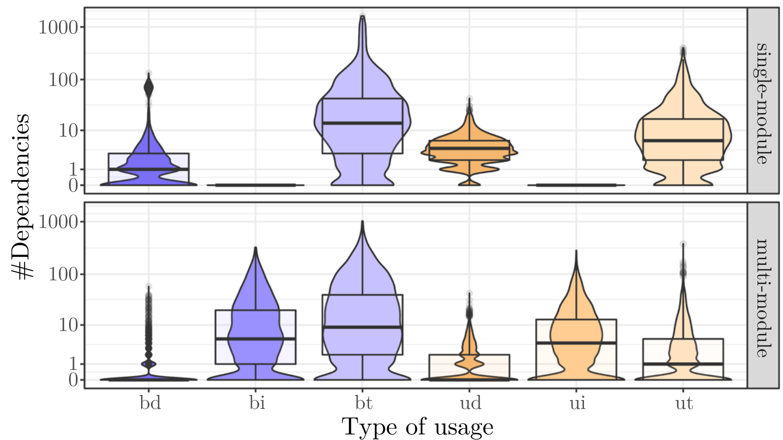



**Fig. 17** Comparison between multi-module and single-module projects according to their distributions of dependency usage relationships

### 5.3 RQ3: To what Extent are Developers Willing to Remove Bloated-Direct Dependencies?

In this research question, our goal is to see how developers react when made aware of bloated-direct dependencies in their projects. We do this by proposing the removal of bloated-direct dependencies to lead developers of mature open-source projects, as described in Section 4.2.2.

Table 4 shows the list of 19 pull requests submitted. Each pull request proposes the removal of at least one bloated-direct dependency in the POM. We received response from developers for 17 pull request. The first and second columns in the table show the name of the project and the pull request on GitHub. Columns three and four represent the number of bloated dependencies removed in the POM and the total number of dependencies removed from the dependency tree with the proposed change, including transitive ones. The last column shows the status of the pull request (✓ accepted, ✓* accepted with changes, ✗ rejected, or ✳ pending). The last row represent the acceptance rate calculated with respect to the projects with response, i.e., the total number of dependencies removed divided by the number of proposed removals. For example, for project `undertow` we proposed the removal of 6 bloated dependencies in its module `benchmarks`. As a result of this change, 17 transitive dependencies were removed from the dependency tree the module.

Overall, from the pull requests with responses from developers, 16/17 were accepted and merged. In total, 75 dependencies were removed from the dependency trees of the projects. This result demonstrates the relevance of handling bloated-direct dependencies for developers, and the practical usefulness of DEPCLEAN.

Let us now summarize the developer feedback. First, all developers agreed on the importance of refining the projects' POMs. This is reflected in the positive comments received. Second, their quick responses suggest that it is easy for them to understand the issues associated with the presence of bloated-direct dependencies in their projects. In 8/17 projects, the response time was less than 24 hours, which is an evidence that developers consider this type of improvement as a priority.

Our results also provide evidence of the fact that we, as external contributors to those projects, were able to identify the problem and propose a solution using DEPCLEAN. In the following, we discuss four cases of pull requests that are particularly interesting and the feedback provided by developers.

#### 5.3.1 Jenkins

DEPCLEAN detects that `jtidy` and `commons-codec` are bloated-direct dependencies present in the modules `core` and `cli` of `jenkins`. `jtidy` is an HTML syntax checker and pretty printer. `commons-codec` is an Apache library that provides an API to encode/decode from various formats such as Base64 and Hexadecimal.

Developers were reluctant to remove `jtidy` due to their concerns of affecting the users of `jenkins`, which could be potential consumers of this dependency. After further inspection, they found that the class `HTMLParser` of the `nis-notification-lamp-plugin`[10] project relies on `jtidy` transitively for performing HTML parsing.

[10]https://github.com/jenkinsci/nis-notification-lamp-plugin

**Table 4** List of pull requests proposing the removal of bloated-direct dependencies created for our experiments

| Project | Pull-request URL (https://github.com/) | Removed dependencies | | PR* |
|---|---|---|---|---|
| | | #D | Total | |
| jenkins [core, cli] | jenkinsci/jenkins/pull/4378 | 2 | 2 | ✓* |
| mybatis-3 [mybatis] | mybatis/mybatis-3/pull/1735 | 2 | 4 | ✓ |
| flink [core] | apache/flink/pull/10386 | 1 | 1 | ✓ |
| checkstyle | checkstyle/checkstyle/issues/7307 | 1 | 4 | ✗ |
| neo4j [collections] | neo4j/neo4j/pull/12339 | 1 | 2 | ✓ |
| CoreNLP | stanfordnlp/CoreNLP/pull/965 | 1 | 1 | ✓ |
| async-http-client [http-client] | AsyncHttpClient/async-http-client/pull/1675 | 1 | 12 | ✓ |
| error-prone [core] | google/error-prone/pull/1409 | 1 | 1 | ✓ |
| alluxio [core-transport] | Alluxio/alluxio/pull/10567 | 1 | 11 | ✓ |
| javaparser [symbol-solver-logic] | javaparser/javaparser/pull/2403 | 2 | 9 | ✓ |
| undertow [benchmarks] | undertow-io/undertow/pull/824 | 6 | 17 | ✓ |
| handlebars [markdown] | jknack/handlebars.java/pull/719 | 1 | 1 | ✳ |
| jooby | jooby-project/jooby/pull/1412 | 1 | 1 | ✓ |
| couchdb-lucene | rnewson/couchdb-lucene/pull/279 | 3 | 3 | ✳ |
| jHiccup | giltene/jHiccup/pull/42 | 1 | 1 | ✓ |
| subzero [server] | square/subzero/pull/122 | 1 | 4 | ✓ |
| vulnerability-tool [shared] | SAP/vulnerability-assessment-tool/pull/290 | 1 | 1 | ✓ |
| launch4j-maven-plugin | lukaszlenart/launch4j-maven-plugin/pull/113 | 2 | 4 | ✓ |
| jacop | radsz/jacop/pull/35 | 2 | 4 | ✓ |
| Acceptance rate | – | 25/26 | 75/79 | 16/17 |

*Status of the pull request: ✓ Accepted. ✓* Accepted with changes. ✗ Rejected. ✳ Pending

Developers also pointed out the fact that there is no classloader isolation in `jenkins`, and hence all dependencies in its `core` module automatically become part of its public API. A developer also referred to issues related to past experiences removing unused dependencies. He argued that external projects have depended on that inclusion and their builds were broken by such a removal. For example, the Git client plugin of `jenkins` mistakenly included Java classes from certain Apache authentication library. When they removed the dependency, some downstream consumers of the library were affected, and they had to include the dependency directly.

Consequently, we received the following feedback from an experienced developer of `jenkins`:

> We're not precluded from removing an unused dependency, but I think that the project values compatibility more than removal of unused dependencies, so we need to be careful that removal of an unused dependency does not cause a more severe problem than it solves.

After some discussions, developers agreed with the removal of `commons-codec` in module `cli`. Our pull request was edited by the developers and merged to the master branch one month after.

#### 5.3.2 Checkstyle

DEPCLEAN identifies the direct dependency `junit-jupiter-engine` as bloated. This is a test scope dependency that was added to the POM of `checkstyle` when migrating integration tests to JUnit 5. The inclusion of this dependency was necessary due to the deprecation of `junit-platform-surefire-provider` in the Surefire Maven plugin. However, the report of DEPCLEAN about this bloated-direct dependency was a false positive. The reason for this output occurs because `junit-jupiter-engine` is commonly used through reflective calls that cannot be captured at the bytecode level.

Althoughthis pull request was rejected, developers expressed interest in DEPCLEAN, which is encouraging. They also proposed a list of features for the improvement of our tool. For example, the addition of an exclusion list in the configuration of DEPCLEAN for dependencies that are known to be used dynamically, improvements on the readability of the generated report, and the possibility of causing the build process to fail in case of detecting the presence of any bloated dependency. We implemented each of the requested functionalities in DEPCLEAN. As a result, developers opened an issue to integrate DEPCLEAN in the Continuous Integration (CI) pipeline of `checkstyle`, in order to automatically manage their bloated dependencies.[11]

#### 5.3.3 Alluxio

DEPCLEAN detects that the direct dependency `grpc-netty`, declared in the module `alluxio-core-transport` is bloated. Figure 18 shows that this dependency also induces a total of 10 transitive dependencies that are not used (4 of them are omitted by Maven due to their duplication in the dependency tree). Developers accepted our pull request and also manifested their interest on using DEPCLEAN for managing unused dependencies in the future.

[11] https://github.com/checkstyle/checkstyle/issues/7307

```
+- io.grpc:grpc-netty:jar:1.17.1:compile
|  +- (io.grpc:grpc-core:jar:1.17.1:compile - omitted for duplicate)
|  +- io.netty:netty-codec-http2:jar:4.1.30.Final:compile
|  |  +- io.netty:netty-codec-http:jar:4.1.30.Final:compile
|  |  |  \- io.netty:netty-codec:jar:4.1.30.Final:compile
|  |  |     \- (io.netty:netty-transport:jar:4.1.30.Final:compile - omitted for duplicate)
|  |  \- io.netty:netty-handler:jar:4.1.30.Final:compile
|  |     +- io.netty:netty-buffer:jar:4.1.30.Final:compile
|  |     |  \- io.netty:netty-common:jar:4.1.30.Final:compile
|  |     +- (io.netty:netty-transport:jar:4.1.30.Final:compile - omitted for duplicate)
|  |     \- (io.netty:netty-codec:jar:4.1.30.Final:compile - omitted for duplicate)
```

**Fig. 18** Transitive dependencies induced by the bloated-direct dependency `grpc-netty` in the dependency tree of module `alluxio-core-transport`. The tree is obtained with the `dependency:tree` Maven goal

### 5.3.4 Undertow

DEPCLEAN detects a total of 6 bloated-direct dependencies in the `benchmarks` module of the project `undertow`: `undertow-servlet`, `undertow-websockets-jsr`, `jboss-logging-processor`, `xnio-nio`, `jmh-generator-annprocess`, and `httpmime`. In this case, we received a rapid positive response from the developers two days after the submission of the pull request. Removing the suggested bloated-direct dependencies has a significant impact on the size of the packaged JAR artifact of the `undertow-benchmarks` module. We compare the sizes of the bundled JAR before and after the removal of those dependencies: the binary size reduction represents more than 1MB. It is worth mentioning that this change also reduced the complexity of the dependency tree of the module.

**Summary of RQ3:** We used DEPCLEAN to propose 19 pull requests removing bloated-direct dependencies, from which 17/19 were answered. 16/17 pull requests with response were accepted and merged by open-source developers (In total, 75 dependencies were removed from the dependency tree of 16 projects).
**Implications:** Removing bloated-direct dependencies is relevant for developers and it is perceived as a valuable contribution. This type of change in the POM files are small, and they can have a significant impact on the dependency tree of Maven projects.

## 5.4 RQ4: To what Extent are Developers Willing to Exclude Bloated-Transitive Dependencies?

In this research question, our goal is to see how developers react when made aware of bloated-transitive dependencies. We do this by proposing the exclusion of bloated-transitive dependencies to them, as described in Section 4.2.2.

Table 5 shows the list of 13 pull requests submitted. Each pull request proposes the exclusion of at least one transitive dependency in the POM. We received response from developers for 9 pull requests. The first and second columns show the name of the project and the pull request on GitHub. Columns three and four represent the number of bloated-transitive dependencies explicitly excluded and the total number of dependencies removed in the dependency tree as resulting from the exclusion. The last column shows the status of the pull request (✓ accepted, ✗ rejected, or ✳ pending). The last row represents the acceptance rate with respect to the projects with response. For example, for the project `spoon` we propose the exclusion of four bloated-transitive dependencies in its `core` module. As a

result of this change, 31 transitive dependencies were removed from the dependency tree of this module.

Overall, from the pull requests with responses from developers, 5 were accepted and 4 were rejected. In total, 65 bloated dependencies were removed from the dependency trees of 5 projects. We notice that the accepted pull requests involve those projects for which the exclusion of transitive dependencies also represents the removal of a large number of other dependencies from the dependency tree. This result suggests that developers are more careful concerning this type of contribution.

As in RQ3, we obtained valuable feedback from developers about the pros and cons of excluding bloated-transitive dependencies. In the following, we provide unique qualitative insights about the most interesting cases and explain the feedback obtained from developers to the research community.

### 5.4.1 Jenkins

DEPCLEAN detects the bloated-transitive dependencies `constant-pool-scanner` and `eddsa` in the module `core` of `jenkins`. These bloated dependencies were induced through the direct dependencies `remoting` and `cli`, respectively. In the message of the pull request, we explain how their exclusion contributes to make the `core` of `jenkins` slimmer and its dependency tree clearer.

Although both dependencies were confirmed as unused in the `core` module of `jenkins`, developers rejected our pull request. They argue that excluding such dependencies has no valuable repercussion for the project and might actually affect its clients, which is correct. For example, `constant-pool-scanner` is used by external components, e.g., the class `RemoteClassLoader` in the `remoting`[12] project relies on this library to inspect the bytecode of remote dependencies.

As shown in the following quote from an experienced developer of Jenkins, there is a consensus on the usefulness of removing bloated dependencies, but developers need strong facts to support the removal of transitive dependencies:

> Dependency removals and exclusions are really useful, but my recommendation would be to avoid them if there is no substantial gain.

### 5.4.2 Auto

DEPCLEAN reports on the bloated-transitive dependencies `listenablefuture` and `auto-value-annotations` in module `auto-common` of the Google `auto` project. We proposed the exclusion of these dependencies and submitted a pull request with the POM change.

Developers express several concerns related to the exclusion of these dependencies. For example, a developer believes that it is not worth maintaining exclusion lists for dependencies that cause no problem. They point out that although `listenableFuture` is a single class file dependency, its presence in the dependency tree is vital to the project, since it overrides the version of the `guava` library that have many classes. Therefore, the inclusion of this dependency is a strategy followed by `guava` to narrow the access to the interface `ListenableFuture` and not to the whole library.[13]

[12] https://github.com/jenkinsci/remoting

[13] https://groups.google.com/forum/#!topic/guava-announce/Km82fZG68Sw/discussion

**Table 5** List of pull requests proposing the exclusion of bloated-transitive dependencies

| Project | Pull-request URL (https://github.com/) | Excluded dependencies | | PR* |
|---|---|---|---|---|
| | | #T | Total | |
| `jenkins [core]` | jenkinsci/jenkins/pull/4378 | 2 | 2 | ✗ |
| `auto [common]` | google/auto/pull/789 | 2 | 2 | ✗ |
| `moshi [moshi-kotlin]` | square/moshi/pull/1034 | 3 | 3 | ✗ |
| `spoon [core]` | INRIA/spoon/pull/3167 | 4 | 31 | ✓ |
| `moshi [moshi-kotlin]` | square/moshi/pull/1034 | 3 | 3 | ✗ |
| `wc-capture [driver-openimaj]` | sarxos/webcam-capture/pull/750 | 1 | 1 | ✳ |
| `teavm [core]` | konsoletyper/teavm/pull/439 | 1 | 2 | ✳ |
| `tika [parsers]` | apache/tika/pull/299 | 1 | 2 | ✳ |
| `orika [eclipse-tools]` | orika-mapper/orika/pull/328 | 1 | 2 | ✓ |
| `accumulo [core]` | apache/accumulo/pull/1421 | 3 | 3 | ✓ |
| `para [core]` | Erudika/para/pull/69 | 1 | 20 | ✓ |
| `selenese-runner-java` | vmi/selenese-runner-java/pull/313 | 2 | 9 | ✓ |
| `commons-configuration` | apache/commons-configuration/pull/40 | 2 | 9 | ✳ |
| Acceptance rate | – | 11/27 | 65/75 | 5/9 |

*Status of the pull request: ✓ Accepted. ✗ Rejected. ✳ Pending

On the other hand, developers agree that `auto-value-annotations` is bloated. However, they keep it, arguing that it is a test-only dependency, and they prefer to keep annotation-only dependencies and let end users exclude them when desired.

The response from developers suggests that bloated dependencies with test scope are perceived as less harmful. This is reasonable since test dependencies are only available during the test, compilation, and execution phases and are not shipped transitively in the JAR of the artifact. However, we believe that although it is a developers' decision whether they keep this type of bloated dependency or not, the removal of testing dependencies is regularly a desirable refactoring improvement.

#### 5.4.3 Moshi

DEPCLEAN detects that the bloated-transitive dependency `kotlin-stdlib-common` is present in the dependency tree of modules `moshi-kotlin`, `moshi-kotlin-codegen`, and `moshi-kotlin-tests` of project `moshi`. This dependency is induced from a common dependency of these modules: `kotlin-stdlib`.

Developers rejected our pull requests, arguing that excluding such transitive dependency prevents the artifacts from participating in the proper dependency resolution of their clients. They suggest that clients interested in reducing the size of their projects can use specialized shrinking tools, such as ProGuard,[14] for this purpose.

Although the argument of developers is valid, we believe that delegating the task of bloat removal to their library clients imposes an unnecessary burden on them. On the other hand, recent studies reveal that library clients do not widely adopt the usage of dependency analysis tools for quality analysis purposes (Nguyen et al. 2020).

#### 5.4.4 Spoon

DEPCLEAN detects that the transitive dependencies `org.eclipse.core.resources`, `org.eclipse.core.runtime`, `org.eclipse.core.filesystem`, and `org.eclipse.text` `org.eclipse.jdt.core` are bloated. All of these transitive dependencies were induced by the inclusion of the direct dependency `org.eclipse.jdt.core`, declared in the POM of `core` module of the `spoon` library.

Table 6 shows how the exclusion of these bloated-transitive dependencies has a positive impact on the size and the number of classes of the library. As we can see, by excluding these dependencies the size of the `jar-with-dependencies` of the `core` module of `spoon` is trimmed from 16.2MB to 12.7MB, which represents a significant reduction in size of 27.6%. After considering this improvements, the developers confirmed the relevance of this change and merged our pull request into the master branch of the project.

#### 5.4.5 Accumulo

DEPCLEAN detects the bloated-transitive dependencies `listenablefuture`, `httpcore` and `netty` in the `core` module of Apache `accumulo`. These dependencies were confirmed as bloated by the developers. However, they manifested their concerns regarding their exclusion, as expressed in the following comment:

[14] https://www.guardsquare.com/en/products/proguard

**Table 6** Comparison of the size and number of classes in the bundled JAR of the `core` module of `spoon`, before and after the exclusion of bloated-transitive dependencies

| | JAR Size(MB) | #Classes |
|---|---|---|
| Before | 16.2 | 7,425 |
| After | 12.7 | 5,593 |
| Reduction(%) | 27.6% | 24.7% |

> I'm not sure I want us to take on the task of maintaining an exclusion set of transitive dependencies from all our deps POMs, because those can change over time, and we can't always know which transitive dependencies are needed by our dependencies.

After the discussion, developers decided to accept and merge the pull request. Overall, developers considered that the proposal is a good idea. They suggest that it would be better to approach the communities of each of the direct dependencies that they use, and encourage them to mark those dependencies as *optional*, thus they would not be automatically inherited by their users.

### 5.4.6 Para

DepClean detects the bloated-transitive dependency `flexmark-jira-converter`. This dependency is induced through the direct dependency `flexmark-ext-emoji`, declared in the `core` module of the `para` project. Our further investigation on the Maven dependency tree of this module revealed that this bloated dependency adds a total of 19 additional dependencies to the dependency tree of the project, of which 15 are detected as duplicated by Maven.

Because of this large number (19) of bloated-transitive dependencies removed, developers accepted the pull request and merged the change into the master branch of the project the same day of the pull request submission.

**Answer to RQ4:** We used DepClean to propose 13 pull requests to exclude bloated-transitive dependencies, with 9 answered, 5/9 pull requests with response were accepted and merged by developers (65 dependencies were removed from the dependency tree of 5 projects).
**Implications:** The handling of bloated-transitive dependencies is a topic with no clear consensus among developers. Developers consider this kind of bloat as relevant, but they are concerned about the maintenance of a list of exclusion directives. Some developers agree to remove them based on practical facts (e.g., JAR size reduction), while other developers prefer to keep bloated-transitive dependencies in order to avoid the potential negative impact on their clients.

## 6 Discussion

In this section, we discuss the implications of our findings and the threats to the validity of the results obtained.

### 6.1 Implications of Results

Our results indicate that most of the dependency bloat is due to transitive dependencies and the Maven dependency inheritance mechanism. This suggests that the Maven dependency resolution strategy, which always picks the dependency that is closer to the root of the tree, may not be the best selection criterion for minimizing transitive dependency bloat. The official Maven dependency management guidelines[15] encourage developers to take control over the dependency resolution process via explicit declaration of dependencies in the POM file. This is a good practice to provide better documentation for the project and to keep one's artifact dependencies independent of the choices of other libraries down the dependency tree. Dependencies declared in this way have priority over the Maven mediation mechanism, allowing developers to have a clear knowledge about which library version they are expecting to be used through transitive dependencies. However, since backward compatibility is not always guaranteed, having fixed transitive dependency versions, and therefore non-declared dependencies, still remains as a widely accepted practice. In this context, the introduction of the module construct in Java 9 provides a higher level of aggregation above packages. This new language element, if largely adopted, may help to reduce the transitive explosion of dependencies. Indeed, this mechanism enables developers to fine tune public access restrictions of API members, explicitly declaring what set of functionalities a module can expose to other modules. This leads to two benefits: (1) it enables reuse declaration at a finer grain than dependencies, and (2) it makes the debloat techniques described in this work safer as it constrains reflection to white-listed modules.

Our results show that even notable open-source projects, which are maintained by development communities with strict development rules, are affected by dependency bloat. Developers confirmed and removed most of the reported bloated-direct dependencies detected by DepClean. However, they are more careful about excluding bloated-transitive dependencies. The addition of exclusion clauses to the POM files is perceived by some developers as an unnecessary maintainability burden. Interestingly, our quantitative results indicate that bloated-transitive dependency relationships represent the largest portion of bloated dependencies, yet, our qualitative study reveals that these bloated relationships are also the ones that developers find the most challenging to handle and reason about. Overall, this work opens the door to new research opportunities on debloating POMs and other build files.

### 6.2 Threats to Validity

In the following, we discuss construct, internal and external threats to the validity of our study.

**Construct Validity** The threats to construct validity are related to the novel concept of bloated dependencies and the metrics utilized for its measurement. For example, the DUT constructed by DepClean could be incomplete due to issues during the resolution of the dependencies. We mitigate this threat by building DepClean on top of Maven plugins to collect the information about the dependency relationships. We also exclude from the study those artifacts for which we were unable to retrieve the full dependency usage information.

[15] https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

It is possible that developers repackage a library as a bundle JAR file along with its dependencies, or copy the source code of dependencies directly into their source code, in order to avoid dependency related issues. Consequently, DEPCLEAN will miss such dependencies, as they are not explicitly declared in the POM file. Thus, the analysis of dependencies can underestimate the part of bloated dependencies. However, considering the size of our dataset and the feedback obtained from actively maintained projects, we believe that these corner cases do not affect our main results.

**Internal Validity** The threats to internal validity are related to the effectiveness of DEPCLEAN to detect bloated dependencies. The dynamic features of the Java programming language, e.g., reflection or dynamic class loading present particular challenges for any automatic analysis of Java source code (Landman et al. 2017; Lindholm et al. 2014). Since DEPCLEAN statically analyzes bytecode, anything that does not get into the bytecode is not detected (e.g., constants, annotations with source-only retention police, links in Javadocs), which can lead to false positives. To mitigate this threat, DEPCLEAN can detect classes or class members that are created or invoked dynamically using basic constructs such as `class.forName("someClass")` or `class.getMethod("someMethod", null)`.

To evaluate the impact of this limitation in practice, we ran DEPCLEAN on 10 additional popular projects. The experiment consists in running the test suite of the projects with the debloated version of the POM files, i.e., relying on dynamic analysis as a validation mechanism. Table 7 shows the results obtained after running the test suite of the version of the project without bloated dependencies. The first column shows the URL of the project on GitHub, the second and third columns represent the number bloated-direct and bloated-transitive dependencies detected by DEPCLEAN, and the fourth column is the result of the test (✓ pass, or ✗ fail). As we observe, 9/10 projects pass the test suite, and only one project fails: `raft-java`. We found that the reason of the failure was the dependency `org.projectlombok:lombok:1.18.4`, which heavily relies on reflection and other dynamic mechanisms of Java. To prevent the occurrence of false positives, the users of DEPCLEAN can add dependencies that are known to be used only dynamically to an exclusion list. Once added this dependency to the exclusion list of DEPCLEAN, it is not considered as bloated, and all the tests pass with the other bloated dependencies removed.

**Table 7** Evaluation of the results of DepClean by checking if all the test pass after the removal of bloated dependencies

| URL (https://github.com/) | #bd | #bt | Test result |
|---|---|---|---|
| pf4j/pf4j | 3 | 3 | ✓ |
| apilayer/restcountries | 5 | 13 | ✓ |
| modelmapper/modelmapper | 2 | 14 | ✓ |
| xtuhcy/gecco | 0 | 3 | ✓ |
| yaphone/itchat4j | 1 | 1 | ✓ |
| electronicarts/ea-async | 0 | 7 | ✓ |
| twitter/hbc | 0 | 1 | ✓ |
| skyscreamer/JSONassert | 0 | 1 | ✓ |
| wenweihu86/raft-java | 2 | 8 | ✗ |
| liaochong/myexcel | 0 | 2 | ✓ |

**External Validity** The relevance of our findings in other software ecosystems is one threat to external validity. Our observations about bloated dependencies are based on Java and the Maven ecosystem and our findings are restricted to this scope. More studies on other dependency management systems are needed to figure out whether our findings can be generalized. Another external threat relates to the representativeness of the projects considered for the qualitative study. To mitigate this threat, we submitted pull requests to a set of diverse, mature, and popular open-source Java projects that belong to distinct communities and cover various application domains. This means that we contributed to improving the dependency management of projects that are arguably among the best of the open-source Java world, which aims to get as strong external validity as possible.

## 7 Related Work

In this work, we propose the first systematic large-scale analysis of bloat in the Maven ecosystem. Here, we discuss the related works in the areas of software debloating and dependency management.

### 7.1 Analysis and Mitigation of Software Bloat

Previous studies have shown that software tends to grow over time, whether or not there is a need for it (Holzmann 2015; Quach et al. 2017). Consequently, software bloat appears as a result of the natural increase of software complexity, e.g., the addition of non-essential features to programs (Brooks 1987). This phenomenon comes with several risks: it makes software harder to understand and maintain, increases the attack surface, and degrades the overall performance. Our paper contributes to the analysis and mitigation of a novel type of software bloat: *bloated dependencies*.

Celik et al. (2016) presented MOLLY, a build system to lazily retrieve dependencies in CI environments and reduce build time. For the studied projects, the build time speed-up reaches 45% on average compared to Maven. DEPCLEAN operates differently than MOLLY: it is not an alternative to Maven as MOLLY is, but a static analysis tool that allows Maven users to have a better understanding and control about their dependencies.

Yu et al. (2003) investigated the presence of unnecessary dependencies in header files of large C projects. Their goal was to reduce build time. They proposed a graph-based algorithm to statically remove unused code from applications. Their results show a reduction of build time of 89.70% for incremental builds, and of 26.38% for fresh builds. Our work does not focus on build performance, we analyze the pervasiveness of dependency bloat across a vast and modern ecosystem of Maven packages.

In recent years, there has been a notable interest in the development of debloating techniques for program specialization. The aim is to produce a smaller, specialized version of programs that consume fewer resources while hardening security (Azad et al. 2019). They range from debloating command line programs written in C (Sharif et al. 2018), to the specialization of JavaScript frameworks (Vázquez et al. 2019) and fully fledged containerized platforms (Rastogi et al. 2017). Most debloating techniques are built upon static analysis and are conservative in the sense that they focus on trimming unreachable code (Jiang et al. 2016), others are more aggressive and utilize advanced dynamic analysis techniques to remove potentially reachable code (Heath et al. 2019). Our work addresses the same challenges at a coarser granularity. DEPCLEAN removes unused dependencies, which is, according to our empirical results, a significant cause of program bloat.

Qiu et al. (2016) empirically show evidence that a considerable proportion of API members are not widely used, i.e., many classes, methods, and fields of popular libraries are not used in practice. (Pham et al. 2016) implement a bytecode based analysis tool to learn about the actual API usage of Android frameworks. Hejderup (2015) study the actual usage of modules and dependencies in the Rust ecosystem, and propose PRÄZI, a tool for constructing fine-grained call-based dependency networks (Hejderup et al. 2018). Lämmel et al. (2011) perform a large-scale study on API usage based on the migration of AST code segments. Other studies have focused on understanding how developers use APIs on a daily basis (Roover et al. 2013; Bauer et al. 2014). Some of the motivations include improving API design (Myers and Stylos 2016; Harrand et al. 2019) and increasing developers productivity (Lim 1994). All these studies hint at the presence of bloat in APIs. To sum up, our paper is the first empirical study that explores and consolidates the concept of bloated dependencies in the Maven ecosystem, and is the first to investigate the reaction of developers to bloated dependencies.

Program slicing (Horwitz et al. 1988; Sridharan et al. 2007; Binkley et al. 2019) is a program analysis technique used to compute the subset of statements ("slice") that affect the values of a given program. Static slicing removes unused code by computing a statement-based dependence graph and identifies the statements that are directly or transitively reachable from a seed on the graph. DEPCLEAN uses a similar approach for debloat, where the slices are bytecode calls between dependencies computed by backtracking usages between the artifact and its dependencies.

### 7.2 Dependency Management in Software Ecosystems

Library reuse and dependency management has become mainstream in software development. McIntosh et al. (2012) analyze the evolution of automatic build systems for Java (ANT and Maven). They found that Java build systems follow linear or exponential evolution patterns in terms of size and complexity. In this context, we interpret bloated dependencies as a consequence of the tendency of build automation systems of evolving towards open-ended complexity over time.

Decan et al. (2019, 2017) studied the fragility of packaging ecosystems caused by the increasing number of transitive dependencies. Their findings corroborate our results, showing that most clients have few direct dependencies but a high number of transitive dependencies. They also found that popular libraries tend to have larger dependency trees. However, their work focuses primarily on the relation between the library users and their direct providers and does not take into account the inherited or transitive dependencies of those providers. We are the first, to the best of our knowledge, to conduct an empirical analysis of bloated dependencies in the Maven ecosystem considering both, users and providers, as potential sources of software bloat.

Bezemer et al. (2017) performed a study of unspecified dependencies, i.e., dependencies that are not explicitly declared in the build systems. They found that these unspecified dependencies are subtle and difficult to detect in make-based build systems. Seo et al. (2014) analyzed over 26 millions builds in Google to investigate the causes, types of errors made, and resolution efforts to fix the failing builds. Their results indicate that, independent of the programming language, dependency errors are the most common cause of failures, representing more than two thirds of fails for Java. Based on our results, we hypothesize that removing dependency bloat would reduce spurious CI errors related to dependencies.

Jezek and Dietrich (2014) describe, with practical examples, the issues caused by transitive dependencies in Maven. They propose a static analysis approach for finding missing, redundant, incompatible, and conflicting API members in dependencies. Their experiments, based on a dataset of 29 Maven projects, show that problems related to transitive dependency are common in practice. They identify the use of wrong dependency scopes as a primary cause of redundancy. Our quantitative study extends this work to the scale of the Maven Central ecosystem, and provides additional evidence about the persistence of the dependency redundancy problem.

Callo Arias et al. (2011) performed a systematic review about dependency analysis solutions in software-intensive systems. Bavota et al. (2015) studied performed an empirical study on the evolution of declared dependencies in the Apache community. They found that build system specifications tend to grow over time unless explicit effort is put into refactoring them. Our qualitative results complement previous studies that present empirical evidence that developers do not systematically update their dependency configuration files (McIntosh et al. 2014; Kula et al. 2018).

## 8 Conclusion

In this work, we presented a novel conceptual analysis of a phenomenon originated from the practice of software reuse, which we coined as *bloated dependencies*. This type of dependency relationship between software artifacts is intriguing: from the perspective of the dependency management systems that are unable to avoid it, and from the standpoint of developers who declare dependencies but do not use them in their applications.

We performed a quantitative and qualitative study of bloated dependencies in the Maven ecosystem. To do so, we implemented a tool, DEPCLEAN, which analyzes the bytecode of an artifact and all its dependencies that are resolved by Maven. As a result of the analysis, DEPCLEAN provides a report of the bloated dependencies, as well as a new version of its POM file which removes the bloat. We use DEPCLEAN to analyze the 723,444 dependency relationships of 9,639 artifacts in Maven Central. Our results reveal that 75.1% of them are bloated (2.7% are direct dependencies, 15.4% are inherited from parent POMs, and 57% are transitive dependencies). Based on these results, we distilled two possible causes: the cascade of unwanted transitive dependencies induced by direct dependencies, and the dependency heritage mechanism of multi-module Maven projects.

We complemented our quantitative study of bloated dependencies with an in-depth qualitative analysis of 30 mature Java projects. We used DEPCLEAN to analyze these projects and submitted the results obtained as pull request on GitHub. Our results indicated that developers are willing to remove bloated-direct dependencies: 16 out of 17 answered pull requests were accepted and merged by the developers in their code base. On the other hand, we found that developers tend to be skeptical regarding the exclusion of bloated-transitive dependencies: 5 out of 9 answered pull requests were accepted. Overall, the feedback from developers revealed that the removal of bloated dependencies clearly worth the additional analysis and effort.

Our study stresses the need to engineer, i.e., analyze, maintain, test POM files. The feedback from developers shows interest in DEPCLEAN to address this challenge. While the tool is robust enough to analyze a variety of real-world projects, developers now ask questions related to the methodology for dependency debloating, e.g., when to analyze bloat? (in every build, in every release, after every POM change), who is responsible for debloat of direct

or transitive dependencies? (the lead developers, any external contributor), how to properly managing complex dependency trees to avoid dependency conflicts? These methodological questions are part of the future work to further consolidate DEPCLEAN.

**Acknowledgments** This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation.

**Funding** Open Access funding provided by Royal Institute of Technology

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

## References

Azad BA, Laperdrix P, Nikiforakis N (2019) Less is more: Quantifying the security benefits of debloating web applications. In: Proceedings of the 28th USENIX conference on security symposium, SEC, pp 1697–1714, USA, USENIX Association

Bauer V, Eckhardt J, Hauptmann B, Klimek M (2014) An exploratory study on reuse at Google. In: Proceedings of the 1st International workshop on software engineering research and industrial practices, SERIP. ACM, New York, pp 14–23

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

Benelallam A, Harrand N, Soto-Valero C, Baudry B, Barais O (2019) The Maven dependency graph: a temporal graph-based representation of Maven Central. In: 16th international conference on mining software repositories (MSR). IEEE/ACM, Montreal

Bezemer C.-P., McIntosh S, Adams B, German DM, Hassan AE (2017) An empirical study of unspecified dependencies in make-based build systems. Empir Softw Eng 22(6):3117–3148

Binkley D, Gold N, Islam S, Krinke J, Yoo S (2019) A comparison of tree-and line-oriented observational slicing. Empir Softw Eng 24(5):3077–3113

Brooks FP (1987) No silver bullet: Essence and accidents of software engineering. Computer 20(4):10–19

Callo Arias TB, van der Spek P, Avgeriou P (2011) A practice-driven systematic review of dependency analysis solutions. Empir Softw Eng 16(5):544–586

Celik A, Knaust A, Milicevic A, Gligoric M (2016) Build system with lazy retrieval for java projects. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, FSE,. ACM, New York, pp 643–654

Cox R (2019) Surviving software dependencies. Commun ACM 62(9):36–43

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in OSS packaging ecosystems. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering, SANER, pp 2–12

Decan A, Mens T, Grosjean P (2019) An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir Softw Eng 24(1):381–416

Gkortzis A, Feitosa D, Spinellis D (2019) A double-edged sword? Software reuse and potential security vulnerabilities. In: Reuse in the big data era. Springer International Publishing, pp 187–203

Harrand N, Benelallam A, Soto-Valero C, Barais O, Baudry B (2019) Analyzing 2.3 million Maven dependencies to reveal an essential core in APIs. arXiv:1908.09757

Heath B, Velingker N, Bastani O, Naik M (2019) PolyDroid: Learning-driven specialization of mobile applications. arXiv:1902.09589

Hejderup J (2015) In dependencies we trust: How vulnerable are dependencies in software modules? PhD thesis, Delft University of Technology

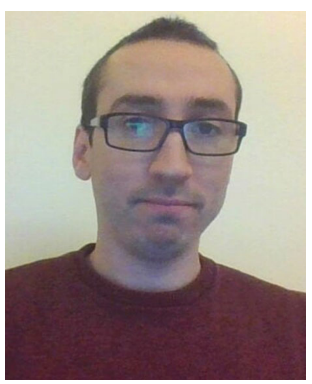
Hejderup J, Beller M, Gousios G (2018) PRÄZI: From package-based to precise call-based dependency network analyses. Delft University of Technology
Holzmann GJ (2015) Code inflation. IEEE Softw 1(2):10–13
Horwitz S, Reps T, Binkley D (1988) Interprocedural slicing using dependence graphs. SIGPLAN Not 23(7):35–46
Jezek K, Dietrich J (2014) On the use of static analysis to safeguard recursive dependency resolution. In: 2014 40th EUROMICRO conference on software engineering and advanced applications, pp 166–173
Jiang Y, Wu D, Liu P (2016) JRed: Program customization and bloatware mitigation based on static analysis. In: 2016 IEEE 40th annual computer software and applications conference (COMPSAC), vol 1, pp 12–21
Krueger CW (1992) Software reuse. ACM Comput Surv 24(2):131–183
Kula RG, German DM, Ouni A, Ishio T, Inoue K (2018) Do developers update their library dependencies? Empir Softw Eng 23(1):384–417
Lämmel R, Pek E, Starek J (2011) Large-scale, AST-based API-usage analysis of open-source java projects. In: Proceedings of the 2011 ACM symposium on applied computing, SAC ’11. ACM, New York, pp 1317–1324
Landman D, Serebrenik A, Vinju JJ (2017) Challenges for static analysis of java reflection - literature review and empirical study. In: 2017 IEEE/ACM 39th international conference on software engineering (ICSE), pp 507–518
Lim WC (1994) Effects of reuse on quality, productivity, and economics. IEEE Softw 11(5):23–30
Lindholm T, Yellin F, Bracha G, Buckley A (2014) The java virtual machine specification. Pearson Education
McIntosh S, Adams B, Hassan AE (2012) The evolution of java build systems. Empir Softw Eng 17(4):578–608
McIntosh S, Poehlmann M, Juergens E, Mockus A, Adams B, Hassan AE, Haupt B, Wagner C (2014) Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In: Companion proceedings of the 36th international conference on software engineering, ICSE Companion 2014. Association for Computing Machinery, New York, pp 145–154
Myers BA, Stylos J (2016) Improving API usability. Commun ACM 59(6):62–69
Naur P., Randell B. (eds) (1969) Software engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968, Brussels, Scientific Affairs Division, NATO. Newcastle University, Newcastle upon Tyne
Nguyen PT, Di Rocco J, Di Ruscio D, Di Penta M (2020) CrossRec: Supporting software developers by recommending third-party libraries. J Syst Softw 161:110460
Pham HV, Vu PM, Nguyen TT et al (2016) Learning API usages from bytecode: A statistical approach. In: Proceedings of the 38th international conference on software engineering. ACM, pp 416–427
Qiu D, Li B, Leung H (2016) Understanding the API usage in Java. Info Softw Technol 73:81–100
Quach A, Erinfolami R, Demicco D, Prakash A (2017) A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In: Proceedings of the 2017 workshop on forming an ecosystem around software transformation. ACM, pp 65–70
Rastogi V, Davidson D, De Carli L, Jha S, McDaniel P (2017) Cimplifier: Automatically debloating containers. In: Proceedings of the 2017 11th Joint meeting on foundations of software engineering, ESEC/FSE 2017. ACM, New York, pp 476–486
Roover CD, Lämmel R, Pek E (2013) Multi-dimensional exploration of API usage. In: 21st International conference on program comprehension, ICPC, pp 152–161
Salza P, Palomba F, Di Nucci D, De Lucia A, Ferrucci F (2019) Third-party libraries in mobile apps. Empirical Software Engineering
Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers’ build errors: A case study (at Google). In: Proceedings of the 36th international conference on software engineering, ICSE 2014. ACM, New York, pp 724–734
Sharif H, Abubakar M, Gehani A, Zaffar F (2018) TRIMMER: Application specialization for code debloating. In: Proceedings of the 33rd ACM/EEE international conference on automated software engineering, ASE 2018. ACM, New York, pp 329–339
Shull F, Singer J, Sjøberg DI (2007) Guide to advanced empirical software engineering. Springer, Berlin
Soto-Valero C, Benelallam A, Harrand N, Barais O, Baudry B (2019) The emergence of software diversity in Maven Central. In: 16th international conference on mining software repositories, MSR 2019. ACM, New York, pp 1–10
Sridharan M, Fink SJ, Bodik R (2007) Thin slicing. In: Proceedings of the 28th ACM SIGPLAN conference on programming language design and implementation, PLDI’07. Association for Computing Machinery, New York, pp 112–122
Vázquez H, Bergel A, Vidal S, Pace JD, Marcos C (2019) Slimming Javascript applications: An approach for removing unused functions from Javascript libraries. Inf Softw Technol 107:18–29

Wu Y, Manabe Y, Kanda T, German DM, Inoue K (2017) Analysis of license inconsistency in large collections of open source projects. Empir Softw Eng 22(3):1194–1222

Yu Y, Dayani-Fard H, Mylopoulos J (2003) Removing false code dependencies to speedup software build processes. In: Proceedings of the 2003 conference of the centre for advanced studies on collaborative research, CASCON. IBM Press, pp 343–352

**Publisher's note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**César Soto-Valero** is a Ph.D. student in Software Engineering at KTH Royal Institute of Technology, Sweden. His research work focuses on leveraging static and dynamic program analysis techniques to mitigate software bloat. César received his MSc degree and BSc degree in Computer Science from Universidad Central "Marta Abreu" de Las Villas, Cuba.



**Nicolas Harrand** is a Ph.D. student in Software Engineering at KTH Royal Institute of Technology, Sweden. His current research is focused on automatic software diversification. Nicolas studied Computer Science and Applied Mathematics in Grenoble, France.

**Martin Monperrus** is a Professor of Software Technology at KTH Royal Institute of Technology. He was previously an associate professor at the University of Lille and an adjunct researcher at Inria. He received a Ph.D. from the University of Rennes and a Master's degree from the Compiègne University of Technology. His research lies in the field of software engineering with a current focus on automatic program repair, program hardening, and chaos engineering.

**Benoit Baudry** is a Professor of Software Technology at KTH Royal Institute of Technology in Stockholm, Sweden. He received his Ph.D. in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis, and automatic diversification. He has led the largest research group in software engineering at INRIA, as well as collaborative projects funded by the European Union, and software companies.

# Paper III

# A Longitudinal Analysis of Bloated Java Dependencies

César Soto-Valero
KTH Royal Institute of Technology
Sweden
cesarsv@kth.se

Thomas Durieux
KTH Royal Institute of Technology
Sweden
thomas@durieux.me

Benoit Baudry
KTH Royal Institute of Technology
Sweden
baudry@kth.se

## ABSTRACT

We study the evolution and impact of bloated dependencies in a single software ecosystem: Java/Maven. Bloated dependencies are third-party libraries that are packaged in the application binary but are not needed to run the application. We analyze the history of 435 Java projects. This historical data includes 48,469 distinct dependencies, which we study across a total of 31,515 versions of Maven dependency trees. Bloated dependencies steadily increase over time, and 89.2 % of the direct dependencies that are bloated remain bloated in all subsequent versions of the studied projects. This empirical evidence suggests that developers can safely remove a bloated dependency. We further report novel insights regarding the unnecessary maintenance efforts induced by bloat. We find that 22 % of dependency updates performed by developers are made on bloated dependencies, and that Dependabot suggests a similar ratio of updates on bloated dependencies.

## CCS CONCEPTS

• **Software and its engineering** → Software libraries and repositories.

## KEYWORDS

software bloat, dependencies, java

**ACM Reference Format:**

César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3468264.3468589

## 1 INTRODUCTION

Software is bloated. From single Unix commands [14] to web browsers [23], most applications embed a part of code that is unnecessary to their correct operation. Several debloating tools have emerged in recent years [15, 22, 23, 25, 27, 30] to address the security and maintenance issues posed by excessive code at various granularity levels. However, these works do not analyze the evolution of bloat over time. Understanding software bloat in the perspective of software evolution [13, 31, 33] is crucial to promote debloating tools towards software developers. In particular, developers, when proposed to adapt a debloating tool, wonder if a piece of bloated code might be needed in coming releases, or what is the actual issue with bloat.

This work proposes the first longitudinal analysis of software bloat. We focus on bloat among software dependencies [5, 7, 11, 28] in the Java/Maven ecosystem. Bloated dependencies are software libraries that are unnecessarily part of software projects, i.e., when the dependency is removed from the project, it still builds successfully. In previous work [30], we showed that the Maven ecosystem is permeated with bloated dependencies, and that they are present even in well maintained Java projects. Our study revealed that software developers are keen on removing bloated dependencies, but that removing code is a complex decision, which benefits from solid evidence about the actual benefits of debloating.

Motivated by these observations about bloated dependencies, we conduct a large scale empirical study about the evolution of these dependencies in Java projects. We analyze the emergence of bloat, the evolution of the dependencies statuses, and the impact of bloat on maintenance. We have collected a unique dataset of 31,515 versions of dependency trees from 435 open-source Java projects. Each version of a tree is a snapshot of one project's dependencies, for which we determine a status, i.e. bloated or used. We rely on DepClean, the state-of-the-art tool to detect bloated dependencies in Maven projects. We analyze the evolution of 48,469 distinct dependencies per project and we observe that 40,493/48,469 (83.5 %) of them are bloated at one point in time, in our dataset.

Our longitudinal analysis of bloated Java dependencies investigates both the evolution of bloat and, as well as its impact on the maintenance of dependencies. We first show a clear increasing trend in the number of bloated dependencies. Next, we investigate how the usage status of dependencies evolves over time. This analysis is a key contribution of our work where we demonstrate that a dependency that is bloated is very likely to remain bloated over subsequent versions of a project. We present the first observations about the impact of bloat on maintenance activities, and the role of Dependabot, a popular dependency management bot, on these activities. We observe that developers spend significant efforts updating dependencies, either as part of their regular maintenance efforts, or after a Dependabot suggestion, even though the dependency is bloated. Furthermore, we systematically investigate the root of the bloat emergence, and find that 84.3 % of the bloated dependencies are bloated as soon as they are added in the dependency tree of a project. To summarize, the contributions of this paper are:

- A longitudinal analysis of software dependencies' usage in 31,515 versions of Maven dependency trees of 435 Java projects.
- Evidence about the stability of bloat: once they are bloated, 89.2 % of direct dependencies remain bloated. This is a strong incentive to remove bloat.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*ESEC/FSE '21, August 23–28, 2021, Athens, Greece*
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468589

```
...
<dependencies>
    <dependency>
        <groupId>org.d1</groupId>
        <artifactId>d1</artifactId>
        <version>1.1.0</version>
    </dependency>
    <dependency>
        <groupId>org.d2</groupId>
        <artifactId>d2</artifactId>
        <version>0.1.1</version>
    </dependency>
    <dependency>
        <groupId>org.d3</groupId>
        <artifactId>d3</artifactId>
        <version>3.0.0</version>
    </dependency>
</dependencies>
...
```

**Figure 1: Dependency declaration.**

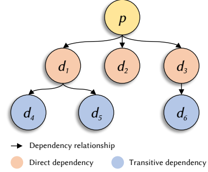



**Figure 2: Dependency tree.**

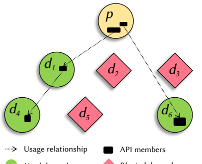



**Figure 3: Dependency usage.**

- Evidence that developers spend some unnecessary maintenance effort on bloated dependencies, including maintenance suggested by Dependabot.
- A qualitative analysis of the origin of bloated dependencies, which reveals that adding dependencies is the principal root cause for this type of software bloat.

## 2 BACKGROUND

In this work, we consider a software project as a collection of Java source code files and configuration files organized to be build with Maven.[1] In this section, we present the key concepts for the analysis of a project $p$ in the context of the set of its software dependencies, denoted as $\mathcal{D}$.

*Definition 2.1.* **Maven dependency:** A Maven dependency defines a relationship between a project $p$ and another compiled project $d \in \mathcal{D}$. Dependencies are compiled JAR files, uniquely identified with a triplet (`G:A:V`) where `G` is the `groupId`, `A` is the `artifactId`, and `V` is the `version`. Dependencies are defined within a scope, which determines at which phase of the Maven build cycle the dependency is required (e.g., `compile`, `test`, `runtime`).

A Maven project declares a set of direct dependencies in a specific configuration file known as `pom.xml` (acronym for "Project Object Model"), located at the root of the project. Figure 1 shows an excerpt of the dependency declaration in the `pom.xml` of a project $p$. In this example, developers explicitly declare the usage of three dependencies: $d_1$, $d_2$, and $d_3$. Note that the `pom.xml` of a Maven project is a configuration file subject to constant change and evolution: developers usually commit changes to add, remove, or update the version of a dependency.

*Definition 2.2.* **Direct dependency:** The set of direct dependencies $\mathcal{D}_{\text{direct}} \subset \mathcal{D}$ of a project $p$ is the set of dependencies declared in $p$'s `pom.xml` file. Direct dependencies are declared in the `pom.xml` by the developers, who explicitly manifest the intention of using the dependency.

*Definition 2.3.* **Transitive dependency:** The set of transitive dependencies $\mathcal{D}_{\text{transitive}} \subset \mathcal{D}$ of a project $p$ is the set of dependencies obtained from the transitive closure of direct dependencies. Transitive dependencies are resolved automatically by Maven, which means that developers do not need to explicitly declare these dependencies.

*Definition 2.4.* **Dependency tree:** The dependency tree of a Maven project $p$ is a direct acyclic graph of the dependencies of $p$, where $p$ is the root node and the edges represent dependency relationships between $p$ and the dependencies in $\mathcal{D}$.

To construct the dependency tree, Maven relies on its specific dependency resolution mechanism [1]. First, Maven determines the set of declared dependencies based on the `pom.xml` file of the project. Then, it fetches the JARs of the dependencies that are not present locally from external repositories, e.g., Maven Central.[2]

Figure 2 illustrates the dependency tree of the project $p$, which `pom.xml` file is in Figure 1. The project has three direct dependencies, as declared in its `pom.xml`, and three transitive dependencies, as a result of the Maven dependency resolution mechanism. $d_4$ and $d_5$ are induced transitively from $d_1$, whereas the transitive dependency $d_6$ is induced from $d_3$. Note that all the bytecode of these transitive dependencies is present in the classpath of project $p$, and hence they will be packaged with it, whether or not they are actually used by $p$.

*Definition 2.5.* **Bloated dependency:** A dependency $d \in \mathcal{D}$ in a software project $p$ is said to be bloated if there is no path in the dependency tree of $p$, between $p$ and $d$, such that none of the elements in the API of $d$ are used, directly or indirectly, by $p$.

We introduced the concept of bloated dependencies in 2020 [30]. Although they are present in the dependency tree of software projects, bloated dependencies are useless and, therefore, developers can consider removing them.

*Definition 2.6.* **Dependency usage status:** The usage status of a dependency $d \in \mathcal{D}$ determines if $d$ is *used* or *bloated* w.r.t. to $p$, at a specific time of the development of $p$.

Figure 3 shows an hypothetical example of the dependency usage tree of project $p$. Suppose that $p$ directly calls two sets of instructions in the direct dependency $d_1$ and the transitive dependency $d_6$. Then, the subset of instructions called in $d_1$ also calls instructions in $d_4$. In this case, the dependencies $d_1$, $d_4$, and $d_6$ are used by $p$, while dependencies $d2$, $d3$, and $d5$ are bloated dependencies.

Figures 1, 2 and 3 illustrate the status of a project's dependencies at some point in time. Yet, the `pom.xml` file, the dependency tree, and the status of dependencies are prone to change for several reasons. For example, a dependency that was used can become bloated after a dependency migration or after some refactoring activities that remove the usage link between the project and some of its dependencies. It is also possible that developers add dependencies in the `pom.xml` file or that more transitive dependencies appear in the tree, e.g., when updating the direct dependencies. This work

[1] https://maven.apache.org

[2] https://mvnrepository.com/repos/central

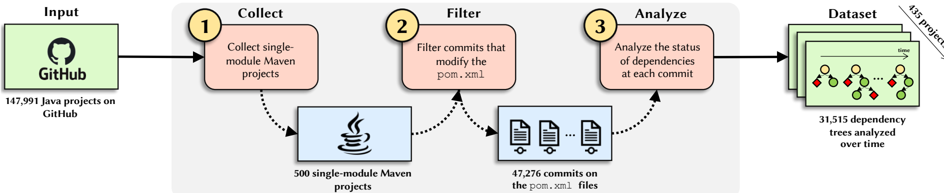



**Figure 4: Overview of our data collection pipeline. From a set of 147,991 Java projects on GitHub, we analyze the usage status of the dependencies in 435 Maven projects over time, to produce a dataset of 31,515 dependency trees.**

investigates these software evolution changes and their impact on bloat and maintenance.

## 3 STUDY DESIGN

In this section, we present the research protocols that we follow to conduct our empirical study, including the research questions (RQs), the tooling utilized to detect bloated dependencies, the data collection, and our methodology to address each RQ.

### 3.1 Research Questions

In this paper, we study four different aspects of bloated dependencies. Our analysis is guided by the following research questions.

**RQ1. How does the amount of bloated dependencies evolve across releases?** With this first question, we aim at consolidating the body of knowledge about software bloat. Several recent studies have shed light on the massive presence of bloat in different types of software projects [6, 15, 22, 25, 27]. The growth of bloat is an important motivation for these works. Yet, this growth has never been assessed nor quantified. Our first research question addresses this lack, analyzing the evolution of the amount of bloat over time.

**RQ2. Do bloated dependencies stay bloated across time?** Tools that remove bloated code are designed under the assumption that a piece of code that is bloated at some point in time will always be bloated, hence it makes sense to remove it. In this second research question, we investigate whether this assumption holds true in the case of bloated Java dependencies. We analyze how the usage status of dependencies evolves over time, from used to bloated, or vice versa.

**RQ3. Do developers maintain dependencies that are bloated?** Bloated dependencies needlessly waste time and resources, e.g., space on disk, build time, performance. However, one of the major issues related to this type of dependency is the unnecessary maintenance effort. In this research question, we investigate how often developers modify the `pom.xml` to update dependencies that are actually bloated.

**RQ4. What development practices change the usage status of dependencies?** The emergence of bloat is due to various code maintenance activities, e.g., refactoring the code, or modifying the `pom.xml`. In this research question, we expand the quantitative analysis of the status of each dependency and perform an in-depth analysis of the causes of dependency bloat.

### 3.2 Detection of Bloated Dependencies

To analyze the status of dependencies of Maven projects, we rely on DepClean.[3] This is an open-source tool that implements a practical way of detecting bloated dependencies in the complete dependency tree of a Java Maven project. DepClean runs a static analysis, at the bytecode level, to detect the usage of direct and transitive dependencies. To do so, DepClean constructs a static call-graph of API members' calls among the bytecode of the project and its dependencies. Then, it determines which dependencies are referenced, either directly by the project or indirectly via transitive dependencies. If none of the API members of a dependency are referenced, DepClean reports the dependency as bloated, i.e., the dependency is not necessary to build the project. DepClean generates a report with the status of each dependency, a list of API members that are used at least once, for each used dependency. The tool also generates a modified version of the `pom.xml` without bloated dependencies.

### 3.3 Data Collection

The dataset used in our study consists of a collection of subsequent versions of Maven dependency trees [9]. Each dependency in these trees is analyzed in order to determine its status: used or bloated. Figure 4 summarizes the process we follow to collect this dataset. Rounded rectangles represent procedures, non-rounded rectangles represent intermediate data results.

❶ ***Collect.*** Our data collection pipeline starts from the list of Java projects extracted from GitHub by Loriot et al. [20]. The authors queried the GitHub API on June 9th of 2020, and provide a list of GitHub URLs including all projects that use Java as the primary programming language. From this list, we keep only projects with more than 5 stars. This initial dataset contains a total of 147,991 Java projects. Then, we inspect the projects' files and select those containing a single `pom.xml` file in the root of the repository, to focus our longitudinal analysis on single-module Maven projects. This first data collection step provides a set of 34,560 Java projects.

❷ ***Filter.*** In this second step, we check all the commits on the `pom.xml` file to determine the version of the project declared in the `pom.xml`. Each time the version of the project changes and it is not a SNAPSHOT or a beta-version, we consider that the commit represents a new release. We sort the list of projects by the number of releases and then we select the first 500 projects. We focus on release commits since a release represents a stable version of the project, which is a suitable moment to consider the presence

[3] https://github.com/castor-software/depclean

**Table 1: Descriptive statistics of the dependencies in the 435 analyzed projects.**

| | Min | 1st Qu. | Median | Avg. | 3rd Qu. | Max |
|---|---|---|---|---|---|---|
| # Months | 5 | 48.5 | 75.5 | 81.01 | 109.5 | 235 |
| # Analyzed commits | 2 | 41.0 | 58.0 | 73.51 | 79.0 | 819 |
| # Direct initial | 0 | 3.0 | 5.0 | 8.28 | 10.0 | 120 |
| # Transitive initial | 0 | 2.0 | 10.5 | 46.77 | 56.0 | 300 |
| # Direct final | 0 | 5.0 | 10.0 | 13.97 | 18.0 | 111 |
| # Transitive final | 0 | 6.5 | 25.0 | 66.56 | 82.5 | 515 |

of bloated dependencies. In addition to the project releases, we collect the commits that have been created by Dependabot,[4] a popular software bot that automatizes the update of dependencies on GitHub [8]. The goal is to determine how many bloated dependencies have been updated as a result of a pull request not made by a human. We identify 2,017 Dependabot commits for 143/500 (28.6 %) projects. At the end of this step, we have a total of 500 projects, as well as 49,293 commits, including 47,276 release commits.

❸ ***Analyze.*** The final and most complex step in our pipeline is to analyze the status of dependencies in the 49,293 commits. We perform the following tasks: 1) clone the repository and checkout the commit, 2) compile the project using Maven, 3) if the project compiles, then we execute DepClean on the commit to obtain the dependency usage status. We analyze dependencies that have a `compile` or `test` scope. The compilation task is the most crucial and difficult task because it involves downloading dependencies, having the correct version of Java and having a proper project state, i.e., the Java code needs to be valid. We mitigate those problems by compiling the projects with Java 11 and then with Java 8. By trying to compile with Java 8 when the project does not compile with Java 11, we increase the number of successful compilations by around 20 %. We also use a proxy for Maven that caches and looks for dependencies in five different repositories to increase the chances to resolve them. In total, the proxy cached 198,611 dependencies and 165 Gb of data. As side effects, the proxy speeds up the resolution of dependencies and increases the reproducibility of the study, i.e., Maven will always resolve the same dependencies even if we recompile the projects after several years.

This final step of our pipeline outputs the definitive dataset for our longitudinal study: the dependency usage trees of 31,515 (63.9 %) commits collected from 435 (87.0 %) projects. These trees capture the history of 48,469 dependency relationships, including 1,987 direct dependencies and 23,442 transitive dependencies. Among the commits, 29,822 (63.1 %) are project releases and 1,693 (83.9 %) are Dependabot commits. We have kept only the projects for which we can successfully analyze at least two dependency tree versions. The dataset consists of a `JSON` file per commit for each project, containing the status of each dependency at every point in time. The dataset and the scripts are available in our experiment repository.[5]

Table 1 shows descriptive statistics of our dataset. The 435 projects have been active for periods ranging from five months to 235 months (12 years and 7 months), with most of them in the range 48.5 months (1st Qu.) to 109.5 months (3rd Qu.). The number of dependency trees analyzed for each project ranges from 2 to 819 (Median = 58, 1st Qu. = 41, 3rd Qu. = 79). The table also reports the number of direct dependencies in the oldest analyzed commit (Median = 5, 1st Qu. = 3, 3rd Qu. = 10), and transitive dependencies (Median = 10.5, 1st Qu. = 2, 3rd Qu. = 56). The last two lines in the table give the number of direct dependencies in the most recent analyzed commit (Median = 10, 1st Qu. = 5, 3rd Qu. = 18), and transitive dependencies (Median = 25, 1st Qu. = 6.5, 3rd Qu. = 82.5).

[4] https://dependabot.com

[5] https://github.com/castor-software/longitudinal-bloat

## 3.4 Methodology for RQ1

In RQ1, we analyze the evolution of the number of bloated dependencies over time. We start with a global analysis of the bloat trend in direct and transitive dependencies. To do so, we aggregate the total number of bloated dependencies in all projects on a monthly basis and compute the average values. Next, we look at each project separately and assign a bloat evolution trend to each of them. We represent the number of dependencies at each commit in a project as a time series. Let $p$ be a Maven project, $\mathcal{B}_p = b_1, b_2, ...b_n$ represents a time series of length $n$. A time step in this series represents one commit that modifies the `pom.xml` of $p$. Each $b_i$ is the total number of bloated dependencies reported by DepClean at the $i^{th}$ commit on the `pom.xml`. We collect two series for each project, for bloated-direct and bloated-transitive dependencies.

For each project $p$, we determine the overall trend for the evolution of the number of bloated dependencies: increase, decrease or stable. The following function over $\mathcal{B}_p$ shows how we determine the trend for a project:

$$f(\mathcal{B}_p) = \begin{cases} inc & \text{if } slope(lm(\mathcal{B}_p)) > 0 \land \exists b_j \in \mathcal{B}_p : b_j < b_{j-1} \\ dec & \text{if } slope(lm(\mathcal{B}_p)) < 0 \land \exists b_j \in \mathcal{B}_p : b_j > b_{j-1} \\ stable & \text{if } \forall b_i \in \mathcal{B}_p : b_i = b_{i-1} \end{cases}$$

We notice that several projects do not have a monotonic trend in the number of bloated dependencies (i.e. the value increases and decreases at different time intervals). To account for projects that have a non-monotonic number of bloated dependencies, we fit a simple linear regression model, denoted as *lm*, and determine the trend of the time series based on the sign of the *slope* of the regression line. A project labelled as *inc* is a project for which the sign of the *slope* is positive, i.e., the number of bloated dependencies increase over time. A project labelled as *dec* is a project for which the sign of the *slope* is negative, i.e., the number of bloated dependencies decreases over time. If the number of bloated dependencies is the same across all the data points in the time series of a project, we label it as *stable*.

## 3.5 Methodology for RQ2

In this research question, we analyze the evolution of the usage status of the 48,469 dependencies in our dataset. Given a dependency $d \in \mathcal{D}$, present in the dependency tree of a project $p$, we collect the status of $d$ at each analyzed commit (see data collection Section 3.3). This provides a sequence of usage statuses for $d$ and serves as the basis to determine the occurrence of transitional patterns between used and bloated statuses.

Let $\mathcal{V}_d$ be a vector representing the history of usage statuses of dependency $d$ across the releases of a project, where each release is ordered by its date. We label the usage status of a dependency $d$ as `B` if it is a bloated dependency, or `U` if it is a used dependency.

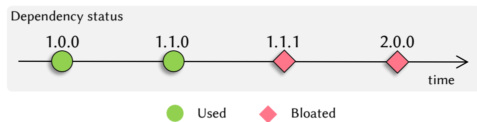



**Figure 5: Example of a dependency analyzed over time. It has a transition of usage status: from used at version 1.1.0 to bloated at 1.1.1 (RQ2). The dependency has two subsequent updates after bloated: at versions 1.1.1 and 2.0.0 (RQ3).**

Figure 5 illustrates a transition in the usage status of a dependency from used (U) to bloated (B). In this case, the dependency is identified as used at the two first releases of the project, then it becomes bloated at the third release, and stays as such. Therefore, the usage pattern for this dependency results in [U, U, B, B]. Since we are interested in analyzing transitional patterns, the consecutive elements of the same category in the vector can be compressed to a single status, e.g., the previous example is represented as [U, B].

In this research question, we focus on analysing the occurrence of five transitional patterns: [U], [B], [U,B], [B,U], and *fluctuating*. In the cases where the usage status of a dependency flickers over time, we consider the status of the dependency as *fluctuating*.

## 3.6 Methodology for RQ3

We conjecture that developers could save some maintenance efforts in the absence of bloated dependencies. In this research question, we investigate how many times developers update the version of dependencies that are in fact bloated. This type of change in the `pom.xml` of a project is an unnecessary engineering effort that could be avoided. We analyze two types of commits: the commits where the developers update the version of the project to a new stable version (e.g., 1.0.0), and the Dependabot commits. Dependabot[6] is a dependency management bot very active on GitHub. It creates pull requests that update the dependencies to remove known vulnerabilities. Dependabot was launched on May 26, 2017 with support for Ruby and JavaScript, and now it is supporting more than ten languages, including Java since August, 2018.

We analyze Dependabot commits because they only contain edits on the dependency versions in the `pom.xml`. It provides a clean point of analysis to detect the impact of a dependency update. And it allows us to study how many bloated dependencies are updated by developers as a result of the suggestion of automatic bots.

Figure 5 illustrates a typical case of a dependency that continues to be updated even after it becomes bloated. The dependency is used by the project until version 1.1.0. Afterward, the dependency is no longer used, but it is still updated twice, to version 1.1.1 and then to version 2.0.0.

To answer this research question, we consider the 15,230 commits in our dataset that perform dependency updates in projects that have at least one Dependabot commit. We obtain the number of times a dependency is updated by a developer, by Dependabot, and how many of those updates are performed on bloated dependencies. For the dependency usage analysis, we tag each dependency as used or bloated. We count every time the version of a direct dependency is updated, and we count separately the number of updates applied on bloated dependencies. In the example of Figure 5, we count one update on a used dependency (when the used dependency is updated to version 1.1.0), and two updates on a bloated dependency (when the bloated dependency is updated to version 1.1.1 and version 2.0.0). Using this approach, we can compare the ratio of updates made by developers and by Dependabot.

[6] https://dependabot.com

## 3.7 Methodology for RQ4

In this research question, we investigate the origins of bloated dependencies. Each time a bloated dependency appears for the first time in a project's history, we first determine if it was used in the commit that immediately precedes the apparition of bloat. If the dependency was used in the previous commits, we determine in which class it was used. By analyzing a dependency at the time it appears as bloated, we can identify what causes the emergence of bloat. We have identified four different situations:

1. New dependency (ND): The bloated dependency was not present in the previously analyzed commit. It indicates that the dependency was introduced in the project but never used.
2. Removed code (RC): The bloated dependency was present in the previously analyzed commit and all the classes where the dependency was used are removed.
3. Updated code (UC): The bloated dependency was present in the previously analyzed commit, yet at least one class where the dependency was used is still present in this commit. It means that the code has been updated to remove the usage of the dependency but the `pom.xml` still contains the dependency.
4. New version (NV): The bloated dependency was present in the previously analyzed commit and the version of the dependency changed. In the case of transitive dependency, the parent dependency has been updated and the project does not use the transitive dependency anymore.

For each of the 31,515 dependency trees, we identify the bloated dependencies. Then, we check the status of the dependency in the previous commit. If the dependency is not present in the previous commit, we consider the origin as ND. Otherwise, we check in the previous commit in which classes the bloated dependency is used. We then compare those classes with the new commit. If all classes are removed, we consider the origin of the bloat as RC. If at least one of the classes is still present, we consider the origin of the bloat as UC. Additionally, we compare the version of the bloated dependency with the previous commit. If the version changes, and at least one class is still present, we mark the origin as UC and NV, since both reasons could be the origin of the bloat.

# 4 RESULTS

In this section, we answer the four RQs presented in Section 3.1.

## 4.1 RQ1. Bloat Trend

In this research question, we analyze how the number of bloated dependencies evolves over time. We hypothesize that this number tends to grow. Following the protocol described in Section 3.3, we analyze the usage status of each dependencies in 31,515 dependency trees along the history of 435 projects, as reported by DepClean.

We assign bloat trend labels to each project, according to the three categories defined in Section 3.4.

Figure 6 shows the monthly evolution trend of the number of bloated-direct and bloated-transitive dependencies, from January 2011 to November 2020. The y-axis is the average number of bloated dependencies of the 435 projects. Each data point represents the average of bloat measured each month. The lines represent linear regression functions, fitted to show the trend of bloated-direct and bloated-transitive dependencies, at a 95% confidence interval.

We observe that bloated-transitive dependencies have a clear tendency to grow over time, whereas bloated-direct dependencies grow at significantly lower pace. For example, the number of bloated-transitive dependencies in 2011 was 1,695, and by the end 2020 this number grew up to 286,228 (increase > 250×). The bloat is more pervasive and variable (SD = 17.2) among transitive dependencies, representing a larger share in comparison with direct dependencies that are less numerous and less variable (SD = 1.3). We conclude that, overall, the amount of bloat increases, being more notable for transitive dependencies.

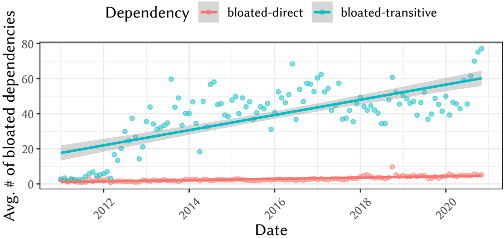



**Figure 6: Trend of the average number of bloated-direct and bloated-transitive dependencies per month.**

Figure 6 shows an overall growing trend for the number of bloated dependencies. Now, we look in more details at each project separately. We count the number of projects that have different trend of bloated dependencies. Figure 7 shows examples of time series of projects in our dataset for which the bloated-direct dependencies are labelled according to each category (increasing, decreasing, and stable). The name of the projects correspond to the <*user*>/<*repository*> on GitHub. The x-axis is the date of the analyzed commits. The y-axis represents the number of bloated dependencies detected. For instance, the time series of the project zapr-oss/druidry has a total of 51 commits on the `pom.xml` (i.e., data points in the time series), and it is labelled as *inc* w.r.t. to both the direct and transitive dependencies because both series tend to continuously increase over time.

Figure 8 shows the distribution of the trend of bloated-direct and bloated-transitive dependencies. The x-axis indicates the number of projects with bloated-direct dependencies in each specific evolution trend, given on the y-axis. Each bar in the plot is partitioned in three parts that correspond to the share of projects with a given trend for the number of bloated-transitive dependencies. For example, the top bar of Figure 8 shows (i) that the number of bloated-direct dependencies tends to increase for 245 (56.3 %) projects; and (ii) among these 245 projects, 180 also have a number of bloated-transitive

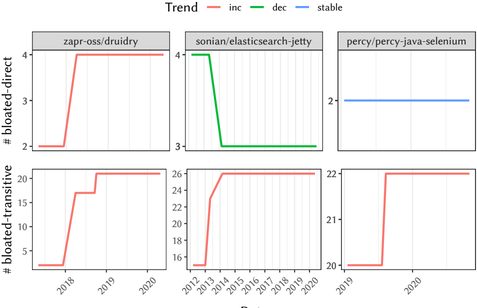



**Figure 7: Example of projects in the three classes of bloat trend defined in Section 3.4.**

dependencies that tends to increases, 59 of these projects have a decreasing number of bloated-transitive dependencies and 6 projects have a stable number of bloated-transitive dependencies. The bar in the middle of the figure indicates that the number of bloated-direct dependencies tends to decrease for 106 (24.4 %) projects and the bottom bar shows that this type of bloat is stable for 84 (19.3 %) projects because no new bloated dependencies are introduced in the `pom.xml`.

Looking at the partitions of each bar in Figure 8, we first observe that whatever the trend for the number of bloated-direct dependencies, the number of bloated-transitive dependencies can evolve in any way. Yet, the majority of projects have an increasing number of bloated dependencies among their transitive dependencies. In total, 286 (65.7 %) projects have an increasing number of bloated-transitive, whereas for 113 (26.0 %) projects this number decreases. The number of projects with stable transitive-dependencies, 36 (8.3 %), is relatively low.

Interestingly, from the 84 projects with a stable number of bloated-direct dependencies, 41 (48.8 %) of the bloated-transitive dependencies increase and 18 (21.4 %) decreases (e.g., as in the project percy/percy-java-selenium in Figure 7). This result indicates that the usage status of dependencies change regardless of the modification of the `pom.xml`. The transition from used to bloated in transitive dependencies becomes unnoticed. In other words, even if developers update only the version of direct dependencies, without

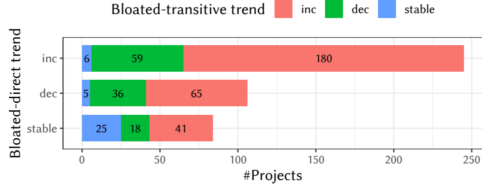



**Figure 8: Distribution of the number of projects with increasing, decreasing, and stable trend of bloated-direct and bloated-transitive dependencies.**

doing anything else, then the bloat grows naturally due to the inflation of the rest of the dependency tree. It happens, for example, in the project jpmml/jpmml-sparkml when a developer updates `spark-mllib_2.11` from version 2.0.0 to 2.2.0, introducing 133 new transitives dependencies.

On the other hand, we observe that for 65 (61.3 %) out of the 106 projects with a decreasing number of bloated-direct dependencies, the number of bloated-transitive increases. It indicates that even in projects for which direct dependencies decreases, the number of bloated-transitive dependencies can increase and eventually lead to a global growth of bloated dependencies for the project.

**Answer to RQ1:** Bloated dependencies tend to naturally emerge and grow through software evolution and maintenance. The number of bloated-direct dependencies and bloated-transitive dependencies increases over time for 56.3 % and 65.7 % of the projects, respectively.

## 4.2 RQ2. Bloated Across Time

This research question addresses an essential concern when developers think about removing bloat: is a piece of software identified as bloat at one point in time prone to usage in future revisions? We answer this question through a post-mortem analysis of the transitioning in the usage status of dependencies across the evolution of the studied projects. Our hypothesis is that dependencies do not change their usage status very frequently, i.e., a dependency that is used in one commit is used in future commits, and similarly for bloated dependencies. If our hypothesis holds, then it indicates that developers can be more confident when removing bloated dependencies.

We analyzed the five usage patterns described in Section 3.5. Figure 9 shows one concrete example for each pattern. The examples are taken from our dataset and the patterns are illustrated on the period January 2017 to December 2020. The y-axis shows the name of the direct dependency, with the pattern in square brackets. For example, we analyze the usage status of the direct dependency `h2` in the project dieselpoint/norm, from May 2018 to October 2020. As we can observe, this dependency was always reported as bloated. On the other hand, the dependency `json` in project PAXSTORE/paxstore-openapi-java-sdk was reported as bloated in first four analyzed commits, September 2018 to November 2019, and then it was used in all the subsequent releases of the project.

Figure 10 shows the distribution of the five transitional usage patterns among the 1,987 direct and 23,442 transitive dependencies in our dataset. The x-axis represents the percentage of occurrence of each pattern with respect to the total. The top bar of the plot indicates that 64.3 % of the direct dependencies are used through their whole lifespan, whereas 29.9 % are always bloated. This means that 94.2 % of direct dependencies never change their status through the evolution of the software projects. This also means that most bloated-direct dependencies are bloated by the time they are added in the dependency tree and are likely to remain bloated forever. We conjecture that this happens as a side effect of some development practices, such as copy-pasting of `pom.xml` files, the use of Maven Archetypes, or the deliberate addition of dependencies when setting up the development environment.

The bottom bar of the plot shows a similar stability for the status of transitive dependencies: 91.1 % of transitive dependencies do not change their usage status over their lifespan. A key difference here is that most of the dependencies are always bloated: 78.3 % of the transitive dependencies are bloated from the start, whereas 12.8 % are always used. We hypothesize that most transitive dependencies are unnoticed by the developers. Consequently, they are not managed and stay in the dependency tree for no reason in most cases.

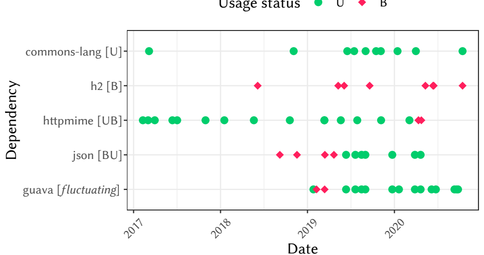



**Figure 9: Example of direct dependencies with distinct usage patterns. Each dependency belongs to a different project, the status of the dependency is analyzed at each commit that changes the `pom.xml` of the project.**

A key motivation for this research question is to determine whether a dependency identified as bloated is likely to stay bloated. We compute the percentage of dependencies bloated from the start (`B`) or that remain bloated after being used (`UB`), with respect to the total number of dependencies that are bloated at some point in the future, i.e., (`B+UB`)/(`B+UB+BU+`*fluctuating*). We find that 89.2 % of bloated-direct dependencies and 93.3 % of bloated-transitive dependencies remain bloated over time.

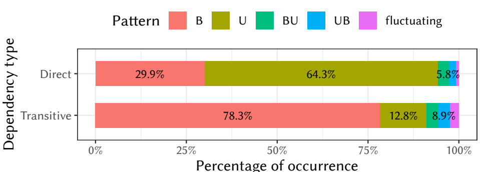



**Figure 10: Percentage of occurrence of usage patterns of bloated-direct and bloated-transitive dependencies.**

**Answer to RQ2:** A dependency that is detected as bloated most likely stays bloated: 89.2 % of direct-bloated dependencies and 93.3 % of the transitive-bloated dependencies stay bloated over time. This is strong evidence that developers can confidently take a debloating action when detecting bloated dependencies.

## 4.3 RQ3. Unnecessary Updates

In this research question, we investigate how the update of dependencies, a regular maintenance practice for all software projects,

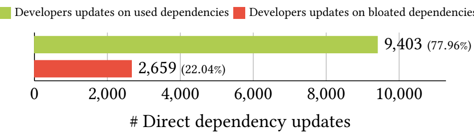



**Figure 11: Number of updates made by developers on direct dependencies in projects that use Dependabot.**

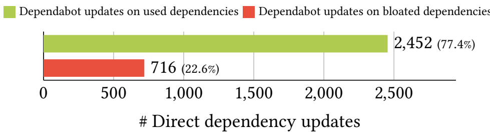



**Figure 12: Number of updates made by Dependabot on direct dependencies in projects that use Dependabot.**

more and more encouraged by automatic bots, relates to bloated dependencies. We hypothesize that developers invest some effort in updating some of these dependencies, while this is not required. To verify this hypothesis, we count how many times bloated-direct dependencies are updated in the `pom.xml` and compare it to the number of updates of used-direct dependencies. The methodology for this count is described in Section 3.6. We analyze separately the updates performed manually by developers and the updates suggested by Dependabot that are eventually accepted by a developer.

Figures 11 and 12 present our main results for this research question. Those plots present the number of dependency updates on direct dependencies made by developers and by Dependabot respectively. We focus on the 143 projects in our dataset that have at least one Dependabot commit. All the projects do not use Dependabot since its Java support is relatively recent (August 2018). The total number of updates on direct dependencies in these projects is 15,230, of which 12,062 have been performed by humans and 3,168 have been suggested by Dependabot.

Figure 11 shows that, over a total of 12,062 updates on direct dependencies made by developers, 9,403 (78.0 %) are preformed on used dependencies, and 2,659 (22.0 %) are made on bloated dependencies. These 2,659 unnecessary updates represent a significant effort, as updating dependencies is a non trivial maintenance task [17]. Figure 12 shows the number of updates on direct dependencies made by accepting a suggestion from Dependabot. From Figure 12, 2,452 (77.4 %) of Dependabot updates are performed on non-bloated dependencies and 716 (22.6 %) on bloated dependencies. Overall, we observe that developers perform a significantly larger number of dependency updates than Dependabot. Yet, the most interesting fact is that developers and Dependabot perform the same ratio of updates on bloated dependencies, 22.0 % and 22.6 % respectively.

The consequences of updating a bloated dependency are not only about the time and effort wasted by the developer. We have observed that a possible side-effect of these unnecessary updates is the increase of the total number of bloated dependencies in the project. In RQ1, we showed that the number of bloated dependencies increases over time, with a strong trend for transitive dependencies. In fact, a portion of this increasing transitive bloat is introduced through the update of direct dependencies, i.e., the new version has more dependencies. Note that this scenario can happen even when updating a bloated-direct dependency. We have observed this phenomenon in our dataset. The 6,091 updates on bloated-direct dependencies have introduced 1,883 new bloated-transitive dependencies.

> **Answer to RQ3:** Maintenance effort is lost on bloated dependencies: 22.0 % of developer updates and 22.6 % of Dependabot accepted updates are performed on bloated-direct dependencies. This represents a total of 6,091 updates over 143 projects. This is novel evidence that software bloat artificially increases maintenance effort and that dependency bots need to be improved to detect bloated dependencies.

## 4.4 RQ4. Bloat Origin

In this research question, we investigate what type of maintenance activity is at the origin of bloat emergence. In other words, we perform an in-depth analysis of the usage patterns `B` and `UB` presented in RQ2 by categorizing the origin of the bloat in four possible activities: new dependency (`ND`), removed code (`RC`), updated code (`UC`), and new version (`NV`) as described in Section 3.7. Table 2 summarizes the number of occurrences of activities that introduce bloat for direct and transitive dependencies. In total, we analyze the 25,359 dependencies that become bloated at some point in time (1,987 directs, 23,442 transitives) and determine in what condition they become bloated. This corresponds to 2,215 and 34,071 transitions to bloat, on direct and transitive dependencies respectively.

We observe that the primary origin of bloat is the addition of new dependencies `ND`, with 1,868 (84.3 %) such additions that lead to more bloated-direct dependencies and 33,370 (97.9 %) new dependencies that introduce more bloated-transitive dependencies. This result confirms our findings in RQ2, where we observed that the status of most dependencies does not change over time, which hinted to the fact that bloated dependencies are bloated as soon as they appear in the dependency tree. Additionally, the larger number of `ND` that grow the number bloated-transitive dependencies is consistent with the results of RQ1, where we showed a larger increase of bloated-transitive dependencies than bloated-direct ones. This new result consolidates the finding with the root cause of the transitive bloat. The second most frequent origin of bloated dependencies is different for direct and transitive dependencies. The action of removing code `RC` is the second most frequent cause of the emergence of bloated-direct dependencies, with 8, 8% of the cases. Updating code `UC` is the second most important root cause for bloated-transitive dependencies. While these two actions are similar in nature (evolve the code base), we did not find a clear explanation for the difference between the types of bloated dependencies. Updating to a new version of a dependency `NV` is the least frequent cause of bloat emergence. The rarity of this cause is explained by the fact that it can only happen in very specific conditions, when the new version of the dependency changes drastically.

**Table 2: Number of occurrence for each origin of bloat. The occurrences are separated between the new bloated-direct and transitive dependencies.**

| Origin | Bloated-direct | Bloated-transitive |
|---|---|---|
| New dependency (ND) | 1,868 (84.3 %) | 33,370 (97.9 %) |
| Removed code (RC) | 194 (8.8 %) | 206 (0.6 %) |
| Updated code (UC) | 153 (6.9 %) | 495 (1.5 %) |
| New version (NV) | 47 (2.1 %) | 124 (0.4 %) |

We now illustrate the different situations of bloat introduction with real-world case studies observed in our dataset. The most frequent cause of bloat introduction is a new transitive dependency in the dependency tree (ND), which is never used. For example, this happens in the project couchbase/couchbase-java-client at the commit 47ac44, where the dependency `jackson-databind`, which is induced transitively when `encryption:1.0.0` has been added to the `pom.xml`. `jackson-databind` is used in the class `HashicorpVaultKeyStoreProvider` which is never used by the couchbase/couchbase-java-client and, therefore, `jackson-databind` is a bloated-transitive dependency in this project.

This case occurs with direct dependencies as well. For example, the direct dependency `jackson-core` is added as a direct dependency in the `pom.xml` of project jenkinsci/elasticbox-plugin, at commit 008358. Yet, the dependency is never used in the code of the project. One year and 4 months later a pull-request, #41, fixes the bloat issue by removing the dependency directly. However, at the time of writing this paper, the pull-request has not been merged.

Projects are evolving, adding and removing code is part of the life cycle of a project. A consequence of code removal can be to eliminate the need for a dependency. Yet, developers currently have no tool support to determine that a dependency can also be removed as part of their maintenance activities. Consequently, the dependency is likely to become bloated (RC). For example, we observed that scenario happens in the project apache/commons-lang. The commit def3c4 introduces the dependency `bcel`, which contains annotations to document thread safety. However, the commit 796b05 removes all classes where these annotations were used. According to the commit, more discussions were needed to design the annotation, and the maintainers reverted partially the changes to release a new version. A developer removed the bloated dependency after five months (see commit 66226e).

A similar scenario occurs when developers update classes (UC). For example, the commit 62aad3 introduces the annotation `IgnoreJRERequirement` on a method in the project jenkinsci/remoting. However, this method is updated and deprecated in the commit 49c67e. The annotation `IgnoreJRERequirement` is removed and the dependency `animal-sniffer-annotation` became bloated.

The project apache/commons-dbcp contains an interesting case of bloat introduced when a dependency is updated (NV). In the commit 3550ad, the direct dependency `geronimo-transaction` is detected as bloated. However, this dependency was not bloated in the previous commit d7aa66, when the project was using the version `1.2-beta` of `geronimo-transaction`. The dependency was updated to version `2.2.1` with commit 3550ad. This new version brought major changes in the dependency and, in `2.2.1`, all the classes used by the project had been move in a transitive dependency of `geronimo-transaction`. Therefore, the direct dependency towards `geronimo-transaction` became bloated.

> **Answer to RQ4:** The addition of new dependencies is the root cause of the emergence of 84.3 % of the bloated-direct dependencies. Meanwhile, 15.7 % of bloated dependencies appear after code updates or removals. This indicates that new dependencies should be carefully reviewed the first time they are added, and we recommend developers to check the usage status of dependencies when removing code.

## 5 IMPLICATIONS

Our findings provide practical, empirically justified implications for improving dependency maintenance [7, 11]. The results of RQ1 and RQ2 show that a dependency that is bloated is likely to remain bloated in the future. This is empirical evidence that can motivate developers and increase their confidence when they are faced with the opportunity to remove bloated dependencies. Motivation comes from the benefits associated with reducing the number of dependencies of the project and hence reduce associated maintenance activities. Confidence comes with the strong likelihood that the dependency that is removed will not be necessary in the future.

Our results highlight several practical challenges to integrate the management of software dependencies on the development lifecycle. This can raise the awareness of developers about the importance of understanding what dependencies are more likely to become bloated, and how their projects can reduce the size of dependency trees without breaking the build. In particular, the use of tools, such as DepClean, to automatically detect and suggest changes in the build files can contribute to a better awareness of developers about the state of their dependencies. For example, we recommend to include a bloat analysis when preparing a major release of a project, to ensure that no bloat is shipped, distributed and deployed. This reduces the size of the released binary and hence the resources that are necessary to distribute it. This also reduces the number of transitive dependencies for all projects that depend on the new release.

In RQ3, we present original results of the negative impact of bloated dependencies on the maintenance of the projects. In particular, we shed a new light on the limitations of dependency bots, such as Dependabot, and provide evidence that developers accept bots' suggestions when updating dependencies without checking if the dependency is actually used. Bot creators should consider improving their tools to automatically detect bloat and suggest the removal of unused dependencies. On the same line, compilers and IDEs should also warn developers when dependencies are not used anymore and when a dependency is introduced without encountering its counterpart usage on code.

Our dataset and our case studies on the origin of bloat provide valuable references for the rapid identification of practices that result in dependency bloat. Those references can be used to build dedicated bots that ask for additional checks, e.g. when a new dependency appears in the dependency tree, or to establish guidelines for developers when they maintain `pom.xml` files.

## 6 THREATS TO VALIDITY

***Internal Validity.*** The first internal threat relates to the detection of bloated dependencies. The results of our study are tied to the accuracy of DepClean to find bloated dependencies in Maven projects. This tool is based on advanced static analysis. Therefore, some usages that rely on Java dynamic features might be missed, reporting some used dependencies as bloated. For example, `lombok` is a Java library that relies on annotations to manipulate the bytecode at compilation time, adding boilerplate code constructs such as getters and setters. This mechanism makes the dependency to be flagged as bloated by DepClean, since no reference to this dependency remains in the bytecode of the compiled project. Nevertheless, we consider that DepClean is a solid tool, evaluated on millions of dependencies, used in industry, and developers have removed hundreds of bloated dependencies thanks to its analysis [30]. The second threat relates to the representativeness of the data and the analysis performed. We mitigate those threats by collecting a large dataset of projects from multiple domains and released across several years. This allows us to draw general conclusions about the evolutionary trend of bloated dependencies, regardless of the existence of some false positives, which are known to be hard to detect using static analysis [18, 19, 26].

***External Validity.*** When conducting this study, we focus on bloated Java dependencies in projects that build with Maven. As explained in Section 3.3, the analysis of the dependency trees of 435 projects requires compiling and analyzing the bytecode at different time periods. There were cases where the compilation failed for several reasons, making it difficult to obtain the complete history of dependency changes. As we consider a large number of open-source projects, we believe our results are generalizable in this specific domain. Meanwhile, additional studies with proprietary projects or other programming languages should be considered to consolidate these very first result about software bloat evolution.

***Construct Validity.*** This threat is related to the rationality of the questions asked. We investigate the evolution of bloated dependencies over time. To achieve this goal, we focus on four aspects: bloat trend, usage patterns, unnecessary updates, and bloat origins. We believe that these are rational questions that provide unique and novel insights for researchers and developers.

## 7 RELATED WORK

***Software Bloat.*** Previous research on software bloat has mainly focused on reducing C/C++ binaries to mitigate the security risks associated with unnecessary code [22, 24, 27]. Holzmann [14] reports the historical growth in the size of the `true` command in Unix systems. Similarly, we observed that the number of bloated dependencies tends to grow over time, whether or not there is a need for it. In the last years, there is a recent resurgence of interest in debloating Java bytecode [6, 15, 21, 29, 30]. These tools remove Java bytecode using static and dynamic analysis. In contrast, our study focuses on the evolution and the emergence of bloat in Java projects, while spotting some of the current research gaps and tooling for effective dependency management. Other studies have focused on eliminating bloat in source code [34], binary shared libraries [2], highly configurable programs [16], or containers [25]. Other works have focused on improving the debloat process through various optimizations techniques [3, 4, 12, 32, 36]. As far as we know, we are the first to conduct a longitudinal study to analyze software bloat.

***Bloated Dependencies.*** Our work follows up on our previous study of bloated dependencies [30]. Our quantitative and qualitative study of bloated dependencies in the Maven ecosystem, revealed the importance of the phenomenon in Maven Central. Our interactions with software developers showed that removing bloated dependencies is perceived as a valuable contribution. Here we extend this previous study in two ways. First, we perform a study of bloated dependencies with distinct study subjects on a chronological basis. This brings novel insights on the evolution of bloated dependencies through the history of software projects, corroborating the importance of maintaining `pom.xml` files. We bring novel evidence in favor of removing bloated dependencies. Second, we perform a unique study on the interaction between maintenance activities and the emergence of bloat. These new results contribute to understanding the origin of bloat as well as estimating the maintenance effort unnecessarily invested when performing dependency updates.

***Software Bots.*** Erlenhov et al. [10] perform an empirical study about the interaction between practitioners and software bots. They found that there is currently a lack of general-purpose smart bots that go beyond simple automation tools, such as dependency version updating. This is in line with our results, as we have seen that dependency bots do not perform advanced dependency usage analysis, sending unnecessary warnings that could be avoided. Wessel et al. [35] rise attention on the inconvenient side of software bots. They present empirical evidence that pull requests made by bots are, in some cases, perceived as disruptive and unwelcoming by developers. Thus, motivating our work on reducing the number of warnings caused by bots dedicated to automatically update dependencies.

## 8 CONCLUSION

This paper presented the first large-scale longitudinal study about the evolution of software bloat, with a focus on bloated Java dependencies. We collected a unique dataset of 31,515 dependency tree versions, tagged with usage dependency status, from 435 Java projects hosted on GitHub. Through the analysis of 48,469 dependencies, we provided evidence about an essential phenomenon: 89.2 % of the dependencies that become bloated over evolution stay bloated over time. As a consequence, developers spend significant time updating dependencies that are actually bloated. We find that 22 % of dependency updates are made on bloated dependencies. These updates include a significant number of updates suggested by Dependabot. We also demonstrate that bloated dependencies are primarily originated from the addition of new dependencies that are never used, rather than from code changes.

Our work paves the way to better understand the importance of debloating tools, such as DepClean, to handle the increasing phenomenon of software bloat. In particular, evidence that bloated code stays bloated is important for developers who need to decide if they should remove code. Our novel findings about the role of Dependabot on the unnecessary maintenance effort provide concrete insights to improve the suggestions that this single bot shares with developers.

## ACKNOWLEDGMENTS

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by the TrustFull project funded by the Swedish Foundation for Strategic Research.

## REFERENCES

[1] [n.d.]. Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. Accessed: 2021-05-18.

[2] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-scale Debloating of Binary Shared Libraries. *Digital Threats: Research and Practice* 1, 4 (2020), 1–28. https://doi.org/10.1145/3414997

[3] Thibaud Antignac, David Sands, and Gerardo Schneider. 2017. Data Minimisation: A Language-Based Approach. In *ICT Systems Security and Privacy Protection - 32nd IFIP TC 11 International Conference, SEC 2017, Rome, Italy, May 29-31, 2017, Proceedings (IFIP Advances in Information and Communication Technology, Vol. 502)*, Sabrina De Capitani di Vimercati and Fabio Martinelli (Eds.). Springer, 442–456. https://doi.org/10.1007/978-3-319-58469-0_30

[4] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1697–1714.

[5] Paolo Boldi and Georgios Gousios. 2021. Fine-Grained Network Analysis for Modern Software Ecosystems. *ACM Trans. Internet Techn.* 21, 1 (2021), 1:1–1:14. https://doi.org/10.1145/3418209

[6] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern Java applications. In *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/3368089.3409738

[7] Russ Cox. 2019. Surviving software dependencies. *Commun. ACM* 62, 9 (2019), 36–43. https://doi.org/10.1145/3347446

[8] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and Characterizing Bots that Commit Code. In *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*. Association for Computing Machinery, New York, NY, USA, 209–219. https://doi.org/10.1145/3379597.3387478 arXiv:2003.03172

[9] Thomas Durieux, César Soto-Valero, and Benoit Baudry. 2021. DUETS: A Dataset of Reproducible Pairs of Java Library-Clients. In *IEEE International Working Conference on Mining Software Repositories*.

[10] Linda Erlenhov, Francisco Gomes De Oliveira Neto, and Philipp Leitner. 2020. *An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective*. Association for Computing Machinery, New York, NY, USA, 445–455. https://doi.org/10.1145/3368089.3409680 arXiv:2005.13969

[11] Tomas Gustavsson. 2020. Managing the Open Source Dependency. *Computer* 53, 2 (2020), 83–87. https://doi.org/10.1109/MC.2019.2955869

[12] Matthew Hague, Anthony W. Lin, and Chih-Duo Hong. 2019. CSS Minification via Constraint Solving. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 12:1–12:76. https://doi.org/10.1145/3310337

[13] Michael Hilton, Jonathan Bell, and Darko Marinov. 2018. A large-scale study of test coverage evolution. In *Proceedings of ASE*. ACM, 53–63.

[14] Gerard J. Holzmann. 2015. Code inflation. *IEEE Software* 32, 2 (March 2015), 10–13. https://doi.org/10.1109/MS.2015.40

[15] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings - International Computer Software and Applications Conference*, Vol. 1. IEEE Press., New York, 12–21. https://doi.org/10.1109/COMPSAC.2016.146

[16] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-Driven Software Debloating. In *Proceedings of the 12th European Workshop on Systems Security* (Dresden, Germany) *(EuroSec '19)*. Association for Computing Machinery, New York, NY, USA, Article 9, 6 pages. https://doi.org/10.1145/3301417.3312501

[17] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies?: An empirical study on the impact of security advisories on library migration. *Empirical Software Engineering* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5 arXiv:1709.04621

[18] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press., New York, 507–518. https://doi.org/10.1109/ICSE.2017.53

[19] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[20] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2020. Styler: Learning formatting conventions to repair checkstyle errors. *arXiv* 1 (2020), 1–1. arXiv:1904.01754

[21] Konner Macias, Mihir Mathur, Bobby R. Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: A Web Service for Debloating Java Bytecode. In *Proceedings of ESEC/FSE*. 1665–1669.

[22] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-Deployment Software Debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 1733–1750.

[23] Chenxiong Qian, Hyungjoon Koo, Chang Seok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the ACM Conference on Computer and Communications Security*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[24] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 869–886. https://doi.org/10.5555/3277203.3277269

[25] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically debloating containers. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vol. Part F130154. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3106237.3106271

[26] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 107–112. https://doi.org/10.1145/3236454.3236503

[27] Hashim Sharif, Ashish Gehani, Muhammad Abubakar, and Fareed Zaffar. 2018. Trimmer: Application specialization for code debloating. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[28] Cesar Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *IEEE International Working Conference on Mining Software Repositories (MSR '19, Vol. 2019-May)*. IEEE Press., New York, 333–343. https://doi.org/10.1109/MSR.2019.00059 arXiv:1903.05394

[29] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2020. Trace-based Debloat for Java Bytecode. *arXiv* 1, Article arXiv:2008.08401 (Aug. 2020), 12 pages. arXiv:2008.08401 [cs.SE]

[30] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering* 26, 3 (2021), 1–44.

[31] Diomidis Spinellis. 2017. A repository of Unix history and evolution. *Empir. Softw. Eng.* 22, 3 (2017), 1372–1404. https://doi.org/10.1007/s10664-016-9445-5

[32] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 361–371. https://doi.org/10.1145/3180155.3180236

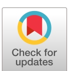
[33] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. 2014. A study of library migrations in Java. *J. Softw. Evol. Process.* 26, 11 (2014), 1030–1052. https://doi.org/10.1002/smr.1660

[34] H. C. Vázquez, A. Bergel, S. Vidal, J. A. Díaz Pace, and C. Marcos. 2019. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology* 107 (2019), 18–29. https://doi.org/10.1016/j.infsof.2018.10.009

[35] Mairieli Wessel and Igor Steinmacher. 2020. The Inconvenient Side of Software Bots on Pull Requests. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (Seoul, Republic of Korea) *(ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 51–55. https://doi.org/10.1145/3387940.3391504

[36] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program Debloating via Stochastic Optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/3377816.3381739

# Paper IV



# Coverage-Based Debloating for Java Bytecode

CÉSAR SOTO-VALERO, THOMAS DURIEUX, NICOLAS HARRAND, and BENOIT BAUDRY, KTH Royal Institute of Technology

Software bloat is code that is packaged in an application but is actually not necessary to run the application. The presence of software bloat is an issue for security, performance, and for maintenance. In this article, we introduce a novel technique for debloating, which we call coverage-based debloating. We implement the technique for one single language: Java bytecode. We leverage a combination of state-of-the-art Java bytecode coverage tools to precisely capture what parts of a project and its dependencies are used when running with a specific workload. Then, we automatically remove the parts that are not covered, in order to generate a debloated version of the project. We succeed to debloat 211 library versions from a dataset of 94 unique open-source Java libraries. The debloated versions are syntactically correct and preserve their original behaviour according to the workload. Our results indicate that 68.3 % of the libraries' bytecode and 20.3 % of their total dependencies can be removed through coverage-based debloating.

For the first time in the literature on software debloating, we assess the utility of debloated libraries with respect to client applications that reuse them. We select 988 client projects that either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the libraries that we debloat. Our results show that 81.5 % of the clients, with at least one test that uses the library, successfully compile and pass their test suite when the original library is replaced by its debloated version.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**; **Empirical software validation;**

Additional Key Words and Phrases: Software bloat, code coverage, program specialization, bytecode, software maintenance

**ACM Reference format:**
César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 38 (March 2023), 34 pages.
https://doi.org/10.1145/3546948

38

## 1 INTRODUCTION

Software systems have a natural tendency to grow in size and complexity over time [18, 22, 43, 56]. A part of this growth comes with new features or bug fixes, while another part is due to useless code that accumulates over time. This phenomenon, known as software bloat, increases when

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, as well as by the TrustFull and the Chains projects funded by the Swedish Foundation for Strategic Research.
Authors' address: C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, KTH Royal Institute of Technology, Department of Software and Computer Systems; emails: cesarsv@kth.se, tdurieux@kth.se, harrand@kth.se, baudry@kth.se.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1049-331X/2023/03-ART38 $15.00
https://doi.org/10.1145/3546948

building on top of software frameworks [3, 30, 44], as well as with code reuse [17, 50, 62]. Software debloating consists of automatically removing unnecessary code [19]. Automatic debloating poses several challenges: determine the location of the bloated parts [11, 42, 46], and remove these parts while preserving the original behavior and providing useful features. The problem of safely debloating real-world applications remains a long-standing software engineering endeavor today.

Most state-of-the-art debloating techniques target this problem using static analysis [26, 46, 49, 54] because it is scalable. Yet, the results lack precision in the presence of dynamic language features, which are prevalent in modern programming languages, and commonly used in practice [51]. Dynamic program analysis techniques outperform static approaches through the runtime collection of program usage information [11, 42]. However, capturing complete and precise dynamic usage information for debloating is challenging, especially at scale.

In this article, we introduce coverage-based debloating for Java bytecode. Our new approach, implemented in the **Java DeBLoater (JDBL)** tool, handles the challenge of capturing precise dynamic usage by leveraging the industry-standard dynamic analysis techniques implemented in software coverage tools. Based on this information, JDBL automatically transforms the bytecode of the compiled project to remove the bloated code. JDBL validates the syntactic correctness of the debloated project, as well as its behavior. To do so, it rebuilds the debloated project with the same configuration as the original and re-executes the test suite to check that the behavior of the original project is preserved.

The key technical contribution of our work consists in collecting accurate code coverage to minimize the risks of generating an ill-formed debloated software artifact (i.e., debloating and packaging a software project for reuse). The loss of information in the compilation from source to bytecode, as well as the existence of software elements that are required but are not executed, are two essential challenges to precisely capturing the code that can be safely removed. Additionally, coverage tools do not handle third-party libraries, which is a primary source of software bloat [1, 50, 63]. In JDBL, we aggregate the coverage data collected by four coverage tools, to address those challenges. The tools implement complementary, custom heuristics to cover the corner cases. JDBL also extends the Maven build mechanism to collect coverage information for third-party libraries.

We evaluate JDBL by debloating 211 versions from a dataset of 395 versions of 94 unique open-source Java libraries. This represents a total of 10M+ lines of code analyzed, 103,032 classes, and 187 unique third-party dependencies. We assess the effectiveness of our technique to preserve both syntactic correctness and the original behavior of these libraries. We quantify the impact of coverage-based debloating on the libraries' size at three granularity levels: number of removed methods, classes, and dependencies. JDBL finds that 60.1 % of classes are bloated, and 20.3 % of the third-party libraries can be completely removed. A comparison with JShrink [8], the state-of-the-art tool for Java debloating, indicates that JDBL achieves significantly larger reduction rates, while systematically preserving the original behavior.

For the first time in the literature of software debloating, we assess the usability of the debloated libraries with respect to actual usages, by building client programs that declare a dependency on these libraries. First, we check if the client program compiles correctly with the debloated library to assess binary compatibility. Then, we check if the program's test suite still passes. We evaluate the utility of coverage-based debloating with respect to 988 programs that have at least one direct reference to the debloated library in their source code. For 81.5 % of programs whose test suite covers at least one class of the library the test suite passes with the debloated libraries.

JDBL is a Java debloating tool that combines diverse coverage data sources with bytecode removal transformations. It validates the debloating results throughout the whole software build

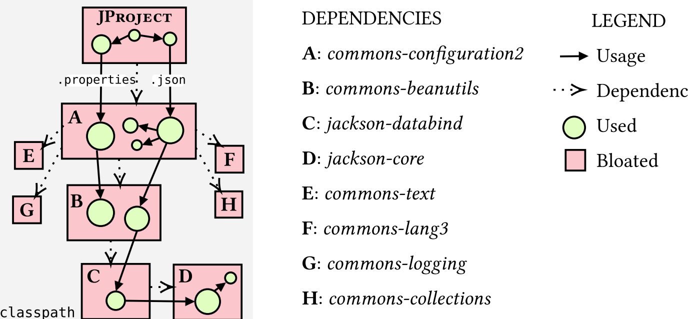



Fig. 1. Typical code reuse scenario in the Java ecosystem. The Java project, JProject, uses functionalities provided by the library *commons-configuration2*, which has seven dependencies. Rectangles, in red, represent Java artifacts. Circles inside artifacts, in green, represent API members used by JProject.

pipeline. Unlike existing Java debloating techniques [8, 26, 28, 50, 54], our approach exploits the diversity of bytecode coverage tools to collect complete coverage information through the whole dependency tree. The complete automation of the debloating procedure and our more reliable approach for collecting usage information allows us to evaluate JDBL on the largest debloating dataset up to date. Moreover, this is the first work in the debloating literature that assesses the utility of the debloated libraries with respect to their clients. In summary, the contributions of this article are the following:

— A practical, automated bytecode debloating approach for Java artifacts based on the collection of complete coverage information from multiple sources.
— An open-source tool, JDBL, which executes throughout the Maven build pipeline and automatically generates debloated versions of Java artifacts.
— The largest empirical study on software debloating was performed with 211 debloated libraries and investigated code reduction at three granularity levels.
— The first assessment of the impact of debloated third-party libraries on their clients, with 988 clients of the libraries that JDBL successfully debloats.

## 2 MOTIVATING EXAMPLE

In this section, we illustrate the impact of software bloat in the context of a Java application with dependencies. Figure 1 shows the dependency tree of a typical Java project. JProject implements a set of features and reuses functionalities provided by third-party dependencies. To illustrate the notion of software bloat, we focus on one specific functionality that JProject reuses: parsing a configuration file located in the file system, provided by the *commons-configuration2* library.[1]

In our example, JProject uses this library to read `properties` and `json` configuration files. However, *commons-configuration2* supports additional file formats, which are not necessary for JProject to run correctly, i.e., they are considered as bloat. Yet, all the classes of the library must be added to the `classpath` of JProject, as well as all the runtime dependencies of the library. The green circles and red squared components in Figure 1 highlight this phenomenon: only the API members in green are necessary for the JProject. All the code that belongs to the components in red, which includes all the functionalities for parsing other types of files than `properties` and

[1] https://commons.apache.org/proper/commons-configuration2.

`json`, are bloated with respect to JPROJECT. This represents a considerable amount of bytecode from *commons-configuration2* that is included in JPROJECT but is not needed. In addition, the dependency towards *commons-configuration2*, implies that JPROJECT has to include the classes of a total of seven transitive dependencies in its `classpath`. Some classes in the dependency **B** are used to process the file formats used by JPROJECT, and parsing `json` files requires functionalities from dependencies **C** and **D**. Notice that the classes in the dependencies **E**, **F**, **G**, and **H**, are not necessary for JPROJECT.

This example illustrates the characteristics of Java projects: they are composed of a main module and import third-party dependencies. All the code of the main module, the dependencies, and the transitive dependencies is packaged in the project's `JAR`. Also, the existence of disjoint execution paths makes Java projects susceptible to including unnecessary functionalities from third-party libraries.

In this article, we focus on debloating functionalities from compiled Java projects and their dependencies. This involves the detection and removal of the reachable bytecode instructions that do not provide any functionalities to the project at runtime, both in the project's own classes and in the classes of its dependencies. The objective of this bytecode transformation is to reduce the size of the project while still providing the same functionalities to its clients.

The main challenge for software debloating is to obtain precise usage information of the application and identify which parts can be safely removed. In the next section, we describe our approach to overcome these challenges using code coverage. We motivate our approach and introduce the technical challenges. Then, we present the details of our technique.

## 3 COVERAGE-BASED DEBLOATING

Coverage-based debloating processes two inputs: a Java project, and coverage information collected when running a specific workload on the project. Our debloating technique removes the bytecode constructs that are not necessary to run the workload correctly. It produces a valid compiled Java project as output. The debloated artifact is executable and has the same behavior as the original, w.r.t. the workload.

*Definition 1 (Coverage-based Debloating).* Let $\mathcal{P}$ be a program that contains a set of instructions $\mathcal{S}_{\mathcal{P}}$ and a workload that exercises a set $\mathcal{F}_{\mathcal{P}}$ of instructions, where $\mathcal{F}_{\mathcal{P}} \subseteq \mathcal{S}_{\mathcal{P}}$. The coverage-based debloating technique transforms $\mathcal{P}$ into a syntactically correct program $\mathcal{P}'$, where $|\mathcal{S}_{P'}| \leq |\mathcal{S}_{P}|$ and $\mathcal{P}'$ preserve the same behavior as $\mathcal{P}$ when executing the workload.

The collection of accurate coverage information is a critical task for coverage-based debloating. In the following section, we discuss some key challenges and limitations of current techniques to collect complete Java bytecode coverage information. Then, we introduce the solutions that we implement to address these technical challenges, which are part of our contributions.

### 3.1 Challenges of Collecting Accurate and Complete Coverage for Debloating

Java has a rich ecosystem of tools and algorithms to collect code coverage reports. These tools, which rely on bytecode transformations [61], perform the following three key steps: (i) the bytecode is enriched with probes at particular locations of the program's control flow, depending on the granularity level of the coverage; (ii) the instrumented bytecode is executed in order to collect the information on which probes are activated at runtime; (iii) the activated regions of the bytecode are mapped with the source code, and a coverage report is given to the user.

Existing code coverage techniques are implemented in mature, robust, and scalable tools, which can serve as the foundation for coverage-based debloating. State-of-the-art tools for this purpose

include JaCoCo,[2] JCov,[3] and Clover.[4] Yet, all of them have two essential limitations when used for debloating. First, different instrumentation strategies do not handle specific corner cases, while capturing the program's execution [32]. For example, JaCoCo does not generate a complete coverage report for fields, methods that contain only one statement that triggers an exception, and the compiler-generated methods and classes for Java enumerations. Second, by default, these tools collect coverage only for the bytecode of a compiled project and do not instrument the bytecode of third-party libraries. In the following, we discuss the corner cases for accurate coverage in detail. In Section 3.2, we present our approach to address corner cases and collect coverage information across the whole dependency tree.

Collecting code coverage involves several challenges related to source code compilation and bytecode instrumentation. First, the bytecode instrumentation must be safe and efficient, i.e., it must not alter the functional behavior of the application and have a limited runtime overhead. Second, the instrumentation must generate a coverage report that is complete, i.e., all the bytecode that is necessary to execute the workload should be reported as covered. This latter challenge is the most critical for coverage-based debloating: a single class missed in the report means that a necessary piece of bytecode will be removed, leading to an incorrect debloated application.

Three factors affect the completeness of the coverage. First, no code coverage tool currently captures the coverage information across the whole dependency tree of a Java project. This limits the effect of debloating based on code coverage to the project's sources only. Second, different tools have various instrumentation strategies to handle the variety of existing bytecode constructs [23]. Consequently, these tools provide different reports for the same build setup. Third, the Java compiler transforms the bytecode, causing information gaps between source and bytecode, e.g., by inlining constants or creating synthetic API members in certain situations [33, 52]. In this case, it is not possible for coverage tools to collect information missing in the original bytecode. The following examples illustrate five challenges that we identified:

**Challenge #1** *Implicit Exceptions Thrown From Invoked Methods.* Listing 1 shows an example of an incorrect coverage report caused by a design limitation of JaCoCo. Both methods `m1` and `m2` are executed at runtime and both should be reported as covered. Yet, `m1` (lines 2–4) is missed by JaCoCo, while it is clear that, if we remove it, the `test` in class `FooTest` fails (lines 11–15). This is because the JaCoCo probe insertion strategy does not consider implicit exceptions thrown from invoked methods.[5] These exceptions are subclasses of the classes `RuntimeException` and `Error`, and are expected to be thrown by the JVM itself at runtime. If the control flow between two probes is interrupted by an exception not explicitly created with a `throw` statement, all the instructions in between are missed by JaCoCo due to the non-existence of an instrumentation probe on the exit point of the method. In conclusion, JaCoCo misses one corner case for coverage: methods with a single-line invocation to other methods that throw exceptions.

**Challenge #2** *Implicit Methods in Enumerated Types.* Listing 2 shows an example of incorrect coverage due to the inability of JaCoCo to account for implicit methods in enumerated types. `FooEnum` is a Java enumerated type declaring the string constant `MAGIC` with the value `"forty two"` (line 2). The `test` method in the class `FooEnumTest` asserts the value of the constant in line 14. However, the implicit method `valueOf`[6] in `FooEnum` is not covered according to JaCoCo. The reason is that, in Java, every enumerated type implicitly extends the class `java.lang.Enum`, which

[2]https://www.eclemma.org/jacoco.
[3]https://github.com/openjdk/jcov.
[4]https://openclover.org.
[5]https://www.eclemma.org/jacoco/trunk/doc/flow.html.
[6]https://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html.

implements the methods `Enum.values()` and `Enum.valueOf()`. These methods are generated by the compiler, at compile-time. Therefore, they are not instrumented by coverage tools, which degrades the overall completeness of the produced coverage report.

```
1 public class Foo {
2   public void m1() {
3     m2();
4   }
5   public void m2() {
6     throw new IllegalArgumentException();
7   }
8 }
9
10 public class FooTest {
11   @Test(expected = IllegalArgumentException.class)
12   public void test() {
13     Foo foo = new Foo();
14     foo.m1();
15   }
16 }
```

Listing 1: Example of an incomplete coverage report given by JaCoCo. The method `m1` is executed when running the method `test` in `FooTest`. However, this method is not considered as covered by JaCoCo.

```
1 public enum FooEnum {
2   MAGIC("forty two");
3   public final String label;
4   FooEnum(String label) { this.label = label; }
5   public static <T extends Enum<T>>
6         T valueOf(Class<T> enumType, String name)
7         {...}
8 }
9
10 public class FooEnumTest {
11   @Test
12   public void test() {
13     assertEquals("forty two",
14           FooEnum.valueOf("MAGIC").label);
15   }
16 }
```

Listing 2: Example of an incomplete coverage result given by JaCoCo. The compiler-generated method `valueOf` in `FooEnum` is executed. However, this method is not instrumented and, therefore, is not reported as covered.

**Challenge #3** *Java Compiler Optimizations.* Listing 3 illustrates an example that is incorrectly handled by all code coverage tools based on bytecode instrumentation. The variable `MAGIC`, initialized with a final static integer literal in line 2, is used in the `FooTest` class as `Foo.MAGIC` (line 8). Therefore, the class `Foo` is necessary for the correct compilation and execution of the test method in the class `FooTest`. However, the class `Foo` is not detected as covered by JaCoCo or any other code coverage tool based on bytecode instrumentation. The cause is a bytecode optimization implemented in the `javac` compiler, which inlines constants at compilation time. This is shown in Listing 4, which is the bytecode generated after compiling the sources of the `FooTest` class from Listing 3. As we observe in lines 4–5, the value of the constant `MAGIC` is directly substituted by its integer value, and hence the reference to the class `Foo` is lost during the compilation of the source code. Note that, if we remove the class `Foo`, the program will not compile correctly.

```
1 class Foo(){
2   public static final int MAGIC = 42;
3 }
4
5 public class FooTest {
6   @Test
7   public void test() {
8     assertEquals(42, Foo.MAGIC);
9   }
10 }
```

Listing 3: Example of an inaccurate coverage report. The class `Foo` is not considered covered by any coverage tool, since the primitive constant `MAGIC` is inlined with its actual integer value by the Java compiler at compilation time.

```
1 public class org.example.FooTest {
2   public void test();
3     Code:
4       0: BIPUSH 42
5       2: BIPUSH 42
6       // Method
            junit/framework/TestCase.assertEquals:(II)V
7       4: INVOKESTATIC #3
8       7: RETURN
9 }
```

Listing 4: Excerpt of the disassembled bytecode of Listing 3. The Java compiler does not let any reference to the object `Foo` in the bytecode of the method `test` in class `FooTest`.

**Challenge #4** *Java Interfaces.* In Listing 5, the class `Foo` implements the method `doMagic` of the interface `Magic` (lines 1–3). This class will not compile correctly if its interface is removed.

However, JaCoCo does not instrument non-static methods in interfaces because they have no executable instructions. Interfaces, exceptions, enumerations, and annotations are constructs of the Java language designed to facilitate software engineering tasks and most code coverage tools do not report them as covered.

**Challenge #5** *Third-Party Dependencies.* Listing 6 presents an example of a used class from a third-party dependency that is not reported as covered by JaCoCo. The class `Foo` uses the method `byteCountToDisplaySize` from the class `FileUtils` (line 5). `FileUtils` is provided by the third-party dependency *commons-io* and imported in line 1. However, when executing JaCoCo, the classes from this third-party are not instrumented. This happens because JaCoCo is designed to cover only the project's code.

```
1 public interface Magic {
2   int doMagic();
3 }
4
5 public class Foo implements Magic {
6   @Override
7   public int doMagic() {
8     return 42;
9   }
10 }
11
12 public class FooTest {
13   @Test
14   public void test() {
15     Foo foo = new Foo();
16     assertEquals(42, foo.doMagic());
17   }
18 }
```

Listing 5: Example of an incomplete coverage result given by JaCoCo. The interface `Magic` implemented by class `Foo` is necessary for the compilation of the class but it is not covered.

```
1 import org.apache.commons.io.FileUtils;
2
3 public class Foo {
4   public String showFileSize(long fileSize) {
5     return FileUtils
6         .byteCountToDisplaySize(fileSize);
7   }
8 }
9
10 public class FooTest {
11   @Test
12   public void test() {
13     long fileSize = 50000;
14     Foo foo = new Foo();
15     assertEquals("48 KB",
16         foo.showFileSize(fileSize));
17   }
18 }
```

Listing 6: Example of an incomplete coverage result given by JaCoCo. The class `FileUtils` in the dependency *commons-io* is used but it is not covered.

## 3.2 Addressing Coverage Challenges for Debloating

This section describes our approach to tackle the bytecode coverage challenges presented in the previous section. The goal is to consolidate coverage information that can be used for debloating.

*3.2.1 Aggregating Coverage Reports.* We address the bytecode tracing challenges by aggregating the coverage reports produced by diverse coverage tools. The baseline coverage report is collected with JaCoCo. Then we consolidate this information as follows.

To handle the case of implicit exceptions, illustrated in Listing 1, we develop Yajta,[7] a customized tracing agent for Java. Yajta adds a probe at the beginning of the methods, including the default constructor. Yajta is based on Javassist[8] for bytecode instrumentation. To handle compiler-generated methods, illustrated in Listing 2, we include the reports of JCov. This pure Java implementation of code coverage is officially maintained by Oracle and used for measuring coverage in the Java platform (JDK). It maintains the version of Java which is currently under development and supports the processing of large volumes of heterogeneous workloads.

We leverage the JVM class loader to obtain the list of classes that are loaded dynamically and lead to errors discussed in Listing 3. The JVM dynamically links classes before executing

[7]https://github.com/castor-software/yajta.
[8]https://www.javassist.org.

them. The `-verbose:class` option of the JVM enables logging of class loading and unloading at runtime.

*3.2.2 Keep All Necessary Bytecodes That Cannot Be Covered.* The Java language contains specific constructs designed to achieve programming abstractions, e.g., interfaces, exceptions, enumerations, and annotations. These elements do not execute any program logic and cannot be instantiated. Therefore, they cannot be covered at runtime, and pure dynamic debloating cannot determine if they are a source of bloat. Yet, they are necessary for compilation.

To address this limitation, we always keep interfaces, enumeration types, exceptions, as well as static fields in the bytecode. This approach significantly improves the syntactic correctness of the debloated bytecode artifacts. Meanwhile, the impact on the size of the debloated code is minimal, due to the small size of such language constructs.

*3.2.3 Capturing Coverage Across the Whole Dependency Tree.* To effectively debloat a Java project, we need to analyze bytecode in the compiled project, as well as in its dependencies. To do so, we extend the coverage information provided by JaCoCo to the level of dependencies. This requires modifying the way JaCoCo interacts with Maven during the build.

We rely on the automated build infrastructure of Maven to compile the Java project and to resolve its dependencies. Maven provides dedicated plugins for fetching and storing all the dependencies of the project. Therefore, it is practical to rely on the Maven dependency management mechanisms, which are based on the *pom.xml* file that declares the direct dependencies of the project. These dependencies are `JAR` files hosted in external repositories (e.g., Maven Central [48]).[9]

Only dependencies in the runtime and compile `classpath` are packaged by Maven at the end of the build process. Therefore, we focus on dependencies with these specific scopes. Once the dependencies have been downloaded, we compile the Java sources and unpack all the bytecode of the project and its dependencies into a local directory. Then, probes are injected at the beginning and end of all Java bytecode methods of the classes in this directory. This code instrumentation is performed offline, before the workload execution and coverage collection. At runtime, the coverage tool is notified when the execution hits an injected probe. This way, our coverage-based approach captures the covered classes and methods in all dependencies.

### 3.3 Coverage-Based Debloating Procedure

In this section, we present the details of JDBL, our end-to-end tool for automated coverage-based Java bytecode debloating. JDBL receives as input a Java project that builds correctly with Maven and a workload that exercises the project. JDBL outputs a debloated, packaged project that builds correctly and preserves the functionalities necessary to run that particular workload. The debloating procedure consists of three main phases. The coverage collection phase gathers usage information based on dynamic analysis. The bytecode removal phase modifies the bytecode of the artifact, based on coverage. The artifact validation phase assesses the correctness of the debloated artifact.

Algorithm 1 details the three subroutines, corresponding to each debloating phase. In the following subsections, we describe these phases in more detail.

*3.3.1 ❶ Coverage Collection.* JDBL collects a set of coverage reports that capture the set of dependencies, classes, and methods actually used during the execution of the Java project. The coverage collection phase receives two inputs: a compilable set of Java sources, and a workload, i.e., a collection of entry-points and resources necessary to execute the compiled sources. The workload can be a set of test cases or a reproducible production workload. The coverage collection

[9] https://repo.maven.apache.org/maven2.

**ALGORITHM 1:** Coverage-based debloating procedure for a Java project.

```
Input: A correct program 𝒫 that contains a set of source files 𝒮, and declares a set of dependencies 𝒟.
Input:  A workload 𝒲 that exercises at least one functionality in 𝒫.
Output: A correct version of 𝒫, called 𝒫′, which is smaller than 𝒫 and contains the necessary code to
        execute 𝒲 and obtain the same results as with 𝒫.
   // ❶ Coverage collection phase
 1 CP ← compileSources(𝒮, 𝒫) ∪ getDependencies(𝒟, 𝒫);
 2 INST ← instrument(CP);
 3 USG ← ∅;
 4 foreach w ∈ 𝒲 do
 5     execute(w, INST);
 6     foreach class ∈ INST do
 7         if isExecuted(class) then
 8             USG ← addKey(class, USG);
 9             foreach method ∈ class do
10                 if isExecuted(method) then
11                     USG ← addVal(method, class, USG);
   // ❷ Bytecode removal phase
12 foreach class ∈ CP do
13     if class ∉ keys(USG) then
14         CP ← CP \ class;
15     else
16         foreach method ∈ class do
17             if method ∉ values(class, USG) then
18                 CP ← CP \ method;
   // ❸ Artifact validation phase
19 OBS ← execute(𝒲, 𝒫);
20 if !buildSuccess(CP) | execute(𝒲, CP) ≠ OBS then
21     return ALERT;
22 𝒫′ ← package(CP);
23 return 𝒫′;
```

phase outputs the original, unmodified, bytecode and a set of coverage reports that account for the minimal set of classes and methods required to execute the workload.

Lines 1 to 11 in Algorithm 1 show this procedure. It starts with the compilation of the input project $\mathcal{P}$, resolving all its direct and transitive dependencies $\mathcal{D}$, and adding the bytecode to the `classpath` $CP$ of the project (line 1). Then, the whole bytecode contained in $CP$ (line 2) is instrumented, and a data store is initialized to collect the classes and methods used when executing the workload $\mathcal{W}$ (line 3). JDBL executes the instrumented bytecode with $\mathcal{W}$, and the classes and methods used are saved (lines 8 and 11). JDBL considers $\mathcal{W}$ to be the complete test suite of a Maven project, where each $w \in \mathcal{W}$ is an individual unit test executed by Maven.

*3.3.2 ❷ Bytecode Removal.* The goal of the bytecode removal phase is to eliminate the methods, classes, and dependencies that are not used when running the project with the workload $\mathcal{W}$. This procedure is based on the coverage information collected during the coverage collection phase. The unused bytecode instructions are removed in two passes (lines 12–18 in Algorithm 1).

First, the unused class files and dependencies are directly removed from the `classpath` of the project (lines 14 and 18). Then, the procedure analyzes the bytecode of the classes that are covered. When it encounters a method that is not covered, the body of the method is replaced to throw an `UsupportedOperationException`. We choose to throw an exception instead of removing the entire method to avoid JVM validation errors caused by the nonexistence of methods that are implementations of interfaces and abstract classes.

At the end of this phase, JDBL has removed the bloated methods, classes, and dependencies. A method is considered bloated if it is not invoked while running the workload. A class is considered bloated if it has not been instantiated or called via reflection and none of its fields or methods are used. A third-party dependency is considered bloated if none of its classes or methods are used when executing the project with a given workload.[10]

*3.3.3 ❸ Artifact Validation.* The goal of the artifact validation phase is to assess the syntactic and semantic correctness of the debloated artifact with respect to the workload provided as input. This is how we detect errors introduced by the bytecode removal, before packaging the debloated `JAR`.

To assess syntactic correctness, we verify the integrity of the bytecode in the debloated version. This implies checking the validity of the bytecode that the JVM has to load at runtime, and also checking that no dependencies or other resources were incorrectly removed from the `classpath` of the Maven project. We reuse the Maven tool stack, which includes several validation checks at each step of the build process [34]. For example, Maven verifies the correctness of the *pom.xml* file, and the integrity of the produced `JAR` at the last step of the build life cycle. To assess semantic correctness, we check that the debloated project executes correctly with the workload.

Algorithm 1 (lines 19–23) details this last phase of coverage-based debloating. We run the original version of $\mathcal{P}$ with the workload $\mathcal{W}$, to collect the program's original outputs in the variable *OBS* (line 19). Then, the algorithm performs two checks in line 20: (1) a syntactic check that passes if the build of the debloated program is successful; and (2) a behavioral check that passes if the debloated program produces the same output as $\mathcal{P}$, with $\mathcal{W}$. In other words, it treats *OBS* as an oracle to check that the debloated project preserves the behavior of $\mathcal{P}$. Finally, the debloated artifact is packaged and returned in line 23.

*3.3.4 Implementation Details.* The core implementation of JDBL consists in the orchestration of mature code coverage tools and bytecode transformation techniques. The coverage-based debloating process is integrated into the different Maven building phases. We focus on Maven as it is one of the most widely adopted build automation tools for Java artifacts. It provides an open-source framework with the APIs required to resolve dependencies automatically and to orchestrate all the debloating phases during the project build.

JDBL gathers direct and transitive dependencies by using the `maven-dependency`[11] plugin with the `copy-dependencies` goal. This allows us to manipulate the project's `classpath` in order to extend code coverage tools at the level of dependencies, as explained in Section 3.2.3. For bytecode analysis, the collection of non-removable classes, and the whole bytecode removal phase, we rely on ASM,[12] a lightweight, and mature Java bytecode manipulation and analysis framework. The instrumentation of methods and the insertion of probes are performed by integrating JaCoCo, JCov, Yajta, and the JVM class loader within the Maven build pipeline, as described in Section 3.2.1.

[10]In this work, we refer to Maven dependencies.
[11]https://maven.apache.org/plugins/maven-dependency-plugin.
[12]https://asm.ow2.io.

JDBL is implemented as a multi-module Maven project with a total of 5$K$ lines of code written in Java. JDBL is designed to debloat single-module Maven projects. It can be used as a Maven plugin that executes during the *package* Maven phase. Thus, JDBL is designed with usability in mind: it can be easily invoked within the Maven build life-cycle and executed automatically, no additional configuration or further intervention from the user is needed. To use JDBL, developers only need to add the Maven plugin within the build tags of the *pom.xml* file. The source code of JDBL is publicly available on GitHub, with binaries published in Maven Central. More information on JDBL is available at https://github.com/castor-software/jdbl.

## 4 EMPIRICAL STUDY

In this section, we present our research questions, describe our experimental methodology, and the set of Java libraries utilized as study subjects.

### 4.1 Research Questions

To evaluate our coverage-based debloating approach, we study its *correctness*, *effectiveness*, and *impact*. We assess the debloating results through four different validation layers: compilation and testing of the debloated Java libraries, and compilation and testing of their clients. Our study is guided by the following research questions:

**RQ1:** *To what extent can a generic, fully automated coverage-based debloating technique produce a debloated version of Java libraries?*

**RQ2:** *To what extent do the debloated library versions preserve their original behavior w.r.t. the debloating workload?*

RQ1 and RQ2 focus on assessing the *correctness* of our approach. In RQ1, we assess the ability of JDBL at producing a valid debloated JAR for real-world Java projects. With RQ2, we analyze the behavioral correctness of the debloated artifacts.

**RQ3:** *How much bytecode is removed in the compiled libraries and their dependencies?*

**RQ4:** *What is the impact of using the coverage-based debloating approach on the size of the packaged artifacts?*

**RQ5:** *How does coverage-based debloating compare with the state-of-the-art of Java debloating regarding the size of the packaged artifacts and behavior preservation?*

RQ3, RQ4, and RQ5 investigate the *effectiveness* of our debloating procedure in producing a smaller artifact by removing the unnecessary bytecode. We measure this effectiveness with respect to the amount of debloated methods, classes, and dependencies, as well as with the reduction of the size of the bundled JAR files.

**RQ6:** *To what extent do the clients of debloated libraries compile successfully?*

**RQ7:** *To what extent do the clients behave correctly when using a debloated library?*

In RQ6 and RQ7, we go one step further than any previous work on software debloating and investigate how coverage-based debloating of Java libraries impacts the clients of these libraries. Our goal is to determine the ability of dynamic analysis via coverage at capturing the behaviors that are relevant for the users of the debloated libraries.

### 4.2 Data Collection

We have extracted a dataset of open-source Maven Java projects from GitHub, which we use to answer our research questions. We choose open-source projects because accessing closed-source software for research purposes is a difficult task. Moreover, the diversity of open-source software

Table 1. Descriptive Statistics of the Dataset of Libraries and their Associated Clients

| | | Min | 1st Qu. | Median | 3rd Qu. | Max | Avg. | Total |
|---|---|---|---|---|---|---|---|---|
| 94 Libraries | # Versions | 1 | 1 | 3.0 | 5.0 | 23.0 | 4.2 | 395 |
| | # Tests | 1 | 139.8 | 378.0 | 1,108.2 | 24,946 | 1,830.6 | 713,932 |
| | # LOC | 132 | 5,439.5 | 17,935.5 | 47,866.0 | 341,429 | 35,629.6 | 10,831,394 |
| | Total Class Coverage | 2.7 % | 74.6 % | 94.0 % | 99.0 % | 100.0 % | 84.8 % | N.A |
| 2,874 Clients | # Tests | 1 | 4.5 | 20.0 | 74.0 | 11,415 | 107.7 | 211,116 |
| | # LOC | 0 | 3,130.0 | 9,170.0 | 58,990.0 | 4,531,710 | 72,897.1 | 140,910,102 |
| | JaCoCo Coverage | 0.0 % | 2.1 % | 20.24 % | 57.7 % | 100.0 % | 31.4 % | N.A |

allows us to determine if our coverage-based debloating approach generalizes to a vast and rich ecosystem of Java projects.

The dataset is divided into two parts: a set of libraries, i.e., Java projects that are declared as a dependency by other Java projects, and a set of clients, i.e., Java projects that use the libraries from the first set. The construction of this dataset is performed in five steps:

(1) We identify the 147,991 Java projects on GitHub that have at least five stars. We use the number of stars as an indicator of interest [7].
(2) We select the 34,560 (23.4 %) Maven projects that are single-module. We focus on single-module projects because they generate a single `JAR`. For this, we consider the projects that have a single Maven build configuration file (i.e., *pom.xml*).
(3) We ignore the projects that do not declare JUnit as a testing framework, and we exclude the projects that do not declare a fixed release, e.g., `LAST-RELEASE`, `SNAPSHOT`. We identify 155 (0.4 %) libraries, and 25,557 (73.9 %) clients that use 2,103 versions of the libraries.
(4) We identify the commit associated with the version of the libraries, e.g., `commons-net:3.4` is defined in the commit SHA: 74a2282. For this step, we download all the revisions of the *pom.xml* files to identify the commit for which the release has been declared. We successfully identified the commit for 1,026/2,103 (48.8 %) versions of the libraries. 143/155 (92.3 %) libraries and 16,964/25,557 (66.4 %) clients are considered.
(5) We execute three times the test suite of all the library versions and all clients, as a sanity check to filter out libraries with flaky tests. We keep the libraries and clients that have at least one test and have all the tests passing: 94/143 (65.7 %) libraries, 395/1,026 (38.5 %) library versions, and 2,874/16,964 (16.9 %) clients passed this verification. From now on, we consider each library version as a unique library to improve the clarity of this article.

Table 1 summarizes the descriptive statistics of the dataset. The total class coverage of the libraries is computed based on the aggregation of the coverage reports of the tools presented in Section 3.2.1. The number of LOC and the coverage of the clients are computed with JaCoCo. In total, our dataset includes 395 Java libraries from 94 different repositories and 2,874 clients. The 395 libraries include 713,932 test cases that cover 80.83 % of the 10,831,394 LOC. One library in our dataset can generate fake Pokemons [13]. The clients have 211,116 test cases that cover 20.24 % of the 140,910,102 LOC. The dataset is described in detail in Durieux et al. [14].

### 4.3 Experimental Protocol

In this section, we introduce the experimental protocol that we use to answer our research questions. The goal is to examine the ability of JDBL to debloat Java projects configured to build with Maven.

For our experiments, we use the test suite of the projects as a workload. Test suites are widely available while obtaining a realistic workload for hundreds of libraries is extremely difficult.

Another motivation is to integrate JDBL in the build process and deploy the debloated version, which can then be directly used by the clients.

We experiment coverage-based debloating on 395 different versions of 94 libraries. An original step in our experimental protocol consists of further validating the utility of the debloated libraries with respect to their clients. This way, we check if coverage-based debloating preserves the elements that are required to compile and successfully run the test suites of the clients.

*4.3.1 Coverage-based Debloating Execution.* To run JDBL at scale, we created an execution framework that automates the execution of our experimental pipeline. The framework orchestrates the execution of JDBL and the collection of data to answer our research questions. As JDBL is implemented as a Maven plugin, most of the steps rely on the Maven build life cycle.

The execution of JDBL is composed of three main steps:

(1) *Compile and test the original library.* We build the original library (i.e., using `mvn package`) to ensure that it builds correctly and that all its test cases pass. We configure the project to generate a JAR file that contains all the binaries of the project. This change of configuration may be in conflict with the original project build configuration and therefore fail in some scenarios. At the end of the execution of the test suite, a fat JAR file is produced, which contains the bytecode of the library and all its dependencies. We also store the reports concerning test execution and the corresponding logs. The data produced during this step is used as a reference for further comparison with respect to the debloated version of the library, in RQ1 and RQ2.
(2) *Configure the library to run* JDBL. The second step injects JDBL as a plugin inside the Maven configuration (*pom.xml*) and resets the configuration of the `maven-surefire-plugin` for our experiments.[13] This reset ensures that its original configuration is not in conflict with the execution of the coverage collection phase of JDBL. A manual configuration of JDBL could prevent this problem. Yet, we decided to standardize the execution for all the libraries in order to scale up and automate the evaluation.
(3) *Execute* JDBL. The third of our experiment framework executes JDBL on the library, i.e., it runs `mvn package` with JDBL configured in the *pom.xml*. At the end of this step, we collect the report generated by JDBL with information about the debloated JAR (for RQ1, RQ3, and RQ4), the coverage report, and the test execution report (for RQ2).

The execution was performed on a workstation running Ubuntu Server with an `i9-10900K` CPU (16 cores) and 64 GB of RAM. We set a maximum timeout constraint of 1:00:00 per project, which allows scaling up our experiments without an excessive debloating time. It took 4 days, 8:39:09 to execute the complete JDBL experiment on our dataset, and 1 day, 10:55:04 to only debloat the libraries. Each debloating execution is performed inside a Docker image in order to eliminate any potential side effects. The Docker image that we used during our experiment is available on DockerHub: tdurieux/jdbl which uses JDBL commit SHA: c57396a. The execution framework is publicly available on GitHub [47], and the raw data obtained from the complete execution is available on Zenodo: 10.5281/zenodo.3975515. The JDBL execution framework is composed of $3K$ lines of Python code.

*4.3.2 Debloating Correctness (RQ1 and RQ2).* To answer RQ1 and RQ2, we run JDBL on each of the 395 versions of 94 libraries. RQ1 assesses the ability of JDBL to produce a debloated JAR file, i.e., to successfully build the debloated Maven project. For RQ2, we analyze whether the test suite of the library has the same behavior before and after debloating.

Figure 2 illustrates the pipeline of RQ1 and RQ2. First, we check that the library compiles correctly before the debloat. If it does, then we verify if JDBL has generated a JAR (RQ1). If no JAR

[13] https://maven.apache.org/surefire/maven-surefire-plugin.

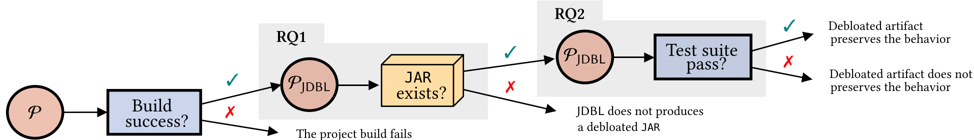



Fig. 2. Pipeline of our experimental protocol to answer RQ1 and RQ2.

file is generated, then the debloating is considered as failed and the library is excluded for the rest of the evaluation. The last step verifies that the test suite behaves the same before and after the bytecode removal phase. This approach is consistent with previous works [8, 41] in which existing tests are executed, and the results are used as a proxy for semantic preservation.

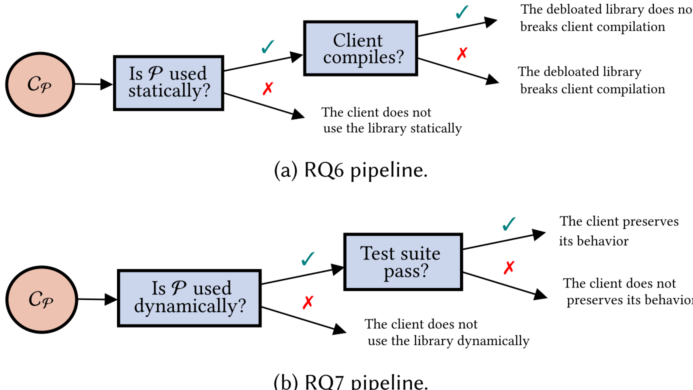
We compare the test execution reports produced during the first step of the JDBL execution (see Section 4.3.1) and the test report generated during the verification step of JDBL. We consider that the test suite has the same behavior on both versions if the number of executed tests is the same for both versions, and if the number of passing tests is also the same. The number of executed tests might vary between the two versions because we modify the `maven-surefire-plugin` configuration to run as default in order to standardize and scale our experiments. If the number of passing tests is not the same between the two reports, JDBL is considered as having failed and the libraries are excluded for the rest of the evaluation. We manually analyze the execution logs of the failing debloating executions to understand what happened.

*4.3.3 Debloating Effectiveness (RQ3, RQ4, and RQ5).* We assess the effectiveness of JDBL regarding two different aspects. The first aspect is related to code removal, checking the number of classes, and methods that are debloated. The second aspect is the size on disk that JDBL allows saving by removing unnecessary parts of the libraries.

To answer RQ3, RQ4, and RQ5, we use the debloating reports of the original and debloated JAR files. These reports contain the list of all the methods and classes of the libraries (including the dependencies), and if the element was debloated or not. For RQ3, we compute the ratio of methods and classes that are debloated. For RQ4, we extract the original and debloated JAR, and we compare the size in bytes of all the extracted files. To answer RQ5, we compare the bytecode size reduction and the test results after debloating with JDBL and with JShrink. JShrink is the most recent tool for debloating Java bytecode applications using dynamic analysis. The source code of JShrink is publicly available, and its debloating capabilities for a benchmark of Java projects are presented in its companion research article [8].

For RQ3 and RQ4, we consider the 211 library versions that successfully pass the debloating correctness assessment. We separate the 141/211 (66.8 %) libraries that do not have dependencies and the 70/211 (33.2 %) libraries that have at least one dependency. We decided to do so because we observed that the libraries that have dependencies contain many more elements (bytecode and resources), which may negatively impact the analysis compared to libraries that do not have a dependency. For RQ5, we consider 17 Java projects in the original benchmark used to evaluate JShrink and compare JDBL against the debloating results reported in the JShrink article [8].

*4.3.4 Debloating Impact on Clients (RQ6 and RQ7).* In the two final research questions, we analyze the impact of debloating Java libraries on their clients. This analysis is relevant since we are debloating libraries that are mostly designed to be used by clients. This analysis also provides further information on the validity of this approach. As far as we know, this is the first time that a software debloating technique is validated with the clients of the debloated artifacts. We perform debloating validation from the clients' side at two layers: client's compilation and client's testing.




(a) RQ6 pipeline.

(b) RQ7 pipeline.

Fig. 3. Pipelines of our experimental protocol to answer RQ6 and RQ7.

For RQ6, we verify that the clients still compile when the original library is replaced by its debloated version. We check that JDBL does not remove classes or methods in libraries that are necessary for the compilation of their client. Figure 3(a) illustrates the pipeline for this research question. First, we check that the client $C_{\mathcal{P}}$ uses the library statically in the source code. To do so, we analyze the source code of the clients. If there is at least one element from the library present in the source code of a client, then we consider the library as statically used by the client.

If the library is used, we inject the debloated library and build the client again. If the client successfully compiles, we conclude that JDBL debloated the library while preserving the useful parts of the code that are required for compilation.

A debloated library stored on a disk is of little use compared to a debloated library that provides the behavior expected by its clients. Therefore, with RQ7 we wish to determine if JDBL preserves the functionalities that are necessary for the clients. Figure 3(b) illustrates the pipeline for this research question. First, we execute the test suite of the client $C_{\mathcal{P}}$ with the original version of the library. We check that the library is covered by at least one test of the client. If this is true, we replace the library with the debloated version and execute the test suite again. If the test suite behaves the same as with the original library, we conclude that JDBL is able to preserve the functionalities that are relevant to the clients.

To ensure the validity of this protocol, we perform additional checks on the clients. All the clients have to use at least one of the 211 debloated libraries. We only consider the 988/1,354 (73.0 %) clients that either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the library (static or dynamic usage). The 988 clients that statically use the library serve as the study subjects to answer RQ6. The 281/988 (28.4 %) clients that have at least a test that reaches the debloated library serve as the study subjects to answer RQ7.

## 5 RESULTS

We present our experimental results on the correctness, effectiveness, and impact of coverage-based debloating for automatically removing unnecessary bytecode from Java projects.

### 5.1 Debloating Correctness (RQ1 and RQ2)

In this section, we report on the successes and failures of JDBL to produce a correct debloated version of Java libraries.

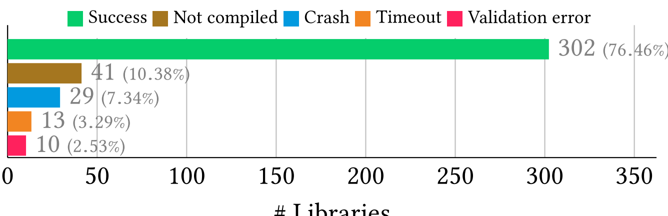



Fig. 4. Number of libraries for which JDBL succeeds or fails to produce a debloated JAR file.

*5.1.1 RQ1. To what extent can a generic, fully automated coverage-based debloating technique produce a debloated version of Java libraries?* In the first research question, we evaluate the ability of JDBL at performing automatic coverage-based debloating for the 395 libraries in our initial dataset. Here, we consider the debloating procedure to be successful if JDBL produces a valid debloated JAR file for a library. To reach this successful state, the project to be debloated must go through all the build phases of the Maven build life-cycle, i.e., compilation, testing, and packaging, according to the protocol described in Section 4.3.2.

Figure 4 shows a bar plot of the number of successfully debloated libraries. It also displays the number of cases where JDBL does not produce a debloated JAR file, due to failures in the build.

For the 395 libraries of our dataset, JDBL succeeds in producing a debloated JAR file for a total of 302 libraries and fails to debloat 93 libraries. Therefore, the overall debloating success rate of JDBL is 76.5 %. When considering only the libraries that were originally compiled, JDBL succeeds in debloating 85.3 % of the libraries. We manually identify and classify the causes of failures in four categories:

- *Not compiled.* As a sanity-check, we compile the project before injecting JDBL in its Maven build. The only modification consists in changing the *pom.xml* to request the generation of a JAR that contains the bytecode of the project, along with all its runtime dependencies. If this step fails, the project does not compile, and it is ignored for the rest of the evaluation.
- *Crash.* We run a second Maven build, with JDBL. This modifies the bytecode to remove unnecessary code. In certain situations, this procedure causes the build to stop at some phase and terminate abruptly, i.e., due to accessing invalid memory addresses, using an illegal opcode, or triggering an unhandled exception.
- *Time-out.* JDBL utilizes various coverage tools that instrument the bytecode of the project and its dependencies. This process induces an additional overhead in the Maven build process. Moreover, the incorrect instrumentation with at least one of the coverage tools may cause the test to enter into an infinite loop, e.g., due to blocking operations.
- *Validation error.* Maven includes dedicated plugins to check the integrity of the produced JAR file. JDBL alters the behavior of the project build by packaging the debloated JAR using the `maven-assembly-plugin`. Some other plugins may not be compatible with JDBL (e.g., when using customized assemblies), triggering validation errors during the build life-cycle. Moreover, we observe that for some libraries, the tests in the debloated JAR are not correctly executed due to particular library configurations in the `maven-surefire-plugin`.

We manually investigate the causes of the validation errors for the 10 libraries that fall into this category. We found that Maven fails to validate the execution of the tests, either due to errors when running the instrumented code to collect coverage or incompatibilities among plugins that exercise the instrumented version of the library. For example, in the case of *org.apache.commons:collection:4.0*, the `MANIFEST.MF` file is missing in the debloated JAR due to an incompatibility with

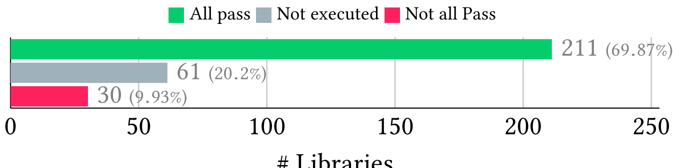



Fig. 5. Number of debloated libraries for which the test suite passes; number of debloated libraries for which the number of executed tests does not match the original test execution (ignored for the research question); number of debloated libraries that have at least one failing test case.

library plugins. Therefore, Maven fails to package the debloated bytecode. As another example, the Maven build of *org.yaml:snakeyaml:1.17* fails because of Yajta's instrumentation. This tool relies on Javassist for inserting probes in the bytecode. In this case, JDBL changes a class that was frozen by Javassist when it was loaded. Consequently, Javassist crashes because further changes in a frozen class are prohibited.[14]

**Answer to RQ1:** JDBL successfully produces a debloated JAR file for 302 libraries in our dataset, which represents 85.3 % of the libraries that compile correctly. This is the largest number of debloated subjects in the literature.

*5.1.2 RQ2. To what extent do the debloated library versions preserve their original behavior w.r.t. the debloating workload?* Our second research question evaluates the behavior of the debloated library with respect to its original version. This evaluation is based on the test suite of the project. We investigate if the code debloated by JDBL affects the results of the tests of the 302 libraries for which JDBL produces a valid JAR file. This behavioral correctness assessment corresponds to the last phase in the execution of JDBL.

Figure 5 summarizes the comparison between the test suite executed on the original and the debloated libraries. From the 302 successfully debloated, 211 (69.9 %) preserve the original behavior (i.e., all the tests pass). In the case of 30 (9.9 %) libraries, we observe at least one test failure. This high test success rate is a fundamental result to ensure that the debloated version of the artifact preserves the behavior of the library. A table with the full list of the 211 successfully debloated libraries that pass all the tests is available in the replication package of this article.[15]

We excluded 61 (20.2 %) libraries because the numbers of executed tests before and after the debloating did not match. This is due to changes in the tests' configuration after injecting JDBL into the build of the libraries. We excluded those libraries since different numbers of test runs imply a different test-based specification for the original and the debloated version of the library. Consequently, the results of the tests do not provide a sound basis for behavioral comparison. The manual configuration of the libraries is a solution to handle this problem (expected usage of JDBL), yet it is impractical in our experiments because of the large number of libraries that we debloat.

In total, we execute 342,835 unique tests, from which 341,430 pass and 1,405 do not pass (973 fail, and 432 result in an error). This represents an overall behavior preservation ratio of 99.59 %, considering the total number of tests. This result shows that our code-coverage debloating approach is able to capture most of the project behavior, as observed by the tests while removing the unnecessary bytecode.

[14] https://www.javassist.org/tutorial/tutorial.html.
[15] https://github.com/castor-software/jdbl-experiments/blob/master/list_of_libs_succesfully_debloated_with_jdbl.md.

We investigate the causes of test failures in the 30 libraries that have at least one failure. To do so, we manually analyze the logs of the tests, as reported by Maven. We find the following five causes:

— `NoClassDefFound (NCDF)`: JDBL mistakenly removes a necessary class.
— `TestAssertionFailure (TAF)`: the asserting conditions in the test fail for multiple reasons, e.g., flaky tests, or test configuration errors.
— `UnsupportedOperationException (UOE)`: JDBL mistakenly modifies the body of a necessary method, removing bytecode used by the test suite.
— `NullPointerException (NPE)`: a necessary object is referenced before being instantiated.
— `Other`: The tests are failing for another reason than the ones previously mentioned.

Table 2 categorizes the test failures for the 30 libraries with at least one test that does not pass. They are sorted in descending order according to the percentage of tests that fail on the debloated version. The first column shows the name and version of the library. Columns 2–7 represent the five causes of test failure according to our manual analysis of the tests' logs: `TAF`, `UOE`, `NPE`, `NCDF`, and `Other`. The column labeled as `Other` shows the number of test failures that we were not able to classify. The last column shows the percentage of tests that do not pass with respect to the total number of tests in each library. For example, *equalsverifier:3.4.1* has the largest number of test failures. After debloating, we observe 605 test failures out of 921 tests (283 `TAF`, 221 `NCDF`, and 1 `Other`). These test failures represent 65.7 % of the total number of tests in *equalsverifier:3.4.1.* This is an exceptional case, as for most of the debloated libraries, the tests that do not pass represent less than 5 % of the total.

The most common cause of test failure is `NCDF` (735), followed by `TAF` (592). We found that these two types of failures are related to each other: when the test uses a non-covered class, the log shows a `NCDF`, and the test assertion fails consequently. We notice that `NCDF` and `UOE` are directly related to the removal procedure during the debloating procedure, meaning that JDBL is removing necessary classes and methods, respectively. This occurs because there are some Java constructs that JDBL does not manage to cover dynamically, causing an incomplete debloating result, despite the union of information gathered from different coverage tools. Primitive constants, custom exceptions, and single-instruction methods are typical examples. These are ubiquitous components of the Java language, which are meant to support robust object-oriented software design, with little or no procedural logic. They are important for humans and they are useless for the machine to run the program. Consequently, they are not part of the executable code in the bytecode, and cannot be covered dynamically.

JDBL can generate a debloated program that breaks a few test cases. These cases reveal some limitations of JDBL concerning behavior preservation, i.e., it fails to cover some classes and methods, removing necessary bytecode. One of the explanations is that the coverage tools modify the bytecode of the libraries. Those modifications can cause some test failures. A failing test case stops the execution of the test and can introduce a truncated coverage report of the execution. Since some code is not executed after the failing assertion, some required classes or methods will not be covered and therefore debloated by JDBL. For example, in the *reflections* library, a library that provides a simplified reflection API, some tests verify the number of fields of a class extracted by the library. However, JaCoCo injects a field in each class, which will invalidate the asserts of *reflections* tests.

More generally, this reveals the remaining challenges of coverage-based debloating for real-world Java applications when using the test suite as a workload. For this study, handling these challenging cases to achieve 100 % correctness requires significant engineering effort, providing only marginal insights. Therefore, we recommend always using our validation approach to be safe of semantic alterations when performing aggressive debloating transformations.

Table 2. Classification of the Tests that Fail for the 30 Libraries that do not Pass All the Tests

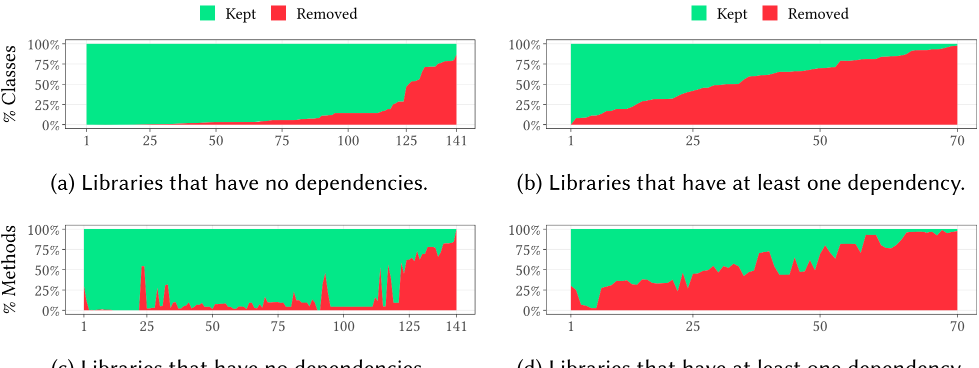
| Library | TAF | UOE | NPE | NCDF | Other | Test failures |
|---|---|---|---|---|---|---|
| *jai-imageio-core:1.3.1* | | | | 3 | | 3/3 (100.0 %) |
| *jai-imageio-core:1.3.0* | | | | 3 | | 3/3 (100.0 %) |
| *reflectasm:1.11.7* | 3 | | | 11 | | 14/16 (87.5 %) |
| *equalsverifier:3.3* | 273 | | | 315 | 1 | 589/894 (65.9 %) |
| *equalsverifier:3.4.1* | 283 | | | 321 | 1 | 605/921 (65.7 %) |
| *spark:2.0.0* | 3 | | | | 22 | 25/57 (43.9 %) |
| *logstash-logback-encoder:6.2* | | | | 74 | 36 | 110/307 (35.8 %) |
| *reflections:0.9.9* | 3 | | | | | 3/63 (4.8 %) |
| *reflections:0.9.10* | 3 | | | | | 3/64 (4.7 %) |
| *reflections:0.9.12* | 3 | | | | | 3/66 (4.5 %) |
| *reflections:0.9.11* | 3 | | | | | 3/69 (4.3 %) |
| *commons-jexl:2.0.1* | 6 | | 2 | | | 8/223 (3.6 %) |
| *commons-jexl:2.1.1* | 5 | | 3 | | | 8/275 (2.9 %) |
| *jackson-dataformat-csv:2.7.3* | | 3 | | | | 3/129 (2.3 %) |
| *sslr-squid-bridge:2.7.0.377* | | 1 | | | | 1/43 (2.3 %) |
| *jline2:2.14.3* | 2 | | | | | 2/141 (1.4 %) |
| *jackson-annotations:2.7.5* | | 1 | | | | 1/77 (1.3 %) |
| *commons-bcel:6.0* | 1 | | | | | 1/103 (1.0 %) |
| *commons-bcel:6.2* | 1 | | | | | 1/107 (0.9 %) |
| *commons-compress:1.12* | 2 | 1 | | 2 | | 5/577 (0.9 %) |
| *commons-net:3.4* | | | | 2 | | 2/271 (0.7 %) |
| *commons-net:3.5* | | | | 2 | | 2/274 (0.7 %) |
| *jline2:2.13* | 1 | | | | | 1/141 (0.7 %) |
| *commons-net:3.6* | | | | 2 | | 2/283 (0.7 %) |
| *kryo-serializers:0.43* | | 1 | | | | 1/660 (0.2 %) |
| *jongo:1.3.0* | | | | | 1 | 1/551 (0.2 %) |
| *commons-codec:1.9* | | 1 | | | | 1/616 (0.2 %) |
| *commons-codec:1.10* | | 1 | | | | 1/662 (0.2 %) |
| *commons-codec:1.11* | | 1 | | | | 1/875 (0.1 %) |
| *commons-codec:1.12* | | 1 | | | | 1/903 (0.1 %) |
| Total | 592 | 11 | 5 | 735 | 61 | 1,405 (15.0 %) |

We identified five causes of failures through the manual inspection of the Maven build testing logs: TestAssertionFailure (`TAF`), UnsupportedOperationException (`UOE`), NullPointerException (`NPE`), NoClassDefFound (`NCDF`), and other unknown causes (`Other`).

**Answer to RQ2:** JDBL automatically generates a debloated `JAR` that preserves the original behavior of 211 (69.9 %) libraries. A total of 341,430 (99.59 %) tests pass on 241 libraries. This behavioral assessment of coverage-based debloating demonstrates that JDBL preserves a large majority of the libraries' behavior, which is essential to meet the expectations of the libraries' users.

## 5.2 Debloating Effectiveness (RQ3, RQ4, and RQ5)

In this section, we report on the effects of debloating Java libraries with JDBL in terms of bytecode size reduction.




(a) Libraries that have no dependencies.

(b) Libraries that have at least one dependency.

(c) Libraries that have no dependencies.

(d) Libraries that have at least one dependency.

Fig. 6. Percentage of classes kept and removed in (a) libraries that have no dependencies, and (b) libraries that have at least one dependency. Percentage of methods kept and removed in (c) libraries that have no dependencies, and (d) libraries that have at least one dependency.

*5.2.1 RQ3. How much bytecode is removed in the compiled libraries and their dependencies?* To answer our third research question, we compare the status (kept or removed) of dependencies, classes, and methods in the 211 libraries correctly debloated with JDBL. The goal is to evaluate the effectiveness of JDBL to remove these bytecode elements through coverage-based debloating.

Figure 6 shows area charts representing the distribution of kept and removed classes and methods in the 211 correctly debloated libraries. To analyze the impact of dependencies, we separate the libraries into two sets: the libraries that have no dependency (Figure 6(a) and (c)), and the libraries that have at least one dependency (Figure 6(b) and (d)). In each figure, the $x$-axis represents the libraries in the set, sorted in increasing order according to the number of removed classes, whereas the $y$-axis represents the percentage of classes (Figure 6(a) and (b)) or methods (Figure 6(c) and (d)) kept and removed. The order of the libraries, on the $x$-axis, is the same for each figure.

Figure 6(a) shows the comparison between the percentages of kept and removed classes in the 141 libraries that have no dependency. A total of 116 libraries have at least one removed class. The library with the largest percentage of removed classes is *jfree-jcommon:1.0.23* with the 86.7 % of its classes considered as bloated. On the other hand, Figure 6(b) shows the percentage of removed classes for the 70 libraries that have at least one dependency. We observe that the ratio of removed classes in these libraries is significantly higher with respect to the libraries with no dependencies. All the libraries that have dependencies have at least one removed class, and 45 libraries have more than 50 % of their classes bloated. This result hints at the importance of reducing the number of dependencies to mitigate software bloat.

Figure 6(c) shows the percentage of kept and removed methods in the 141 libraries that have no dependencies. We observe that libraries with a few removed classes still contain a significant percentage of removed methods. For example, the library *net.iharder:base64:2.3.9* has 42.2 % of its methods removed in the 99.4 % of its kept classes. This suggests that a fine-grained debloat, to the level of methods, is beneficial for some libraries. The used classes may still contain a significant number of bloated methods. On the other hand, Figure 6(d) shows the percentage of kept methods in libraries with at least one dependency. All the libraries have a significant percentage of removed methods. As more bloated classes are in the dependencies, the artifact globally includes more bloated methods.

Now we focus on determining the difference between the bloat that is caused exclusively by the classes in the library, and the bloat that is a consequence of software reuse through the declaration

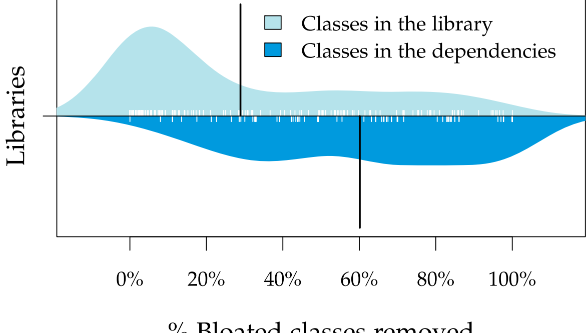



Fig. 7. Distribution of the percentage of bloated classes that belong to libraries, and bloated classes that belong to dependencies. The strip chart (white marks) in between represents the libraries that belong to each of the two groups. The two vertical bars represent the average value for each group.

Table 3. Summary of the Number of Dependencies, Classes, and Methods Removed in the 211 Libraries Correctly Debloated with JDBL

| | Removed (%) |
|---|---|
| Dependencies | 38/187 (20.3 %) |
| Classes | 61,929/103,032 (60.1 %) |
| Methods | 411,997/693,703 (59.4 %) |

of dependencies. Figure 7 shows a beanplot [29] comparing the distribution of the percentage of bloated classes in libraries, with respect to the bloated classes in dependencies. The density shape at the top of the plot shows the distribution of the percentage of bloated classes that belong to the 211 libraries. The density shape at the bottom shows this percentage for the classes in the dependencies of the 70 libraries that have at least one dependency. The average bloat in libraries is 27.3 %, whereas in the dependencies it is 59.8 %. Overall, the average of bloated classes removed for all the libraries, including their dependencies, is 32.5 %. We perform a two-samples Wilcoxon test, which confirms that there are significant differences between the percentage of bloated classes in the two groups (p-value $< 0.01$). Therefore, we reject the null hypothesis and confirm that the ratio of bloated classes is more significant among the dependencies than among the classes of the artifacts.

Table 3 summarizes the debloating results for the dependencies, classes, and methods. Interestingly, JDBL completely removes the bytecode for 20.3 % of the dependencies. In other words, 38 dependencies in the dependency tree of the projects are not necessary to successfully execute the workload in our dataset. At the class level, we find 60.1 % of bloat, from which we determine that 36.7 % belong to dependencies. JDBL debloats 59.4 % of the methods, from which 41.8 % belong to dependencies.

**Answer to RQ3:** JDBL removes bytecode in all libraries. It reduces the number of dependencies, classes, and methods by 20.3 %, 60.1 %, and 59.4 %, respectively. This result confirms the relevance of the coverage-based debloating approach for reducing the unnecessary bytecode of Java projects, while preserving their correctness.

Table 4. Size in Bytes of the Elements in the JAR Files

| Metrics | Size in bytes (%) |
|---|---|
| Resources | 257,982,707 (22.4 %) |
| Bytecode | 893,531,088 (77.6 %) |
| Non-bloated classes | 283,424,921 (31.7 %) |
| Bloated classes | 596,337,540 (66.7 %) |
| Bloated methods | 13,768,627 (1.5 %) |
| Total size | 1,151,513,795 |

*5.2.2 RQ4. What is the impact of using the coverage-based debloating approach on the size of the packaged artifacts?* We consider all the elements in the JAR files before the debloating, and study the size of the debloated version of the artifact, with respect to the original bundle. Decreasing the size of JAR files by removing bloated bytecode has a positive impact on saving space on disk, and helps reduce overhead when the JAR files are shipped over the network.

JAR files contain bytecode, as well as additional resources that depend on the functionalities of the artifacts (e.g., HTML, DLL, SO, and CSS files). JAR files also contain resources required by Maven to handle configurations and dependencies (e.g., MANIFEST.MF and *pom.xml*). However, JDBL is designed to debloat only executable code (class files). Therefore, we assess the impact of bytecode removal with respect to the executable code in the original bundle.

Table 4 summarizes the main metrics related to the content and size of the JAR files in our dataset. We observe that the additional resources represent 22.4 % of the total JAR size, whereas 77.6 % of the size is dedicated to the bytecode. This observation supports the relevance of debloating the bytecode in order to shrink the size of the Maven artifacts.

Overall, the bloated elements in the compiled artifacts in our dataset represent 610.3/893.7 MB (68.3 %) of pure bytecode: 596.5 MB of bloated classes and 13.8 MB of bloated methods. The used bytecode represents 31.7 % of the size. Interfaces, enumeration types, annotations, and exceptions represent 15.8% of the size on the disk of all the class files. In comparison with the classes, the debloat of methods represents a relatively limited size reduction. This is because we are reporting the removal of methods in the classes that are not entirely removed by JDBL. Furthermore, the methods cannot be completely removed, only the body of the method is replaced by an exception as detailed in Section 3.3.

Figure 8 shows a beanplot comparing the distribution of the percentage of bytecode reduction in the libraries that have no dependency, with respect to the libraries that have at least one dependency. From our observations, the average bytecode size reduction in the libraries that have dependencies (46.7 %) is higher than in the libraries with no dependencies (14.5 %). Overall, the average percentage of bytecode reduction for all the libraries is 25.8 %. We performed a two-sample Wilcoxon test, which shows that there are significant differences between those two groups (p-value $< 0.01$). Therefore, we reject the null hypothesis that the coverage-based debloating approach has the same impact in terms of reduction of the JAR size for libraries that declare dependencies, and libraries that do not.

We perform a Spearman's rank correlation test between the original number of classes in the libraries and the size of the removed bytecode. We found that there is a significant positive correlation between both variables ($\rho = 0.97$, p-value $< 0.01$). This result confirms the intuition that projects with many classes tend to occupy more space on disk due to bloat. However, the decision of what is necessary or not heavily depends on the library, as well as on the workload.

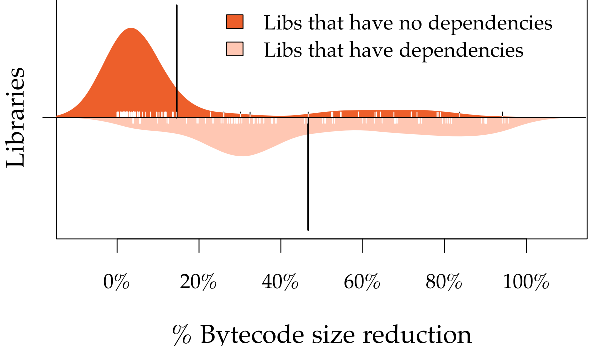



Fig. 8. Distribution of the percentage of reduction of the JAR size in libraries that have no dependencies and libraries that have at least one dependency, with respect to the original bundle.

**Answer to RQ4:** JDBL removes 68.3 % of pure bytecode in JAR files, which represents an average size reduction of 25.8 % per library JAR file. The JAR size reduction is significantly higher in libraries with at least one dependency compared to libraries with no dependency.

*5.2.3 RQ5. How does coverage-based debloating compare with the state-of-the-art of Java debloating regarding the size of the packaged artifacts and behavior preservation?* In this research question, we compare the debloating results of JDBL with respect to JSHRINK. The comparison is based on two metrics: the reduction of bytecode size after debloat, and the preservation of test results after debloat. The JSHRINK benchmark includes 25 Java projects. To answer RQ5, we discard eight projects: Six that are multi-module (JDBL is designed to debloat single-module Maven projects), and two projects whose builds fail due to test errors caused by unavailable network resources. Therefore, our comparison of JDBL and JSHRINK is based on 17 projects in total. For each project, we configure it to execute JDBL and generate a debloated version of the fat JAR. To validate the semantic correctness of the debloated artifact, we execute the project's test suite on the debloated version.

Table 5 describes the benchmark along with the debloating results obtained with JSHRINK and JDBL. The first column shows the name of the project as it appears on GitHub. The second column shows the commit SHA of the project, which is the same used in the companion article of JSHRINK [8]. The third column is the size of the original packaged JAR of the project, which includes all its dependencies with a `compile` scope. Projects are sorted in decreasing order according to their original size.

We report the bytecode size reduction of the debloated version of the projects achieved with JSHRINK and JDBL. The average size reduction achieved with JDBL is 35.1 %, which is more than double the size reduction obtained with JSHRINK. An explanation is that JSHRINK makes a more conservative debloating decision by setting all public methods, main methods, and test methods of libraries as entry points to approximate possible usages, whereas JDBL debloats according to a workload (i.e., the coverage information collected by running the test suite of the library). We observe that the percentage of reduction varies greatly among the projects depending on their size. We performed a Spearman's rank correlation test between the size of the original compiled project and the percentage of size reduction obtained with JSHRINK and JDBL. We found that there is a significant positive correlation between both variables for JSHRINK ($\rho$ = 0.52, p-value < 0.05) and

Table 5. Debloating Results of JShrink and JDBL in the Benchmark of Bruce et al. [8]

| Project Name | Commit SHA | Size (kB) | #Tests | JShrink Size Reduction | JShrink #Test Failures | JDBL Size Reduction | JDBL #Test Failures |
|---|---|---|---|---|---|---|---|
| lanterna | 5982dbf | 18,050.3 | 34 | 2.0 % | ✓ (0) | 66.4 % | ✓ (0) |
| AutoLoadCache | 06f6754 | 14,845.8 | 11 | 20.2 % | ✗ (9) | 66.8 % | ✓ (0) |
| gwt-cal | e7e5250 | 13,059.8 | 921 | 17.5 % | ✓ (0) | 42.2 % | ✓ (0) |
| maven-config-processor-plugin | c92e588 | 5,762.4 | 77 | 29.8 % | ✓ (0) | 73.0 % | ✓ (0) |
| Bukkit | f210234 | 4,781.3 | 906 | 18.5 % | ✓ (0) | 61.5 % | ✓ (0) |
| Mybatis-PageHelper | 525394c | 4,272.3 | 106 | 23.9 % | ✗ (55) | 66.4 % | ✓ (0) |
| RxRelay | 82db28c | 2,295.9 | 58 | 17.5 % | ✓ (0) | 10.0 % | ✓ (0) |
| RxReplayingShare | fbedd63 | 2,117.6 | 20 | 22.1 % | ✓ (0) | 7.3 % | ✓ (0) |
| qart4j | 70b9abb | 1,869.0 | 1 | 46.8 % | ✓ (0) | 59.1 % | ✓ (0) |
| retrofit1-okhttp3-client | 9993fdc | 718.9 | 9 | 11.5 % | ✓ (0) | 21.9 % | ✓ (0) |
| junit4 | 67d424b | 407.1 | 1,081 | 6.8 % | ✗ (13) | 4.7 % | ✓ (0) |
| gson | 27c9335 | 235.8 | 1,050 | 5.5 % | ✓ (0) | 1.7 % | ✓ (0) |
| zt-zip | 6933db7 | 134.3 | 121 | 11.3 % | ✓ (0) | 10.3 % | ✓ (0) |
| TProfiler | 8344d1a | 102.8 | 3 | 10.2 % | ✓ (0) | 83.1 % | ✓ (0) |
| Algorithms | 9ae21a5 | 99.9 | 493 | 5.5 % | ✓ (0) | 12.2 % | ✓ (0) |
| http-request | 2d62a3e | 36.1 | 163 | 6.6 % | ✓ (0) | 7.5 % | ✓ (0) |
| DiskLruCache | 3e01635 | 22.7 | 61 | 1.7 % | ✓ (0) | 2.9 % | ✓ (0) |
| Total | N.A | 6,881,194.8 | 5,115 | N.A | 77 | N.A | 0 |
| Median | N.A | 1,869.0 | 77 | 11.5 % | 0 | 21.9 % | 0 |
| Avg. | N.A | 4,047.8 | 300.9 | 15.14 % | 4.53 | 35.1 % | 0 |

JDBL ($\rho = 0.52$, p-value $< 0.05$). This result confirms the results obtained in RQ4 where we show that larger libraries are prone to become more bloated.

To assess the behavior preservation of the debloated projects w.r.t. their original version, we run existing test cases before and after debloating. Table 5 shows the semantic preservation capabilities of JShrink and JDBL on the benchmark of 17 projects. We consider a debloated project to have broken semantics if at least one of its tests fails after debloating. A project with no broken semantic is denoted by ✓, while ✗ denotes the presence of at least one test failure after debloating. JShrink causes test failures in three projects (77 failures in total). On the contrary, JDBL preserves the behavior of all the projects on this benchmark, according to the results of the tests.

**Answer to RQ5:** JDBL successfully debloats the 17 single-module Java projects in the benchmark of Bruce et al. [8], with a size reduction of 35.1 % on average, and preserves the behavior according to the tests. JShrink reduces size by 15.1 % on average. This is evidence that coverage-based debloating is a promising technique that advances the state-of-the-art of Java bytecode debloating.

### 5.3 Impact of Debloating on Library Clients (RQ6 and RQ7)

In this section, we study the repercussions of performing coverage-based debloating on library clients. To the best of our knowledge, this is the first experimental report that measures the impact of debloating libraries on the syntactic and semantic correctness of their clients.

*5.3.1 RQ6. To what extent do the clients of debloated libraries compile successfully?* In this research question, we investigate how debloating a library with a coverage-based approach impacts the compilation of the library's clients. We hypothesize that the essential functionalities of the library are less likely to be debloated, limiting the syntactic negative impact on their clients.

As described in Section 4.3.4, we consider the 1,354 clients that use the 211 debloated libraries that pass all the tests. We check that the clients use at least one class in the library through static analysis. We identify 988/1,354 (73.0 %) clients that satisfy this condition. Figure 9 shows the

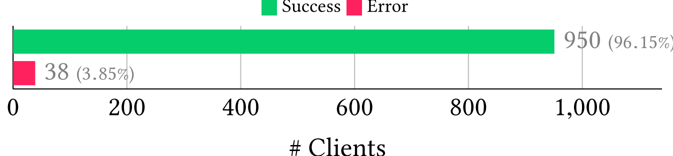



Fig. 9. Results of the compilation of the 988 clients that use at least one debloated library in the source code.

Table 6. Frequency of the Errors for the 38 Unique Clients that have Compilation Errors

| Description | # Libraries | # Clients | Occurrence |
|---|---|---|---|
| Cannot find class | 12/19 (63.2 %) | 20/38 (52.6 %) | 314/640 (49.1 %) |
| Unmappable character for encoding UTF8 | 1/19 (5.3 %) | 1/38 (2.6 %) | 100/640 (15.6 %) |
| Package does not exist | 6/19 (31.6 %) | 13/38 (34.2 %) | 78/640 (12.2 %) |
| Cannot find variable | 7/19 (36.8 %) | 9/38 (23.7 %) | 77/640 (12.0 %) |
| Cannot find method | 1/19 (5.3 %) | 1/38 (2.6 %) | 28/640 (4.4 %) |
| Static import only from classes and interfaces | 1/19 (5.3 %) | 1/38 (2.6 %) | 14/640 (2.2 %) |
| Method does not override a method from a supertype | 2/19 (10.5 %) | 2/38 (5.3 %) | 12/640 (1.9 %) |
| Processor error | 2/19 (10.5 %) | 11/38 (28.9 %) | 11/640 (1.7 %) |
| Cannot find other symbol | 2/19 (10.5 %) | 2/38 (5.3 %) | 4/640 (0.6 %) |
| UnsupportedOperationException | 1/19 (5.3 %) | 1/38 (2.6 %) | 1/640 (0.2 %) |
| Plugin verification | 1/19 (5.3 %) | 1/38 (2.6 %) | 1/640 (0.2 %) |
| 11 Unique errors | 19 Unique libraries | 38 Unique clients | 640 Compilation errors |

Note that a client can have multiple errors from different categories.

results obtained after attempting to compile the clients with the debloated library. JDBL generates debloated libraries for which 950 (96.2 %) of their clients successfully compile.

From the 988 clients that use at least one class of the library, we only observe compilation failures in 38 (3.8 %) clients. Table 6 shows our manual classification of the errors, based on the analysis of the Maven build logs. The first column describes the error message, columns 2–3 represent the number of libraries that trigger this kind of error, the number of clients that are affected, and the percentage relative to the number of libraries or clients impacted by a compilation error. Note that a client can be impacted by several different errors. The fourth column represents the occurrence of the error in the clients, as quantified from the Maven logs.

The causes of compilation errors are diverse. However, they are primarily due to errors related to missing packages, classes, methods, and variables (79.8 % of all the compilation errors). The debloating procedure directly causes those errors, as the bytecode elements are removed, and the clients do not compile. We detect 640 errors in total. Most of them occur for similar causes. Indeed, 20/38 (52.6 %) clients are not compiling because of one single error cause (which can be unique for each client). Moreover, when a client fails for a library, the other clients of the same library are generally failing for the same reason. It means that a single action can solve most of the client's problems, i.e., by adding the missing element to the debloated library.

Several clients face compilation issues because of their plugins. In order to have the client use the debloated library, we inject the debloated library inside the bytecode folder of the clients. Unfortunately, some plugins of the clients will also analyze the bytecode of the debloated library that may not follow the same requirements. `Plugin verification error`, `Unmappable character for encoding UTF8`, and `Processor error` are related to this type of bytecode validation.

In the list of errors, we also observe a runtime exception: `UnsupportedOperationException`. This error is unexpected since the compilation should not execute code and therefore should not trigger a runtime exception. It happens during the build of jenkinsci/warnings-plugin, which uses the *commons-io:2.6* library. In this case, the compilation itself of this client does not fail, but the

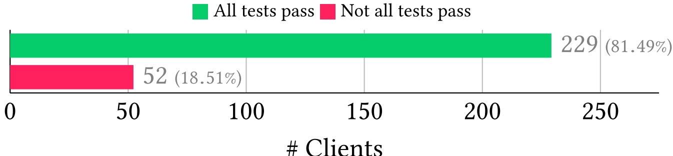



Fig. 10. Results of the tests on the 281 clients that cover at least one debloated library.

Table 7. Frequency of the Exceptions Thrown During the Execution of the Tests for the 52 Unique Clients that have Failing Test Cases

| Exception | # Libraries | # Clients | Occurrence |
|---|---|---|---|
| UnsupportedOperationException | 27/37 (73.0 %) | 40/52 (76.9 %) | 609/716 (85.1 %) |
| IllegalStateException | 1/37 (2.7 %) | 2/52 (3.8 %) | 55/716 (7.7 %) |
| NoClassDefFoundError | 6/37 (16.2 %) | 6/52 (11.5 %) | 24/716 (3.4 %) |
| Assert | 5/37 (13.5 %) | 5/52 (9.6 %) | 12/716 (1.7 %) |
| ExceptionInInitializerError | 4/37 (10.8 %) | 4/52 (7.7 %) | 4/716 (0.6 %) |
| TargetHostsLoadException | 1/37 (2.7 %) | 1/52 (1.9 %) | 4/716 (0.6 %) |
| NullPointerException | 1/37 (2.7 %) | 1/52 (1.9 %) | 3/716 (0.4 %) |
| TimeoutException | 1/37 (2.7 %) | 1/52 (1.9 %) | 2/716 (0.3 %) |
| PushSenderException | 1/37 (2.7 %) | 1/52 (1.9 %) | 2/716 (0.3 %) |
| AssertionError | 1/37 (2.7 %) | 1/52 (1.9 %) | 1/716 (0.1 %) |
| 10 Unique exceptions | 37 Unique libraries | 52 Unique clients | 716 Failing tests |

Note that a client can have multiple exceptions from different categories.

Maven build does. One of the Maven plugins of this project relies on one method that is debloated in `apache/commons-io`, therefore, the compilation does not fail because of the source code of the client but because of one particular Maven plugin used by the client.

**Answer to RQ6:** JDBL preserves the syntactic correctness of 950 (96.2 %) clients that use a library debloated by JDBL. This is the first empirical demonstration that debloating can preserve essential functionalities to successfully compile the clients of debloated libraries.

*5.3.2 RQ7. To what extent do the clients behave correctly when using a debloated library?* In this research question, we analyze another facet of debloating that may affect the clients: the disturbance of their behavior. To do so, we use the test suite of the clients as the oracle for assessing correct behavior. If a test in the client fails with the debloated library, then the coverage-based debloating breaks the behavior of the client.

We consider 1,354 clients to answer this question. They use the 211 debloated libraries that pass all the tests. We check that the client tests cover at least one class in the library, through dynamic code coverage. We identify 281/1,354 (20.8 %) clients that satisfy this condition.

Figure 10 presents the test results after building the clients with the debloated library. In total, 229 (81.5 %) clients pass all their tests, i.e., they behave the same with the original and with the debloated library. There are 52 (18.5 %) clients that have more failing test cases with the debloated library than with the original. However, the number of tests that fail is only 716/44,357 (1.6 %) of the total number of tests in the clients. This indicates that the negative impact of debloated libraries, as measured by the number of affected client tests, is marginal.

We investigate the causes of the test failures. Table 7 quantifies the exceptions thrown by the clients. The first column shows the 10 types of exceptions that we find in the Maven logs. Columns

2–3 represent the number of libraries involved in the failure and the number of clients affected. Column 4 represents the occurrence of the exception, as quantified from the logs.

`UsupportedOperationException` is the most frequent exception, with 609 occurrences in the failing tests, affecting 76.9 % of clients with errors. This exception is triggered when one of the debloated methods is executed. The second most common exception is `IllegalStateException`, with a total of 55 occurrences. This exception happens when the client tries to load a bloated configuration class and fails. The third most frequent exception, with 24 occurrences, is `NoClassDefFoundError`. Similar to `UsupportedOperationException`, it happens when the clients try to load a debloated class in the library.

Interestingly, there are only 12 assertion-related exceptions (11 `Assert` and 1 `AssertionError`). Having runtime exceptions that are triggered during the executions of the clients is less harmful than having behavior changes. The runtime exceptions can be monitored and handled while running the client, while a behavior change can stay hidden and corrupt the state of the clients.

**Answer to RQ7:** JDBL preserves the behavior of 229 (81.5 %) clients of debloated libraries. The 52 other clients still pass 43,684 (98.5 %) of their test cases. In these cases, 99.1 % of the test failures are due to a missing class or method, which can be easily located and fixed. This experiment shows that the risks of removing code in software libraries are limited for their clients.

## 6 DISCUSSION

In this section, we discuss key aspects of the design for coverage-based debloating. Then, we address the threats to the validity of the evaluation of JDBL.

### 6.1 Complementarity of Code-Coverage Tools

As presented in Section 3.3, we leverage the diversity of implementations of code coverage tools and the dynamic logging capabilities of the JVM class loader to collect precise coverage information. Now, we discuss the advantages of using this approach for debloating.

We collect and aggregate the coverage reports of the four tools used by JDBL to capture class usage information: JaCoCo, JCov, Yajta, and the JVM class loader. We consider a class as covered if it is reported as used by at least one of these tools. Figure 11 shows a Venn diagram of the classes reported as covered by each tool. There are a total of 76,549 classes covered by at least one tool in our dataset of 211 successfully debloated libraries. One key observation is that JaCoCo covers only 59,934 (78.3 %) of the classes that are used when running our workloads. This means that if we rely on JaCoCo only, the state-of-the-art coverage tool for Java, we would capture a significant share of false positive cases of bloated classes. This is critical, since removing these classes would produce a debloated project that cannot be properly used to run the workload. Another interesting observation is about the diversity of behaviors in modern coverage tools. There are only 34,582 (45.2 %) classes that are covered by the four tools. The JVM class loader is the one that captures the largest number of unique classes: 14,972 (19.6 %). This is because it logs the usage of dynamically loaded classes at runtime. In contrast, the other coverage tools can provide more fine-grained coverage information (e.g., methods and instruction) but miss usages of dynamically loaded resources (e.g., classes loaded via the Java reflection mechanism). The addition of JCov and Yajta improves the coverage of used classes, accounting for 2 and 662 unique covered classes, respectively.

This is evidence that combining the features of several coverage tools is relevant to accurately capture the code that is used at runtime. Capturing the complete coverage of classes that are necessary for a workload is critical for debloating. Failing to do so leads to the generation of a debloated project that does not compile, or even worse, that leads to runtime errors when client projects use

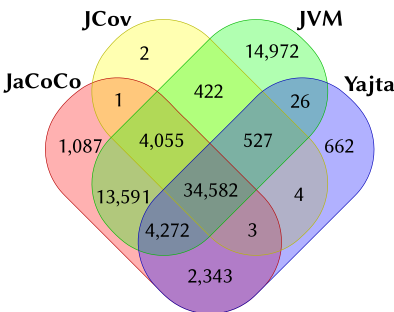



Fig. 11. Number of classes covered by the diverse tools used to collect coverage information in our dataset of 211 successfully debloated libraries.

debloated libraries. A more in-depth analysis of the similarities and differences of code coverage tools is a promising direction for future research on Java bytecode debloating.

## 6.2 Execution Time

The vision for debloating is to integrate it as part of building pipelines. In this context, execution time is an important consideration. Here, we discuss whether the execution time of JDBL makes coverage-based debloating feasible for everyday software development. In our experiments, we executed JDBL on 395 libraries, for a total of 1 day and 10:55:00 h. This represents an average debloating time of 5.3 minutes per library. Table 8 shows the execution times of JDBL and JShrink, with the benchmark of Bruce et al. [8]. The first and second columns show the projects' names and commit SHAs, respectively. The third and fourth columns show the execution time of JDBL and JShrink for each project, measured in seconds and sorted in decreasing order according to the results of JDBL. The comparison is made on the same hardware as the main JDBL experiment, described in Section 4.3. We observe that it took a total of 2,685 seconds (less than one hour) to debloat the benchmark with JDBL, whereas JShrink took a total of 29,397 seconds (more than eight hours). The average debloating time for this benchmark using JDBL is 157.9*s* (less than 3 minutes per project), which is 11 times faster than JShrink.

With times in the range of minutes, coverage-based debloating with JDBL can be used in a daily build. We also show that JDBL significantly improves the state-of-the-art of Java debloating, regarding execution time. This is an important contribution toward the integration of debloating into regular development processes.

## 6.3 Threats to Validity

*6.3.1 Internal Validity.* The threats to internal validity relate to the effectiveness of JDBL on generic real-world Java projects, as well as the design decisions that can influence our results. Coverage-based debloating has some inherent limitations, e.g., inadequate test cases and random behaviors. To mitigate these threats, we use as study subjects libraries with high coverage (see Table 1), and execute the test suite three times to avoid test randomness. If the test suite does not capture all desired behaviors, some necessary code might not be executed and be removed. The debloated libraries can also have non-deterministic test cases. For example, tests that use the current date and time to perform an action or not. Due to these behaviors, executing the application multiple times with the same input may lead to different coverage results.

As explained in Section 3.3, JDBL relies on a complex stack of existing bytecode coverage tools. It is possible that some of these tools may fail to instrument the classes for a particular project. However, since we rely on a diverse set of coverage tools, the failures of one specific tool are

Table 8. Execution Time of JDBL and JShrink in the Benchmark of Bruce et al. [8]

| Project | Commit SHA | JDBL Execution Time (s) | JShrink Execution Time (s) |
|---|---|---|---|
| maven-config-processor-plugin | c92e588 | 612 | 1,159 |
| lanterna | 5982dbf | 365 | 1,628 |
| AutoLoadCache | 06f6754 | 347 | 5,941 |
| gwt-cal | e7e5250 | 293 | 2,075 |
| Bukkit | f210234 | 143 | 10,452 |
| Mybatis-PageHelper | 525394c | 128 | 1,749 |
| RxRelay | 82db28c | 109 | 502 |
| RxReplayingShare | fbedd63 | 95 | 753 |
| junit4 | 67d424b | 80 | 352 |
| retrofit1-okhttp3-client | 9993fdc | 77 | 506 |
| qart4j | 70b9abb | 72 | 2,137 |
| DiskLruCache | 3e01635 | 66 | 223 |
| http-request | 2d62a3e | 63 | 616 |
| gson | 27c9335 | 63 | 842 |
| zt-zip | 6933db7 | 60 | 111 |
| TProfiler | 8344d1a | 56 | 98 |
| Algorithms | 9ae21a5 | 56 | 253 |
| Total | N.A | 2,685 | 29,397 |
| Median | N.A | 80 | 753 |
| Avg. | N.A | 157.9 | 1,729.2 |

likely to be corrected by the others. JDBL relies on the official Maven plugins for dependency management and test execution. Still, due to the variety of existing Maven configurations and plugins, JDBL may crash at some of its phases due to conflicts with other plugins. To overcome this threat and to automate our experiments, we set the `maven-surefire-plugin` to its default configuration, and use the `maven-assembly-plugin` to build the fat JAR of all the study subjects.

*6.3.2 External Validity.* The threats to external validity are related to the generalizability of our findings. Our observations in Section 5 about bloat are made on single-module Maven projects and the Java ecosystem. Our findings are valid for software projects with these particular characteristics. Meanwhile, coverage-based debloating in different languages could yield different conclusions than ours. Moreover, our debloating results are influenced by the coverage of the libraries and clients used as study subjects. However, we took care to select open-source Java libraries available on GitHub, which cover projects from different domains (e.g., logging, database handling, encryption, IO utilities, metaprogramming, and networking).[16] To the best of our knowledge, this is the largest set of study subjects used in software debloating experiments.

*6.3.3 Construct Validity.* The threats to construct validity are related to the relation between the coverage-based debloating approach and the experimental protocol. Our analysis is based on a diverse set of real-world open-source Java projects, with minimal modifications to run JDBL (only the *pom.xml* file is modified). We assume that all the plugins involved in the Maven build life-cycle are correct, as well as all the generated reports. Note that, if a dependency is not resolved correctly by Maven, then its bytecode will not be instrumented. Thus, the quality of the debloat result depends on the effectiveness of the Maven dependency resolution mechanism.

The applicability of coverage-based debloating depends on the quality of the workload. In our experiments, we rely on the projects' test suite. Consequently, our observations partly depend on the coverage of the projects. As explained in Section 4.2, the coverage of the libraries in our dataset is high. On the other hand, the coverage of the clients is lower (20.24% on average), which may cause some used functionalities in the debloated libraries that are not executed by the clients' tests. However, we believe this does not affect our results because we assess the semantic correctness of

[16]See https://github.com/castor-software/jdbl-experiments/blob/master/dataset/data/jdbl_dataset.json.

the client applications when using the debloated version of a library based on the client's usage intent expressed by its test suite.

## 7 RELATED WORK

In this section, we present the works related to software debloating techniques and dynamic analysis.

### 7.1 Software Debloating

Research interest in software debloating has grown in recent years, motivated by the reuse of large open-source libraries designed to provide several functionalities for different clients [15, 27]. Seminal work on debloating for Java programs was performed by Tip et al. [53, 54]. They proposed a comprehensive set of transformations to reduce the size of Java bytecode including class hierarchy collapsing, name compression, constant pool compression, and method inlining. Recent works investigate the benefits of debloating Java frameworks and Android applications using static analysis. Jiang et al. [26] presented JRed, a tool to reduce the attack surface by trimming redundant code from Java binaries. RedDroid [25] and PolyDroid [20] propose debloating techniques for mobile devices. They found that debloating significantly reduces the bandwidth consumption used when distributing the application, improving the performance of the system by optimizing resources. Other works rely on debloating to improve the performance of the Maven build automation system [9], removing bloated dependencies [50], and mitigating runtime bloat [59]. More recently, Haas et al. [19] investigate the use of static analysis to detect unnecessary code in Java applications based on code stability and code centrality measures. Most of these works show that static analysis, although conservative by nature, is a useful technique for debloating in practice.

To improve the debloating results of static analysis, recent debloating techniques drive the removal process using information collected at runtime. In this context, various dynamic analysis strategies can be adopted, e.g., monitoring, debugging, or performance profiling. This approach allows debloating tools to collect execution paths, tailoring programs to specific functionalities by removing unused code [21, 45, 55]. Unfortunately, most of the existing tools currently available for this purpose do not target large Java applications, focusing primarily on small C/C++ executable binaries. Sharif et al. [46] propose Trimmer, a debloating approach that relies on user-provided configurations and compiler optimization to reduce code size. Qian et al. [42] present RAZOR, a tool for debloating program binaries based on test cases and control-flow heuristics. However, the authors do not provide a thorough analysis of the challenges and benefits of using code coverage to debloat software. More recently, Bruce et al. [8] propose JShrink, a tool to dynamically debloat modern Java applications. However, JShrink is not directly automatable within a build pipeline and the effect of debloating on the library clients is not studied. These previous works assess the impact of debloating on the size of the programs, yet, they rarely evaluate to what extent the debloating transformations preserve program behavior.

This work contributes to the state-of-the-art of software debloating. We propose an approach for debloating Java libraries based on the usage of code coverage to identify unused software parts. Our tool, JDBL, integrates the debloating procedure into the Maven build life-cycle, which facilitates its evaluation and its integration in most real-world Java projects. We evaluate our approach on the largest set of programs ever analyzed in the debloating literature, and we provide the first quantitative investigation of the impact of debloating on the library clients.

### 7.2 Dynamic Analysis

Dynamic analysis is the process of collecting and analyzing the data produced from executing a program. This long-time advocated software engineering technique is used for several tasks,

Table 9. Comparison of Existing Java Debloating Techniques, w.r.t. this Work

| Tool | Target | Analysis | Scale | Granularity | | | | Correctness evaluation criteria | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | F | M | C | D | lib compiles | lib's tests pass | lib's clients tests pass |
| DepClean [50] | Source | Static | 30 libs | ✗ | ✗ | ✗ | ✓ | ✓ | ✓ | ✗ |
| Jax [54] | Bytecode | Static | 13 libs | ✓ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| JRed [26] | Bytecode | Static | 9 libs | ✗ | ✓ | ✓ | ✗ | ✓ | ✗ | ✗ |
| J-Reduce [28] | Bytecode | Dynamic | 3 libs | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ |
| JShrink [8] | Bytecode | Hybrid | 26 libs | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✗ |
| JDBL | Bytecode | Dynamic | 395 libs and 1,370 clients | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

Target is the type of artifact considered for debloating; Analysis refers to the type of code analysis performed for debloating; Scale provides the number of study subjects used to evaluate the technique; Granularity is the code level at which debloating is performed: field (F), method (M), class (C) or dependency (D); the last columns enumerate the three ways found in the state-of-the-art to assess the validity and utility of the debloated artifact.

such as program slicing [2], program comprehension [12], or dynamic taint tracking [4]. Through dynamic analysis, developers can obtain an accurate picture of the software system by exposing its actual behavior. For example, trace-based compilation uses dynamically-identified frequently-executed code sequences (traces) as units for optimizing compilation [16, 24]. Mururu et al. [37] implemented a scheme to perform demand-driven loading of libraries based on the localization of call sites within its clients. This approach allows reducing the exposed code surface of vulnerable linked libraries, by predicting the near-exact set of library functions needed at a given call site during the execution. Palepu et al. [40] use dynamic analysis to effectively summarize the execution and behavior of modern applications that rely on large object-oriented libraries and components. In this work, we employ dynamic analysis for bytecode reduction, as opposed to runtime memory bloat, which was the target of previous works [5, 35, 36, 38, 39, 58, 60].

In Java, dynamic analysis is often used to overcome the limitations of static analysis. Landman [31] performed a study on the usage of dynamic features and found that reflection was used in 78 % of the analyzed projects. Recent work from Xin et al. [57] utilizes execution traces to identify and understand features in Android applications by analyzing their dynamic behavior. In order to leverage dynamic analysis for debloating, we need to collect a very accurate coverage report, which guides the debloating procedure.

Our work contributes to the state-of-the-art of dynamic analysis for Java programs. Our technique combines information obtained from four distinct code coverage tools through bytecode instrumentation [6]. The composition of these four types of observations allows us to build a very accurate and complete coverage report, which is necessary to identify exactly what parts of the code are used at runtime and which ones can be removed. To collect coverage, we rely on the test suite of the libraries. This approach is similar to other dynamic analyses, e.g., for finding backward incompatibilities [10].

Table 9 summarizes the state-of-the-art of published techniques for debloating Java applications in comparison with JDBL. As observed, most existing techniques target bytecode instead of source code. DepClean [50] is the exception, which focuses on debloating *pom.xml* files based on static bytecode analysis. JShrink [8] uses a combination of static and dynamic analysis to address the potential unsoundness of static analysis in the presence of new language features. To our knowledge, JDBL is the first fully automatic debloating technique that also debloats code in third-party dependencies, and that assesses the correctness of debloating with respect to both the successful build of the debloated library and the successful execution of the library's clients. Furthermore, our experiments are at least one order of magnitude larger than previous works.

## 8 CONCLUSION

In this work, we introduce coverage-based debloating for Java applications. We have addressed one key challenge of dynamic debloating: collect accurate and complete coverage information that includes the minimum set of classes and methods that are necessary to execute the program with a given workload. We implemented coverage-based debloating in an open-source tool called JDBL. We have performed the largest empirical validation of Java debloating in the literature with 354 libraries and 1,354 clients that use these libraries. We evaluated JDBL using an original experimental protocol that assessed the impact of debloating on the libraries' behavior, their size, as well as on their clients. Our results indicate that JDBL can reduce 68.3 % of the bytecode size and that 211 (69.9 %) debloated libraries compile and preserve their test behavior. We also show that JDBL outperforms JShrink regarding size reduction and behavior preservation when used on the same benchmark as in the JShrink article. For the first time in the literature, we assess the utility of debloated libraries for their clients: 81.5 % of the clients can successfully compile and run their test suite with a debloated library.

Our results provide evidence of the massive presence of unnecessary code in software applications and the usefulness of debloating techniques to handle this phenomenon. Furthermore, we demonstrate that dynamic analysis can be used to automatically debloat libraries while preserving the functionalities that are necessary for their clients.

The next step of coverage-based debloating is to specialize applications with respect to usage profiles collected in production environments, extending the debloating to other parts of the program stack, e.g., to the Java Runtime Environment (JRE), program resources, or containerized applications. As for the empirical investigation of the impact of debloating, we aim at evaluating the effectiveness of coverage-based debloating in reducing the attack surface of modern applications. These are major milestones towards full-stack debloating for software hardening.

## ACKNOWLEDGMENTS

We would like to thank Dr. Bobby Bruce and all the authors of JShrink for sharing the source code and providing valuable feedback during our reproduction of their experiments.

## REFERENCES

[1] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-scale debloating of binary shared libraries. *Digital Threats: Research and Practice* 1, 4 (2020), 28 pages.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. *SIGPLAN Notices* 25, 6 (1990), 246–256.

[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the USENIX Security Symposium.* 1697–1714.

[4] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating dynamic data flow in commodity JVMs. *ACM SIGPLAN Notices* 49, 10 (2014), 83–101.

[5] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. In *Proceedings of the OOPSLA.* 745–764.

[6] Walter Binder, Jarle Hulaas, and Philippe Moret. 2007. Advanced java bytecode instrumentation. In *Proceedings of the Symposium on Principles and Practice of Programming in Java.* 135–144.

[7] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[8] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern java applications. In *Proceedings of the ESEC/FSE.* 135–146.

[9] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for java projects. In *Proceedings of the ESEC/FSE.* 643–654.

[10] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ICSE.* 112–124.

[11] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the FEAST Workshop*. 23–29.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[13] DiUS. 2022. Brings the Popular Ruby Faker Gem to Java. Retrieved November 21st, 2021 from https://github.com/DiUS/java-faker.

[14] Thomas Durieux, César Soto-Valero, and Benoit Baudry. 2021. DUETS: A dataset of reproducible pairs of java library-clients. In *Proceedings of the MSR*. 545–549.

[15] Sebastian Eder, Henning Femmer, Benedikt Hauptmann, and Maximilian Junker. 2014. Which features do my users (Not) use?. In *Proceedings of the ICSME*. 446–450.

[16] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices* 44, 6 (2009), 465–478.

[17] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.

[18] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. 2021. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021), 56 pages.

[19] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is static analysis able to identify unnecessary source code? *ACM Transactions on Software Engineering and Methodology* 29, 1 (2020), 1–11.

[20] Brian Heath, Neelay Velingker, Osbert Bastani, and Mayur Naik. 2019. PolyDroid: Learning-driven specialization of mobile applications. arXiv:1902.09589. Retrieved from https://arxiv.org/abs/1902.09589.

[21] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the CCS*. 380–394.

[22] G. J. Holzmann. 2015. Code inflation. *IEEE Software* 32, 2 (2015), 10–13.

[23] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. 2019. Code coverage differences of java bytecode and source code instrumentation tools. *Software Quality Journal* 27, 1 (2019), 79–123.

[24] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the CGO*.

[25] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *Proceedings of the ISSRE*. 189–199.

[26] Y. Jiang, D. Wu, and P. Liu. 2016. JRed: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the COMPSAC*. 12–21.

[27] Y. Jiang, C. Zhang, D. Wu, and P. Liu. 2016. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the HASE*. 122–131.

[28] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ESEC/FSE*. 556–566.

[29] Peter Kampstra. 2008. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software* 28, 1 (2008), 1–9.

[30] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the EuroSec Workshop*.

[31] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of java reflection: Literature review and empirical study. In *Proceedings of the ICSE*. 507–518.

[32] Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools. In *Proceedings of the ISSRE*. 380–389.

[33] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification*. Pearson Education.

[34] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. 2021. The nature of build changes. *Empirical Software Engineering* 26, 3 (2021), 32.

[35] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Four trends leading to Java runtime bloat. *IEEE Software* 27, 1 (2009), 56–63.

[36] N. Mitchell, E. Schonberg, and G. Sevitsky. 2010. Four trends leading to java runtime bloat. *IEEE Software* 27, 1 (2010), 56–63.

[37] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2019. Binary debloating for security via demand driven loading. arXiv:1902.06570. Retrieved from https://arxiv.org/abs/1902.06570.

[38] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 2018. Understanding and combating memory bloat in managed data-intensive systems. *ACM Transactions on Software Engineering and Methodology* 26, 4 (2018), 41 pages.

[39] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the ESEC/FSE*. 268–278.

[40] Vijay Krishna Palepu, Guoqing Xu, and James A. Jones. 2017. Dynamic dependence summaries. *ACM Transactions on Software Engineering and Methodology* 25, 4, (2017), 41 pages.

[41] S. Ponta, W. Fischer, H. Plate, and A. Sabetta. 2021. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *Proceedings of the ICSME*. 555–558.

[42] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the USENIX Security Symposium*. 1733–1750.

[43] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A multi-OS cross-layer study of bloating in user programs, kernel, and managed execution environments. In *Proceedings of the FEAST Workshop*. 65–70.

[44] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically debloating containers. In *Proceedings of the ESEC/FSE*. 476–486.

[45] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for java. *ACM Transactions on Programming Languages and Systems* 25, 4 + (2003), 452–499.

[46] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *Proceedings of the ASE*. 329–339.

[47] César Soto-Valero. 2022. Open-science repository for the experiments with JDBL. Retrieved November 21st, 2021 from https://github.com/castor-software/jdbl-experiments.

[48] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *Proceedings of the MSR*. 333–343.

[49] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the ESEC/FSE*. 1021–1031.

[50] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

[51] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the soundness of call graph construction in the presence of dynamic language features—a benchmark and tool evaluation. In *Proceedings of the Programming Languages and Systems*. Sukyoung Ryu (Ed.), 69–88.

[52] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. 2016. Negative effects of bytecode instrumentation on java source code coverage. In *Proceedings of the SANER*. 225–235.

[53] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. 1999. Practical experience with an application extractor for java. In *Proceedings of the OOPSLA*. 292–305.

[54] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical extraction techniques for java. *ACM Transactions on Programming Languages and Systems* 24, 6 (2002), 625–666.

[55] H. C. Vázquez, A. Bergel, S. Vidal, J. A. Díaz Pace, and C. Marcos. 2019. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology* 107 (2019), 18–29.

[56] Niklaus Wirth. 1995. A plea for lean software. *IEEE Computer* 28, 2 (1995), 64–68.

[57] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. 2019. Identifying features of android apps from execution traces. In *Proceedings of the MOBILESoft*. 35–39.

[58] Guoqing Xu. 2013. CoCo: Sound and adaptive replacement of java collections. In *Proceedings of the ECOOP*. 1–26.

[59] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology* 23, 3 (2014), 50 pages.

[60] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FoSER*. 421–426.

[61] Qian Yang, J. Jenny Li, and David M. Weiss. 2009. A survey of coverage-based testing tools. *The Computer Journal* 52, 5 (2009), 589–597.

[62] Hao Zhong and Hong Mei. 2019. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2019), 319–334.

[63] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries. *ACM Transactions on Embedded Computing Systems* 18, 5 (2019), 1–18.


Received 8 December 2021; revised 19 May 2022; accepted 1 June 2022

# Paper V

# The Multibillion Dollar Software Supply Chain of Ethereum

**César Soto-Valero, Martin Monperrus, and Benoit Baudry**, KTH Royal Institute of Technology

*Ethereum is the single largest programmable blockchain platform today. Ethereum nodes operate the blockchain, relying on a vast supply chain of third-party software dependencies. In this article, we perform an analysis of the software supply chain of Java Ethereum nodes and distill the challenges of maintaining and securing this blockchain technology.*

Ethereum is a spearhead of the blockchain paradigm, with its smart contract infrastructure supporting a vibrant decentralized finance ecosystem[1] and a blooming art scene.[2] Since the release of Bitcoin in 2008,[3] the adoption of blockchain-based solutions has grown significantly, mainly driven by the promise of secure, reliable, and decentralized monetary and financial transactions.[4] There are several public blockchains running today (for example, Bitcoin, Ethereum, Litecoin, and NEO), each one of them serving a particular purpose and solving specific problems. In this article, we focus on the single case of Ethereum as it is the largest blockchain platform by most notable metrics.

Ethereum is a feature-rich platform, considered by some as the avant-garde of blockchain technologies.[5] It has its own cryptocurrency (Ether), its own consensus protocol, and its own smart contract platform. Ethereum

*Digital Object Identifier 10.1109/MC.2022.3175542*
*Date of current version: 26 September 2022*

digital assets and contracts are executed in a distributed manner in nodes that support the Ethereum Virtual Machine execution model. Due to this functionality, the Ethereum platform is often compared to a globally distributed supercomputer. In January 2022, the Ethereum blockchain held more than hundreds of billions of U.S. dollars in digital assets, and an average of 250 new smart contracts are deployed and verified on Etherscan every day (https://etherscan.io/chart/verified-contracts).

The research community has contributed to the creation and evolution of blockchain technologies. The recent work focuses on three aspects[6]: the theoretical foundations, scalability, and engineering of smart contracts. However, one key aspect of the blockchain has been completely overlooked: its software supply chain.

The software supply chain of an ecosystem is the set of all software libraries, tools, and third-party modules that compose it.[7] In the context of Ethereum, the software supply chain is first and foremost formed of the different open source implementations of Ethereum nodes in Go, Rust, Java, and other languages.[8] The node implementations themselves depend on hundreds of components. Overall, the software supply chain of Ethereum is composed of libraries and tools, as well as dependencies, to develop, deploy, and run Ethereum nodes.

Recent studies have shown that a large network of software dependencies, such as in Ethereum nodes, can turn into an application's Achilles heel.[9] On the one hand, malicious actors may infect a target application from within a reused component.[10] On the other hand, entire software systems may crash because of a bug somewhere deep in the reuse chain.[11] The major stakeholders in the Ethereum ecosystem want it to be resistant to attackers and robust with respect to bugs.[12] Consequently, understanding and hardening its software supply chain has become of utmost importance.

In this article, we take a deep dive into the software supply chain of the two main Ethereum nodes written in Java, namely Besu and Teku. Our focus on Java is motivated by the strong presence of Java in banks and financial institutions, an essential target group of Ethereum, as well as by the availability of advanced tools for analyzing and hardening the supply chain in Java.[11]

IN THIS ARTICLE, WE TAKE A DEEP DIVE INTO THE SOFTWARE SUPPLY CHAIN OF THE TWO MAIN ETHEREUM NODES WRITTEN IN JAVA, NAMELY BESU AND TEKU.

We analyze the software supply chains of two Java Ethereum nodes, looking at their dependencies and their evolution over time. This provides the community with the first ever description of a mission-critical software supply chain for blockchain. Next, we provide actionable results and show that we can harden a complex software supply chain with relevant tools. Our results reveal a number of key insights and technical challenges both for researchers in software supply chains as well as for developers and stakeholders of the Ethereum community.

## THE SUPPLY CHAIN OF JAVA ETHEREUM NODES

### Overview

Ethereum is a public distributed system of nodes supporting a ledger. As a distributed system, Ethereum nodes agree on a consensus protocol to verify the validity of transactions. The protocol for Ethereum v1.0 (Eth1) is based on proof of work, and for Ethereum v2.0 (Eth2), it is based on proof of stake. Ethereum nodes communicate peer-to-peer without a central organizing institution. They run smart contracts and receive transactions from client applications. This is what is depicted in the outer parts of Figure 1.

The top part of Figure 1 illustrates the network of Ethereum nodes. The left part of the figure illustrates the three main categories of clients of the Ethereum blockchain: an individual user, a crypto exchange marketplace, and a bank. The individual user is, for example, an artist who relies on the blockchain to distribute her artwork.[13] The crypto exchange marketplace provides deposits and withdrawals of the Ether cryptocurrency. The bank uses Ethereum to accelerate payments across borders, opening up the possibility to help underbanked populations.[14] The central part of Figure 1

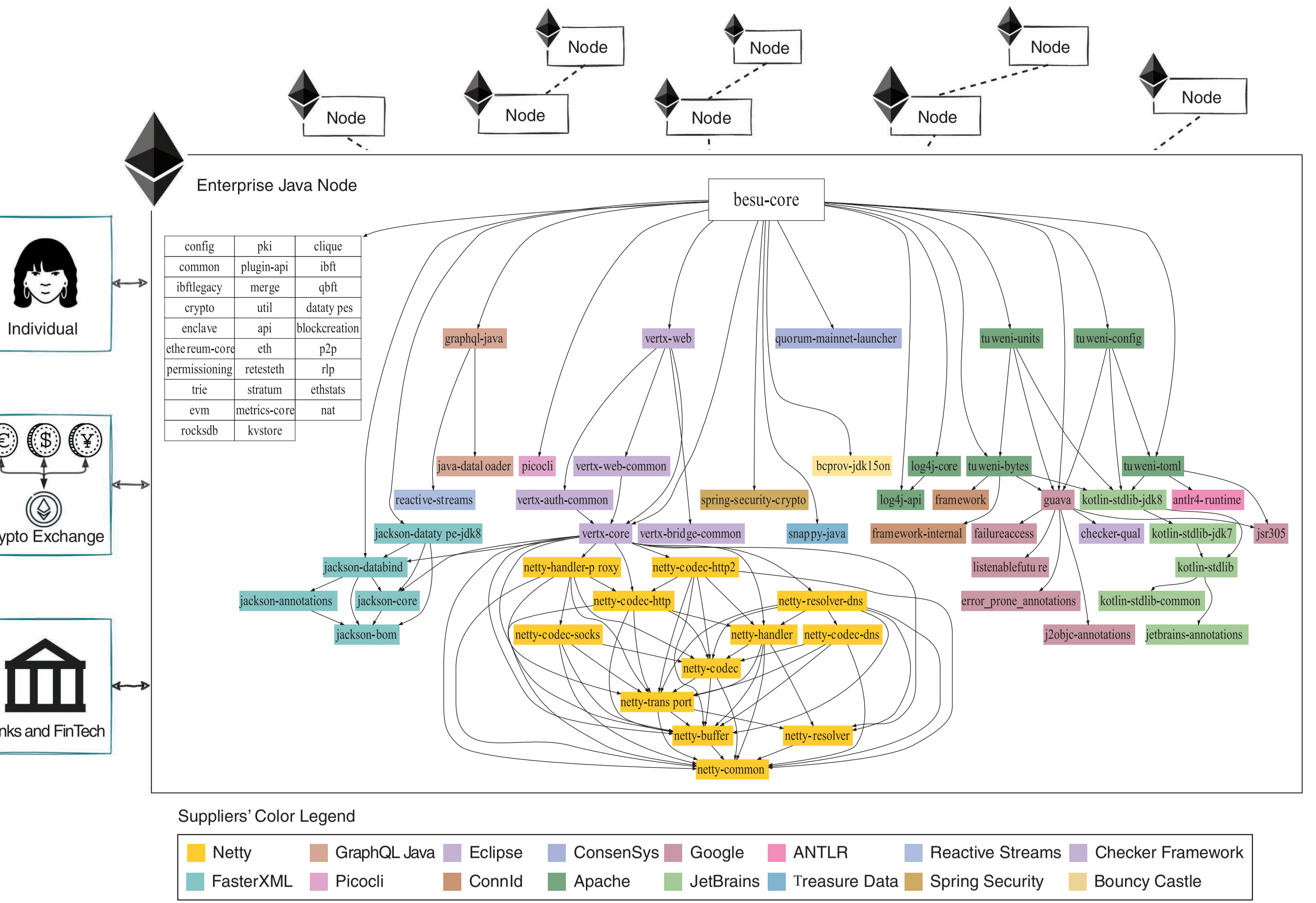



**FIGURE 1.** An excerpt of the software supply chain of one single module in the enterprise Java Ethereum node Besu v21.10.6. The besu-core module depends on 29 other Besu modules as well as on 51 third-party dependencies provided by 16 different supplier organizations; the third-party dependencies are colored according to the name of the supplier. FinTech: financial technology.

focuses on one single Ethereum node and provides a deep dive into its software supply chain.

From now on, we assume that this node runs the Java implementation of Ethereum called *Besu*. Figure 1 shows the graph of dependencies for the core module of Besu, which is only one of the 41 modules of Besu, focusing on its direct dependencies. Such a large number of software dependencies is a potential source of vulnerabilities and supply chain attacks.[15] In the context of Ethereum, this means that the software dependencies of Besu represent a potential source of risk for the financial system and art market built on top of it. In the rest of this article, we study potential countermeasures.

## Besu

Besu is the leading Java implementation for Ethereum Eth1. Besu is led by the Hyperledger Foundation, a nonprofit organization for open source enterprise blockchain tools, which started in December 2015 as a spin-off of the Linux Foundation. As of January 2022, there are at least 44 nodes of Besu running on the Ethereum Mainnet public network, according to Ethernodes (https://www.ethernodes.org). The source code of Besu is available on GitHub (https://github.com/hyperledger/besu). It is a reasonably sized and active project, containing a total of 268,356 lines of code written in Java, contributed through 3,125 commits. Besu is an active project; its code base is developed and maintained by a total of 115 contributors with a unique GitHub account (of which 29 are listed as official maintainers). The contributors reported and closed 916 issues and merged a total of 739 pull requests in 2021. More than half of the contributors work at Hyperledger, according to their GitHub profiles.

In Table 1, we capture some key statistics about the software supply chain of dependencies for Besu v21.10.6, released on 5 January 2022. The raw data and analysis scripts are available online (https://github.com/chains-project/ethereum-ssc). We collected those dependencies using the Gradle dependencies' resolution plugin. On the analyzed release, Besu is made of 41 Gradle modules. These modules are internal dependencies since their development, maintenance, and release lifecycles are under the direct control of the Besu developers. In addition to these 41 modules, Besu relies on 355 unique third-party dependencies provided by 165 distinct supplying organizations. This number represents the number of different third-party Java libraries in the dependency tree of Besu without considering the different versions of a dependency. The supplier organizations are in charge of maintaining these artifacts and releasing new versions to external repositories, with no formal ties with Hyperledger and Besu for most of them.

In the central part of Figure 1, we zoom into the dependency tree of one of the 41 modules of Besu: besu-core. The compilation of this module depends on 29 internal dependencies, shown on the far left of the figure, as well as on 51 third-party dependencies that are also necessary for compilation. The third-party dependencies are colored according to the name of the supplier organization. For example, the third-party dependencies in dark yellow are handled by the supplier "Netty." Overall, the supply chain of besu-core is made of libraries maintained by 16 distinct suppliers. We note that:

**TABLE 1.** Descriptive statistics of the software supply chain of the two major enterprise Java Ethereum nodes: Besu v20.10.4 (commit ID 120d0d4) and Teku v21.1.0 (commit ID dcfb0eb).

| | Besu (Eth1) | Teku (Eth2) |
|---|---|---|
| Lines of Java code | 268,356 | 209,860 |
| Commits | 3,125 | 3,142 |
| Contributors | 115 | 65 |
| Unique internal dependencies | 41 | 57 |
| Unique third-party dependencies | 355 | 293 |
| Unique suppliers | 165 | 146 |
| Unique third-party dependencies introduced since January 2021 | 127 | 79 |
| Unique third-party suppliers introduced since January 2021 | 49 | 22 |
| Unique third-party dependency versions modified since January 2021 | 171 | 150 |

1. Many suppliers are large organizations with high code-quality standards (for example, Apache, Google, and JetBrains), which are trusted and relied on by many clients.
2. There are partner suppliers, such as the quorum-mainnet-launcher library developed by ConsenSys. Although not being maintained by Hyperledger, the developers are close to the professional network of Besu developers.
3. Some libraries in the supply chain of besu-core belong to personal GitHub accounts, such as picocli and snappy-java. They are maintained and released by a single developer and cannot arguably be trusted as much as dependencies from big tech organizations or partner suppliers.[16]

### Teku

Teku is the leading Eth2 Java node built to meet enterprise requirements. For example, Teku provides enterprise features, such as monitoring with Prometheus, Representational State Transfer application programming interfaces for managing Eth2 node operations, and external key management to handle validator signing keys. Teku is an open source project under active development on GitHub (https://github.com/ConsenSys/teku). The first commit to the Teku code base was made on 9 September 2018. Since then, the project has seen a rapid development pace, accounting for 3,142 commits contributed by a total of 65 developers.

Table 1 shows the descriptive statistics for the software supply chain of Teku v22.1.0, released on 3 January 2022. The project contains a total of 57 unique internal dependencies and relies on 293 unique third-party dependencies. Like Besu, Teku ships a large body of code coming from third-party dependencies with each new release. As with Besu, they are provided by 146 distinct suppliers with different code quality and security standards. For the Ethereum ecosystem, the security and reliability of Besu's and Teku's supply chains are equally important. One crashing bug or successful attack on either of them would potentially be devastating.

**WE BELIEVE THAT THE SYSTEMATIC ASSESSMENT AND ENFORCEMENT OF DIVERSITY IN SOFTWARE SUPPLY CHAINS ARE IMPORTANT AND PROMISING RESEARCH AVENUES.**

### Supply chain evolution

In the bottom part of Table 1, we give novel insights about the evolution of the Java Ethereum software supply chains. We built the dependency trees of both supply chains from January 2021: Besu v20.10.4 (commit ID 120d0d4) and Teku v21.1.0 (commit ID dcfb0eb). We compare these trees with the versions released one year later, in January 2022. We collect the number of dependencies introduced and modified in the supply chain of Besu and Teku as well as the number of additional suppliers that have appeared along the evolution of these supply chains. We found 127 unique dependencies in the dependency tree of Besu and 79 dependencies in the dependency tree of Teku that are present in 2022 and that were not in the tree of 2021. This represents a significant growth of both supply chains, indicating the need for regular monitoring and assessment of the projects' dependencies.

The growth also holds for the number of suppliers of dependencies. In one year, there have been 49 and 22 new suppliers in the supply chains of Besu and Teku, respectively. This is clear evidence that a complex supply chain evolves fast. Consequently, an approach based on "allow" and "deny" lists of suppliers is not viable as it would necessitate frequent updates of these lists and potential delays in their assessments. The management of the software supplier risks must be supported by tools that regularly monitor, analyze, and assess the supply chain to cope with this evolution.

### Supply chain diversity

The Ethereum community values and explicitly promotes the development and the maintenance of a diversity of node implementations.[8] Ethereum experts consider that node diversity is essential for the network to be healthy and secure. Besu and Teku are two different node implementations, built by different development teams,

following different development road maps. Meanwhile, their software dependencies represent a large body of their code bases. We assess the diversity among these implementations by looking into the diversity among their dependencies and suppliers. To do so, we extract the intersection of the dependencies of Besu and Teku. The supply chains of both nodes share a total of 190 third-party dependencies, representing the 53.5 and 64.8% of the dependencies of Besu and Teku, respectively. This is illustrated in Figure 2. Furthermore, we observe that 92 suppliers are common to both node implementations. Even though Besu and Teku may look like entirely different node implementations, our results indicate that they actually carry out a large body of common code, a potential common failure point. This suggests that the Ethereum community may work on supply chain diversity in addition to node diversity for further increasing resilience.

Let us discuss the case of a dependency that is shared by both Besu and Teku: the Apache logging library log4j. In December 2021, a new Common Vulnerabilities and Exposures (CVE) was published, documenting an exploit affecting all versions of log4j from version 2.0 to 2.14.1 (CVE-2021-44228).[17] This caused a major disruption on the web as log4j is a third-party dependency in thousands of software supply chains, including the ones of very critical services, such as Amazon and Microsoft Azure. This vulnerability allows an attacker to perform arbitrary remote code execution on the running application, exploiting the vulnerable version of the log4j library. Now, assume that an attacker had had the time to exploit this vulnerability in Ethereum Java nodes.

Since both nodes share the same dependency, it means that the scale of the repercussions would have been amplified. The whole Ethereum ecosystem (both Eth1 and Eth2) would have suffered from potential chain splits, violations of the consensus protocol, and in the worst case, loss of funds and Bored Apes. If the two implementations had relied on diverse suppliers of logging facilities, the common failure risk would have been reduced. For example, Logback or Tinylog are trustworthy alternatives to log4j. Migrating from log4j to Logback in Besu requires minimal engineering effort: fewer than 10 files need to be modified, thanks to modern Java logging architectures. This diversification would benefit Besu nodes by providing different logging implementations from different suppliers, decreasing the chances of vulnerabilities with a blast effect. We believe that the systematic assessment and enforcement of diversity in software supply chains are important and promising research avenues.

## SUPPLY CHAIN REMEDIATIONS

The two enterprise Java Ethereum nodes, Besu and Teku, depend on hundreds of third-party dependencies. Today, there exist tools that can automatically enforce dependency management policies. Those policies include license checking, supplier approval, update frequency, and security. In this article, we focus on the solutions for the latter two: identify outdated dependencies and replace vulnerable dependencies.

### The remediation of outdated dependencies

Third-party libraries constantly evolve to fix defects, patch vulnerabilities,

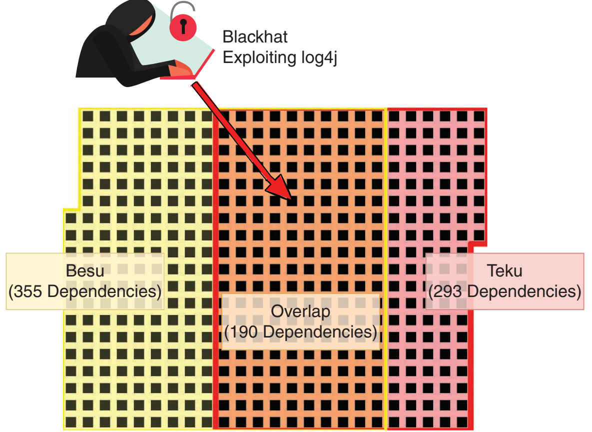



**FIGURE 2.** The limited diversity among the software supply chains of Besu and Teku. The former includes 355 unique third-party dependencies, and the latter is made of 293 dependencies, but 190 dependencies are shared by both supply chains. The shared dependencies represent 53.5 and 64.8% of the Besu and Teku supply chains.

and add features. To take full advantage of third-party code, it is considered best practice to keep the dependencies up to date. However, it is hard to stay up to date when the supply chain of a project includes a large number of dependencies, each of them having different lifecycles and release schedules. In a large dependency tree, it is not uncommon that there is one new version of some dependency in the tree released per day.

Keeping dependency up to date first means being aware of new versions (for example, due to a new release announcement or a security advisory). Once outdated dependencies are identified, the developers ensure that the update does not introduce breaking changes. Finally, they commit a change to bump the dependency version. To our knowledge, the developers of Besu and Teku currently perform this monitoring and update procedure manually.

```
@@ —49,7 +49,7 @@ dependencyManagement {
— dependency 'commons—codec:commons—codec:1.13'
+ dependency 'commons—codec:commons—codec:1.15'
```

**LISTING 1.** An example of a commit diff from a manual pull request (PR #3235; see https://github.com/hyperledger/besu/pull/3235) made by a developer to update the dependency commons-codec in Besu.

**TABLE 2.** An overview of risk metrics in the software supply chain of Besu v20.10.4 (commit ID 120d0d4) and Teku v21.1.0 (commit ID dcfb0eb). The data were obtained in 15 January 2022.

| | Besu (Eth1) | Teku (Eth2) |
|---|---|---|
| Outdated third-party dependency versions (Dependabot) | 3 | 1 |
| Outdated third-party dependency versions (Renovate) | 49 | 19 |
| Vulnerable third-party dependency versions (OWASP) | 11 | 2 |
| Vulnerable third-party dependency versions (WhiteSource) | 15 | 17 |

```
@@ —96,7 +96,7 @@ dependencyManagement {
— dependencySet(group: 'org.apache.logging.log4j', version:
    ↪ '2.13.3') {
+ dependencySet(group: 'org.apache.logging.log4j', version:
    ↪ '2.15.0') {
     entry 'log4j—api'
     entry 'log4j—core'
     entry 'log4j—slf4j—impl'
```

**LISTING 2.** Commit diff (commit ID a52f376) showing a critical security update of the dependency log4j made to prevent a potential remote code exploit in Teku.

For instance, Listing 1 shows an example of a manual dependency update where a Besu developer updated the dependency commons-codec from version 1.13 to 1.15.

However, there exist software bots that automatically scan dependency trees and perform library updates, for example, Dependabot, Renovate, and Jared. To our knowledge, none of them are enabled in Besu and Teku. To assess their relevance in the context of these nodes, we performed a pilot experiment as follows. We forked their GitHub repositories and configured Dependabot and Renovate to identify and remediate outdated dependencies.

Table 2 shows the number of outdated third-party dependencies detected and reported by both dependency bots on 15 January 2022. Dependabot reports three and one outdated dependencies in Besu and Teku, respectively, whereas Renovate reports 49 and 19 outdated dependencies. Renovate identifies many more updates because 1) it supports various package managers and 2) it suggests updating infrastructure dependencies (in addition to application dependencies). Overall, both bots reveal several outdated dependencies that need to be acted upon. This confirms that using these state-of-the-art supply chain tools would allow Besu and Teku developers to be more up to date and more diligent in handling their dependencies. From an economic perspective, it would avoid the engineering burden of manually checking new releases of their dependencies.

## The remediation of vulnerable dependencies

Blackhat actors perform supply chain attacks.[15] They purposefully compromise one dependency to achieve malicious goals, such as theft or denial of

service. To put it simply, vulnerable dependencies may cause a range of problems for Ethereum nodes related to their confidentiality, integrity, or availability. Indeed, in December 2021, the Teku development team urgently mobilized their engineers after an important vulnerability disclosure: that of log4j[17] already mentioned earlier. Listing 2 shows the commit made to fix this critical vulnerability. To our knowledge, no successful attacks have been performed on Besu or Teku thanks to this timely commit.

As with outdated dependencies, technology exists to remediate vulnerable dependencies swiftly, such as Open Web Application Security Project (OWASP) Dependency Checker, Snyk, or WhiteSource. The identification of vulnerable dependencies in a software supply chain relies on scanning curated vulnerability databases, such as the National Vulnerability Database, and mapping vulnerability identifiers to versions in package repositories. To our knowledge, the Teku team runs, as a crontab job, a vulnerability identification tool called *Trivly* but without automated remediation.

We searched for vulnerable dependencies in Besu and Teku with two state-of-the-art tools, considered as among the best tools in this domain: the OWASP Dependency Checker and WhiteSource. Table 2 shows the results of the analysis, performed on 15 January 2022. OWASP detects 11 and 2 vulnerable dependencies in Besu and Teku, respectively, whereas WhiteSource detects 15 and 17 vulnerable dependencies. These results show that each tool focuses on different aspects; thus, there is currently no silver bullet to identify dependency vulnerabilities. Interestingly, OWASP and WhiteSource both report the dependency netty-transport as affected by several vulnerabilities, which can be considered as a severe issue. Also, we note that some vulnerable dependencies exist in both Besu and Teku, which is further evidence of the need for supply chain diversity discussed previously.

Neither Besu nor Teku uses the OWASP Dependency Checker or WhiteSource on a regular basis yet. Indeed, in January 2022, an active developer of Besu opened a pull request to add the OWASP dependency checker in the build pipeline of the project [see PR #3288 (https://github.com/hyperledger/besu/pull/3288), not merged at the time of writing]. Installing the OWASP Dependency Checker in the continuous integration pipeline of Besu would allow analyzing its dependency tree every time the node is built. This way, developers are notified early in the case of potential security issues related to third-party dependencies. At the moment of writing this article, such an initiative has not been taken for Teku. We believe that both Besu and Teku will eventually embed vulnerable dependency checking in their pipeline; this is inevitable for any major software project with a high stake.

In this article, we took a deep dive into the software supply chains of Besu and Teku. These two open source projects are the major enterprise Java Ethereum nodes, which are in charge of financial and artistic transactions worth billions of dollars. Our analysis reveals that both Ethereum node implementations are large software projects that depend on hundreds of libraries provided by a variety of supplier organizations.

Our work contributes to the state of the art of software supply chains, with unique insights about the complex networks of dependencies. We


## ABOUT THE AUTHORS

**CÉSAR SOTO-VALERO** is a Ph.D. student of software technology at the KTH Royal Institute of Technology, Stockholm, 10044, Sweden. His research work focuses on leveraging static and dynamic program analysis techniques to mitigate software bloat. Contact him at cesarsv@kth.se.

**MARTIN MONPERRUS** is a professor of software technology at the KTH Royal Institute of Technology, Stockholm, 10044, Sweden. His research interests include software engineering, with a focus on software reliability. Contact him at monperrus@kth.se.

**BENOIT BAUDRY** is a professor of software technology at the KTH Royal Institute of Technology, Stockholm, 10044, Sweden. His research interests include software engineering, with a focus on software testing and automatic diversification. Baudry received a Ph.D. in 2003 from the University of Rennes. Contact him at baudry@kth.se.

outlined the important growth of the software supply chains as well as the increase of the number of suppliers on which Besu and Teku rely. We showed where the state of the art lies with respect to remediation tools for hardening the software supply chain.

While the Ethereum community stresses the importance of maintaining and incentivizing a diversity of node implementations, we have shown that the supply chains of Besu and Teku share a majority of their third-party dependencies. This is a serious limitation for the software diversity in the Ethereum ecosystem. Also, we have shown that dependency management for Besu and Teku can be improved with automated remediation. The significance of our findings suggests that a similar analysis would be worthwhile for other Ethereum node implementations, such as Geth written in Go. Finally, we sincerely believe that our insights on the engineering of software supply chains hold for any blockchain that matters. ■

## ACKNOWLEDGMENT

This work was partially supported by the Wallenberg Autonomous Systems and Software Program funded by the Knut and Alice Wallenberg Foundation as well as by the TrustFull and the Chains projects funded by the Swedish Foundation for Strategic Research.

## REFERENCES

1. C. Dannen, *Introducing Ethereum and Solidity*. Berlin, Germany: Springer-Verlag, 2017.
2. R. Monroe, “When N.F.T.s invade an art town,” *The New Yorker*, 2022. [Online]. Available: https://www.newyorker.com/culture/infinite-scroll/why-bored-ape-avatars-are-taking-over-twitter
3. S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Rev.*, pp. 21,260, 2008.
4. J. Mackintosh, “Defi is crypto’s wall street, without a safety net,” *Wall Street Journal*, 2021. [Online]. Available: https://www.wsj.com/articles/defi-is-cryptos-wall-street-without-a-safety-net-11631611945
5. G. Wood, “Ethereum: A secure decentralised generalised transaction ledger,” *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.
6. H. Huang, W. Kong, S. Zhou, Z. Zheng, and S. Guo, “A survey of state-of-the-art on blockchains: Theories, modelings, and tools,” *ACM Comput. Surv.*, vol. 54, no. 2, pp. 1–42, Mar. 2021, doi: 10.1145/3441692.
7. C. Lamb and S. Zacchiroli, “Reproducible builds: Increasing the integrity of software supply chains,” *IEEE Softw.*, vol. 39, no. 2, pp. 1–10, 2021, doi: 10.1109/MS.2021.3073045.
8. P. Ranjan, “The state of client diversity in ethereum,” *Medium*, Aug. 20, 2020. [Online]. Available: https://medium.com/ethereum-cat-herders/the-state-of-client-diversity-in-ethereum-2ca915a3d768
9. A. Gkortzis, D. Feitosa, and D. Spinellis, “Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities,” *J. Syst. Softw.*, vol. 172, p. 110653, Feb. 2021, doi: 10.1016/j.jss.2020.110653.
10. M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2020, pp. 23–43.
11. F. Massacci and I. Pashchenko, “Technical leverage in a software ecosystem: Development opportunities and security risks,” in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng. (ICSE)*, 2021, pp. 1386–1397, doi: 10.1109/ICSE43902.2021.00125.
12. J.-P. Aumasson, D. Kolegov, and E. Stathopoulou, “Security review of ethereum beacon clients,” Sep. 2021, *arXiv e-print 2109.11677*.
13. @jawn.eth, “Jawn was here ethereum,” Block Number: 14129361, Notifi, 2022. [Online]. Available: https://www.notifi.xyz/messages/1512
14. S. Schuetz and V. Venkatesh, “Blockchain, adoption, and financial inclusion in India: Research opportunities,” *Int. J. Inf. Manage.*, vol. 52, p. 101936, Jun. 2020, doi: 10.1016/j.ijinfomgt.2019.04.009.
15. I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, “Vulnerable open source dependencies: Counting those that matter,” in *Proc. ACM/IEEE 12th Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, 2018, pp. 1–10, doi: 10.1145/3239235.3268920.
16. T. Gustavsson, “Managing the open source dependency,” *Computer*, vol. 53, no. 2, pp. 83–87, 2020, doi: 10.1109/MC.2019.2955869.
17. “CVE-2021-44228, Apache Log4j vulnerability,” National Vulnerability Database, Dec. 10, 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-44228

# Paper VI

# Automatic Specialization of Third-Party Java Dependencies

César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry
*KTH Royal Institute of Technology, Stockholm, Sweden*
Email: {cesarsv, deepikat, baudry}@kth.se

**Abstract**—Modern software systems rely on a multitude of third-party dependencies. This large-scale code reuse reduces development costs and time, and it poses new challenges with respect to maintenance and security. Techniques such as tree shaking or shading can remove dependencies that are completely unused by a project, which partly address these challenges. Yet, the remaining dependencies are likely to be used only partially, leaving room for further reduction of third-party code. In this paper, we propose a novel technique to specialize dependencies of Java projects, based on their actual usage. For each dependency, we systematically identify the subset of its functionalities that is necessary to build the project, and remove the rest. Each specialized dependency is repackaged. Then, we generate specialized dependency trees where the original dependencies are replaced by the specialized versions and we rebuild the project. We implement our technique in a tool called DEPTRIM, which we evaluate with $30$ notable open-source Java projects. DEPTRIM specializes a total of $343$ $(86.6\,\%)$ dependencies across these projects, and successfully rebuilds each project with a specialized dependency tree. Moreover, through this specialization, DEPTRIM removes a total of $60{,}962$ $(47.0\,\%)$ classes from the dependencies, reducing the ratio of dependency classes to project classes from $8.7\times$ in the original projects to $4.4\times$ after specialization. These results indicate the relevance of dependency specialization to significantly reduce the share of third-party code in Java projects.

**Index Terms**—Software specialization, software debloating, maven, software supply chain, software ecosystem

## 1 INTRODUCTION

SOFTWARE projects are developed by assembling new features and components provided by reusable third-party libraries. Software reuse at large is a known best practice in software engineering [1]. Its adoption has rocketed in the last decade, thanks to the rapid growth of repositories of reusable packages, along with the development of mature package managers [2]. These package managers let developers declare a list of third-party libraries that they want to reuse in their projects. The libraries declared by developers form the set of direct dependencies of the project. Then, at build time, the package manager fetches the code of these libraries, as well as the code of transitive dependencies, declared by the direct dependencies. This forms a dependency tree that the build system bundles with the project code into a package that can be released and deployed.

The large-scale adoption of software reuse [3] is beneficial for software companies as it reduces their delivery times and costs [4]. Meanwhile, reuse today has reached a point where most of the code in a released application actually originates from third-party dependencies [5]. This massive presence of third-party code in application binaries has turned software reuse into a double-edged sword [6]. Recent studies have highlighted the new challenges that third-party dependencies pose for maintenance [7], [8], performance [9], code quality [10], and security [11], [12].

Several techniques have emerged to address the challenges of dependency management. The first type of approach consists of supporting developers in maintaining a correct and secure dependency tree. Software composition analysis [13] and software bots [14] suggest dependency updates and warn about potential vulnerabilities among dependencies. Integrity-checking tools aim at preventing packaging a dependency with code that may have been tempered with. For example, the Go community maintains a global database for authenticating module content [15] and sigstore facilitates the procedure of signing third-party libraries [16]. A second type of approach to maintain healthy dependency trees consists in reducing it, removing the dependencies that are completely unused. Examples of such techniques include package debloating for Linux applications [17] depedency debloating or shading for Java applications [18], or tree shaking for JavaScript applications [19].

In this paper, we aim at advancing the state of the art of dependency tree reduction with a novel technique that specializes dependencies to the needs of an application. While good tools exist to remove dependencies that are completely bloated, i.e., none of their APIs is used by the application, there is little support to remove unused code from dependencies that are partially used. Yet, previous work has shown that applications only reuse a small fraction of the APIs within third-party libraries [20]. Early work on in-depth analysis of the dependency tree [21] has also shown evidence of the potentially large reduction of third-party code [22]. Motivated by these previous results, we introduce DEPTRIM, the first tool that specializes third-party libraries in the dependency tree of Java applications.

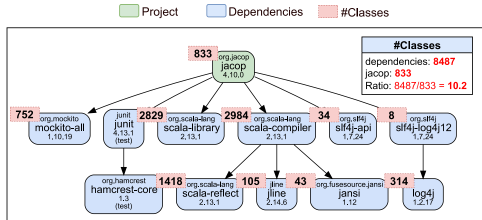
DEPTRIM analyses the bytecode of a Java project, as well as all its direct and transitive third-party dependencies. First, it removes the dependencies that are completely bloated, and identifies the non-bloated ones. Next, for each non-bloated dependency, DEPTRIM identifies the classes for which at least one member is reachable from the project. DEPTRIM

then removes the unused classes and repackages the rest into a specialized version of the dependency. Finally, DEPTRIM modifies the dependency tree of the project by replacing the original dependencies with the specialized versions. The output of DEPTRIM is a specialized dependency tree of the project, with the maximum number of specialized dependencies such that the project still builds correctly (*i.e.*, the project correctly compiles, and all its tests pass, guaranteeing that the expected behavior of the project is unchanged). DEPTRIM simplifies the reuse of specialized dependencies by generating reusable JAR files, which can be readily deployed to external repositories.

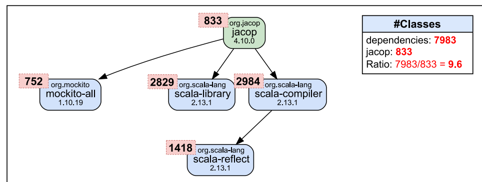
We demonstrate the capabilities of DEPTRIM by performing a study with 30 mature open-source Java projects that are configured to build with MAVEN. DEPTRIM successfully analyzes 135,343 classes across the 467 dependencies of the projects. For 14 projects, it generates a dependency tree in which all `compile`-scope dependencies are specialized. For the 16 other projects, DEPTRIM produces a dependency tree that includes all dependencies that can be specialized without breaking the build, while keeping the others intact. In total, DEPTRIM removes 60,962 (47.0 %) unused classes from 343 third-party dependencies. The specialized dependencies are deployed locally, as reusable JAR files. For each project, DEPTRIM produces a specialized version of the *pom.xml* file that replaces original dependencies with specialized ones, such that the project still correctly builds.

In summary, our contributions are as follows:

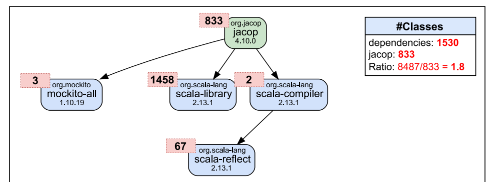
- A novel assessment of the ratio of dependency classes compared to project classes, based on actual class usages, performed on 30 mature open-source projects at three stages of the dependency tree: original, debloated, and specialized.
- A fully automated technique to specialize the dependency tree of Java projects at build time.
- The implementation of this technique in a tool called DEPTRIM, which automatically builds MAVEN projects with the largest subset of specialized dependencies.
- Empirical evidence that DEPTRIM successfully specializes the dependency tree of 14 projects in its entirety, and 16 partially, reducing the number of third-party classes by 47.0 %. The project classes to dependency classes ratio is divided by two, from 8.7 × to 4.4 ×.




(a) MAVEN dependency resolution (default)




(b) DEPCLEAN dependency debloating (state-of-the-art)




(c) Classes actually used in the dependencies (determined by static analysis)

Figure 1: Example of transformations in the dependency tree of the project `jacop v4.10.0`, and the impact of such transformations on the dependency classes to project classes ratio. Dependencies have `compile`-scope by default if not specified. Note that `jacop` reuses only a portion of its third-party dependencies.

## 2 BACKGROUND

In this section, we introduce the essential terminology of our work. Then, we illustrate the existing techniques to reduce the amount of dependency code. This is followed by a discussion about the opportunities for dependency specialization, in the context of a real-world Java project.

### 2.1 Terminology

In this work, we consider a software project as a collection of Java source code files and configuration files organized to be built with MAVEN [23]. MAVEN is a build automation tool for Java-based projects. It is primarily used for managing the dependencies of a project, testing it, and packaging it, as specified in the Project Object Model (POM) expressed in a file called *pom.xml*. This file, located at the root of the project, includes additional information such as the project name and version. We now define the key concepts about dependencies in the MAVEN ecosystem.

**Definition 1.** ***Maven dependency:*** A MAVEN dependency defines a relationship between a project and another compiled project. Dependencies are compiled JAR files, uniquely identified with a triplet (`G:A:V`) where `G` is the `groupId`, `A` is the `artifactId`, and `V` is the `version`. Dependencies are defined in the *pom.xml* within a scope, which determines the phase of the MAVEN build cycle at which the dependency is required. MAVEN distinguishes 6 dependency scopes: `compile`, `runtime`, `test`, `provided`, `system`, and `import`.

For example, the constraint programming solver `jacop` (`6ed0cd0`) is a MAVEN project. As illustrated in Figure 1a, `scala-library` is one of its 11 dependencies. This is a `compile`-scope dependency, which means that `jacop` can use some functionalities of `scala-library` at compile time, and will include all the code of `scala-library` within the packaged binary of `jacop`. The testing framework `junit` is

also declared as a dependency of `jacop` within the `test` scope, indicating that this dependency is required only for executing unit tests.

**Definition 2.** ***Dependency tree:*** The dependency tree of a MAVEN project is a directed acyclic graph that includes all the direct dependencies declared by developers in the project *pom.xml*, as well as all the transitive dependencies, *i.e.* dependencies in the transitive closure of direct dependencies. For a MAVEN project, there exists a dependency resolution mechanism that fetches both direct and transitive dependency `JAR` files not present locally from external repositories such as Maven Central [24]. The project becomes the root node of the tree, while the edges represent dependency relationships between its direct and transitive dependencies.

For example, in Figure 1a, `scala-compiler` is a direct dependency of `jacop` because it is declared by developers in the *pom.xml*. It depends on `scala-reflect`, which makes `scala-reflect` a transitive dependency of `jacop`. The 6 direct and 5 transitive dependencies of `jacop` constitute its dependency tree.

**Definition 3.** ***Bloated dependency:*** A dependency is said to be bloated if none of the elements in its API are used, directly or indirectly, by the project [25]. This means that, although they are present in the dependency tree of software projects, bloated dependencies are entirely unused. Developers are therefore encouraged to remove them [26].

For example, Figure 1b presents the `compile`-scope dependencies of `jacop` after the removal of its bloated dependencies. The dependencies `jline`, `jansi`, `sl4j-api`, `sl4j-log4j12`, and `log4j` are bloated and have been safely removed, as no member of their APIs is exercised by `jacop`.

### 2.2 Example

We now illustrate the MAVEN dependency resolution mechanism and the concept of bloated dependencies. Figure 1 shows the example of the transformations of the dependency tree of the project `jacop`. In Figure 1a, we see the dependency tree of `jacop` as generated by the MAVEN dependency resolution mechanism: it fetches `JAR` files from external repositories while omitting duplication, avoiding conflicts, and constructing a tree representation of the dependencies [27]. `jacop` has a total of 11 third-party dependencies: 6 are direct and 5 are transitive. Direct dependencies are explicitly declared by the developers in the *pom.xml* file of `jacop`, while transitive dependencies are resolved automatically via the MAVEN dependency resolution mechanism. MAVEN uses the concept of scope to determine the visibility and lifecycle of a dependency, *i.e.*, whether it should be included in the classpath of a certain build phase, as well as what the classpath of an artifact should be during the execution of a build phase. For example, `jacop` has 9 `compile`-scope dependencies (the default) and 2 `test` scope dependencies. When `jacop` is packaged for deployment as a `jar-with-dependencies`, its `JAR` file will include the bytecode of all its 9 `compile`-scope dependencies. These `compile`-scope dependencies include 8,487 `class` files, while the number of classes within `jacop`, written and tested by its developers, is 833. As observed, the number of classes contributed by third-party dependencies is one order of magnitude (i.e., 10 $\times$) more than the number of classes written by the `jacop` developers.

When we run DEPCLEAN, a state-of-the-art MAVEN plugin that identifies and removes bloated dependencies [25], [18], we find that 5 dependencies of `jacop` are never used, and are therefore marked as bloated. Figure 1b shows the dependency tree of `jacop` after `test`-scope dependencies and bloated dependencies are removed. In this case, the number of nodes in the tree is reduced from 11 to 4. The reduction in the number of `compile`-scope dependencies represents a removal of 504 (5.9 %) third-party classes (*e.g.*, removing `sl4j-api` leads to the removal of 34 classes). For `jacop`, the removal of bloated dependencies has a minimal impact on the reduction of third-party classes. Consequently, while complete dependency debloating drastically reduces the number of dependencies in the `jar-with-dependencies` of `jacop`, it only leads to a modest reduction in the ratio of dependency classes to project classes, from the original 10.2 $\times$ in Figure 1a to 9.6 $\times$.

To assess the opportunities of further reducing the number of dependency classes, we analyze the `JAR` of each non-debloated dependency of `jacop`. We compute the static call graph of method calls between the classes in the `JAR` files. Based on this graph, we get the list of dependency classes that are reachable from the project at build time. Figure 1c shows the number of reachable classes for each dependency of `jacop`. Consider the direct dependency `scala-compiler`. Of its 2,984 classes, only two are reachable from `jacop`. This confirms that `scala-compiler` is not a bloated dependency for `jacop`, and that it includes way more features than what `jacop` actually needs. This is evidence of the opportunity to specialize this dependency in the context of `jacop`. Similar opportunities exist for 2 other non-bloated dependencies. In fact, we find that 6,453/7,983 (80.8 %) of the third-party classes in these dependencies can be removed, and `jacop` can still build successfully. After dependency specialization, the ratio of the number of dependency classes to `jacop`'s classes is 1.8 $\times$. This is a drastic reduction from 9.6 $\times$ which was the ratio after debloating (Figure 1b), and even more significant if we consider the original ratio of 10.2 $\times$ (Figure 1a).

The number of classes actually used in the dependencies is significantly lower than the original number of classes provided. This observation motivates us to extend the state-of-the-art of Java dependency management with a novel technique to specialize non-bloated dependencies, by identifying and removing unnecessary classes through bytecode removal. In the next section, we present our approach and provide details on DEPTRIM, a tool that automatically specializes the dependencies of MAVEN projects.

## 3 DEPENDENCY SPECIALIZATION WITH DEPTRIM

This section presents DEPTRIM, an end-to-end tool for the automated specialization of third-party Java dependencies. We define the concept of dependency specialization, followed by an explanation of the key phases of DEPTRIM.

### 3.1 Dependency Specialization

This work introduces the concept of specialized dependencies and specialized dependency trees. We define them below.

**Definition 4.** ***Specialized dependency:*** A dependency is said to be specialized with respect to a project if all the

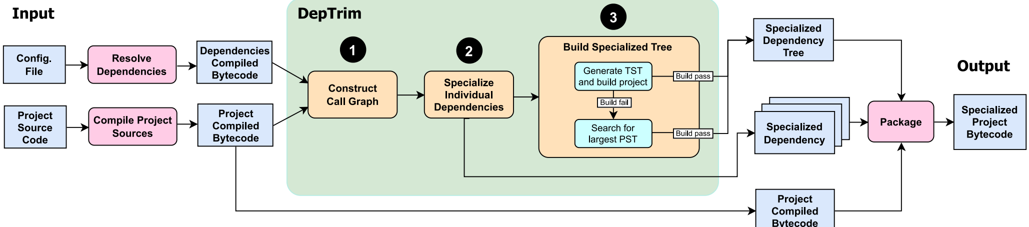



Figure 2: Overview of the dependency specialization approach implemented in DEPTRIM. Blue boxes are software artifacts, pink rounded boxes are actions performed by the build engine, and each of the three main phases of DEPTRIM are indicated within the green rounded box.

classes within the dependency are used by the project, and all unused classes have been identified and removed. Consequently, there is no class in the API of a specialized dependency that is unused, directly or indirectly, by the project or any other dependency in its dependency tree.

Recalling the example in Figure 1b and Figure 1c, `jacop` uses 2 of the 2,984 classes in `scala-compiler`. Therefore, `scala-compiler` could be specialized with respect to `jacop`, by removing the 2,982 unused classes.

**Definition 5.** ***Specialized dependency tree:*** A specialized dependency tree is a dependency tree where at least one dependency is specialized and the project still correctly builds with that dependency tree. This means that in at least one of the used dependencies, unused classes have been identified and removed. A specialized dependency tree may be one of the following two types:

- *Totally Specialized Tree* (`TST`): A dependency tree where all used dependencies are specialized and the project build is successful.
- *Partially Specialized Tree* (`PST`): A dependency tree with the largest possible number of specialized dependencies, such that the project build is successful.

We discuss our approach for building a project with a `TST` or `PST` with DEPTRIM in the following subsections. DEPTRIM identifies unused classes within the non-bloated `compile`-scope dependencies of MAVEN projects, and removes them in order to produce specialized dependencies. Using these, DEPTRIM prepares a specialized dependency tree for the project such that the project still correctly builds. The following subsections explain this technique in detail.

### 3.2 DEPTRIM

Figure 2 illustrates the complete pipeline of the dependency specialization approach implemented in DEPTRIM. DEPTRIM receives as inputs the source code and the *pom.xml* file of a Java MAVEN project. The project must successfully build. DEPTRIM outputs three elements: (i) a specialized version of the *pom.xml* file, which removes the bloated dependencies and includes the largest possible number of specialized dependencies that keep the build passing; (ii) the set of specialized dependencies as reduced `JAR` files; (iii) the project `JAR` compiled from its source, which can be packaged with the specialized dependencies in order to have a smaller `jar-with-dependencies` for release and deployment.

DEPTRIM validates that the project still builds correctly with the specialized dependency tree. Note that DEPTRIM only transforms the bytecode of the third-party dependencies, while the original project source and its compiled bytecode remain intact.

As illustrated in Figure 2, the specialization procedure of DEPTRIM consists of three main phases. First ❶, DEPTRIM leverages state-of-the-art Java static bytecode analysis to construct a static call graph of the class members in the third-party dependencies that are reachable from the project binaries. The completeness of this reachability analysis is critical for the identification of unused third-party classes. Second ❷, DEPTRIM transforms the bytecode in the dependencies to remove unused classes. This task requires integration with the MAVEN build engine to resolve and deploy the modified dependencies to the local repository. Finally ❸, DEPTRIM specializes the dependency tree of the project by modifying its original *pom.xml* file. The modified *pom.xml* should preserve the original configurations, except the dependency declarations, which point to the specialized dependencies instead of the original ones. Moreover, DEPTRIM must validate that dependency specialization does not break project build. We provide more details of these three phases in the following subsections.

#### 3.2.1 ❶ Call Graph Construction

Before it can specialize dependencies, DEPTRIM determines their API usage, based on static analysis of the project binary. To do so, DEPTRIM constructs a call graph using two inputs: the compiled dependencies as resolved by MAVEN (Line 1 of Algorithm 1), and the compiled project sources (Line 2). Then, using the bytecode class members of the project as entry points to this graph (Line 3), DEPTRIM infers and reports class usage information from the bytecode directly, without loading or initializing classes. The report captures the set of dependencies, classes, and methods that are actually used by the project, *i.e.*, that are reachable via static analysis. The output of this phase is a data structure that identifies the minimal set of classes in each of the dependencies that are required to build the project.

The collection of accurate and complete call graphs is essential for specialization. If a necessary class member is not reachable statically, then DEPTRIM will consider it as unused and proceed to remove it in a subsequent phase. To mitigate this limitation, DEPTRIM relies on state-of-the-art static analysis of Java bytecode to capture invocations between

classes, methods, fields, and annotations from the project and its direct and transitive dependencies. Furthermore, it parses the constant pool of `class` files in order to capture dynamic invocations from string literals (*e.g.*, when loading a class using its fully qualified name via reflection).

#### 3.2.2 ❷ Individual Dependency Specialization

The dependency specialization phase receives the call graph as input to specialize individual dependencies. During this phase, DEPTRIM determines which dependencies are bloated (*i.e.*, there is no path from the project bytecode toward any of the class members in the unused dependencies), and removes them from the original *pom.xml* (Line 4 of Algorithm 1). Next, DEPTRIM proceeds to remove the unused classes within non-bloated dependencies (Lines 5 to 10). Any dependency `class` file that is not present in the call graph is deemed unreachable and removed. Note that a Java source file can contain multiple classes, thus resulting in multiple `class` files after compilation. DEPTRIM considers this case as well by design, as it downloads, unzips, and removes the unused compiled classes directly from the project dependencies at build time (*i.e.*, during the MAVEN `package` phase). Once all the unused `class` files in a dependency are removed, DEPTRIM qualifies the dependency as specialized. Moreover, to facilitate reuse, DEPTRIM deploys each specialized dependency in the local MAVEN repository along with its *pom.xml* file and corresponding `MANIFEST.MF` metadata (Line 11).

The output of the second phase is a set of specialized `JAR` files for the dependencies of the project. These files include all the bytecode and resources that are necessary to be shared and reused by the other packages within the dependency tree. In particular, DEPTRIM takes care of keeping the classes in dependencies that may not be directly instantiated by the project, but are accessible from the used classes in the dependencies, with regard to the project.

#### 3.2.3 ❸ Dependency Tree Specialization

After specializing each non-bloated dependency, DEPTRIM produces a specialized version of the project *pom.xml* file that removes the bloated dependencies and points to the specialized dependencies instead of their original versions. This results in a `TST` or a `PST` for the project, as described in Definition 5.

First, DEPTRIM builds the totally specialized dependency tree (`TST`) of the project (Lines 12 to 15 of Algorithm 1). All specialized dependencies replace their original version in the project *pom.xml*. Then, in order to validate that the specialization did not remove necessary bytecode, DEPTRIM builds the project, *i.e.* its sources are compiled and its tests are run. If the build is a `SUCCESS`, DEPTRIM returns this `TST`.

In cases where the build with the `TST` fails, DEPTRIM proceeds to build the project with one specialized dependency at a time (Lines 17 to 24). Thus, rather than attempting to improve the soundness of the static call graph, which is proven to be challenging in Java [28], DEPTRIM performs an exhaustive search of the dependencies that are unsafe to specialize. At this step, DEPTRIM builds as many versions of the dependency tree as there are specialized dependencies, each containing a single specialized dependency. DEPTRIM attempts to build the project with each of these single specialized dependency trees. If the project build is successful, DEPTRIM marks the dependency as safe to be specialized. In case the dependency is not safe to specialize, DEPTRIM keeps the original dependency entry intact in the specialized *pom.xml* file. Finally, DEPTRIM constructs a partially specialized dependency tree (`PST`) with the union of all the dependencies that are safe to be specialized. Then, the project is built with this `PST` to verify that the build is successful. If all build steps pass, DEPTRIM returns this `PST`.

**Algorithm 1** Third-party dependency specialization

**Input:** $\mathcal{P}_{src}$: Project source code
**Input:** $\mathcal{P}_{obf}$: Project original build file (*pom.xml*)
**Output:** $\mathcal{P}_{\mathsf{TST}} \vee \mathcal{P}_{\mathsf{PST}}$

/** Call graph construction **/
1: $\mathcal{P}_{deps} \leftarrow resolve_dependencies(\mathcal{P}_{obf})$
2: $\mathcal{P}_{bin} \leftarrow compile(\mathcal{P}_{src})$
3: $\mathcal{CG} \leftarrow analyze(\mathcal{P}_{deps}, \mathcal{P}_{bin})$
4: $\mathcal{P}_{dbf} \leftarrow debloat(\mathcal{P}_{obf}, \mathcal{CG})$
/** Individual dependency specialization **/
5: $\mathcal{P}_{deps_specialized} \leftarrow \emptyset$
6: **for each** $dep \in \mathcal{P}_{dbf}$ **do**
7: reachable_classes $\leftarrow analyze(dep, \mathcal{CG})$
8: dep_specialized $\leftarrow specialize(dep,$ reachable_classes$)$
9: $\mathcal{P}_{deps_specialized} \leftarrow \mathcal{P}_{deps_specialized} \cup$ dep_specialized
10: **end for**
11: $deploy_locally(\mathcal{P}_{deps_specialized})$
/** Dependency tree specialization **/
12: $\mathcal{P}_{\mathsf{TST}} \leftarrow \emptyset$
13: $\mathcal{P}_{\mathsf{TST}} \leftarrow create_config_file(\mathcal{P}_{deps_specialized})$
14: **if** $build(\mathcal{P}_{\mathsf{TST}}, \mathcal{P}_{bin}) ==$ `SUCCESS` **then**
15: **return** $\mathcal{P}_{\mathsf{TST}}$
16: **else**
17: $\mathcal{P}_{\mathsf{PST}} \leftarrow \emptyset$
18: **for each** $dep \in \mathcal{P}_{deps_specialized}$ **do**
19: $\mathcal{P}_{dep} \leftarrow create_config_file(dep)$
20: **if** $build(\mathcal{P}_{dep}, \mathcal{P}_{bin}) ==$ `SUCCESS` **then**
21: $\mathcal{P}_{\mathsf{PST}} \leftarrow \mathcal{P}_{\mathsf{PST}} \cup dep$
22: **end if**
23: **end for**
24: **return** $\mathcal{P}_{\mathsf{PST}}$
25: **end if**

### 3.3 Implementation Details

DEPTRIM is implemented in Java as a MAVEN plugin that can be integrated into a project as part of the build pipeline, or be executed directly from the command line. This design facilitates its integration as part of the projects' CI/CD pipeline, leading to specialized binaries for deployment. At its core, DEPTRIM reuses the state-of-the-art static analysis of DEPCLEAN [18], located in the `depclean-core` module [29]. DEPTRIM adds unique features to this core static Java analyzer by modifying the bytecode within dependencies based on usage information gathered at compilation time, which is different from the complete removal of unused dependencies performed by DEPCLEAN. It uses the ASM Java bytecode analysis library to build a static call graph of `class` files of the compiled projects and their dependencies. The call graph registers usage towards classes, methods, fields, and annotations. For the deployment of the specialized dependencies, DEPTRIM relies on the `deploy-file` goal of the official `maven-deploy-plugin` from the Apache Software Foundation. For dependency analysis and manipulation, DEPTRIM relies on the `maven-dependency-plugin`. DEPTRIM

provides dedicated parameters to target or exclude specific dependencies for specialization, using their identifier and scope. DEPTRIM is open-source and reusable from the Maven Central repository. Its source code is publicly available at https://github.com/castor-software/deptrim.

## 4 EVALUATION

DEPTRIM is designed for use by developers in deployment environments. It can be run when the developer is ready to create a distribution version of their compiled project for production. Depending on the outcome of specialization, DEPTRIM potentially removes large portions of the `compile`-scope dependencies of the project. The output is a specialized distribution that developers should be ready to distribute to users. The evaluation described in this section is intended to assess that experience: we run DEPTRIM on a project, build the project with specialized dependencies to confirm that its behavior is not negatively impacted, and evaluate the extent to which our technique is effective. Our evaluation is guided by the following research questions:

**RQ1**. What are the opportunities for dependency specialization in real-world projects?
**RQ2**. To what extent can all the used dependencies be specialized and the project built correctly?
**RQ3**. What is the reduction in the number of classes in the dependency tree of the project after specialization?
**RQ4**. In what contexts is static dependency specialization not applicable?

### 4.1 Study Subjects

We evaluate DEPTRIM with 30 open-source projects collected from two data sources. The first source is the dataset of single-module Java projects made available by Durieux *et al.* [30]. This dataset contains 395 popular projects that build successfully with MAVEN, *i.e.* all their tests pass, and a compiled artifact is produced as a result of the build. We analyze the dependency tree of the projects in this dataset and select those that have at least one `compile`-scope dependency. This results in 13 projects. Additionally, we derive a second set of projects through the advanced search feature of GitHub. We filter repositories with a *pom.xml* file and rank the resulting Java MAVEN projects in descending order according to the number of stars. We rely on the number of stars as a proxy for popularity [31]. We systematically build the projects that contain at least one `compile`-scope dependency, and at least one test executed by the `maven-surefire-plugin` until we obtain 17 projects that build successfully with MAVEN. At the end of this curation process, we have a set of 30 study subjects with at least one `compile`-scope dependency, and an executable test suite with tests that pass.

Table 1 presents descriptive statistics for the 30 study subjects. For multi-module projects, we specify the MODULE we use for our experiments. Furthermore, we link to the COMMIT SHA of the version that we consider for the evaluation. The explicit documentation of PROJECT, MODULE, and COMMIT SHA ensure the reproducibility of our evaluation. The projects are well-known in the Java community and have between 155 and 20,488 STARS, for `commons-validator` and `flink` respectively. The median number of stars is 2,751. `flink` also has the maximum number of COMMITS, at 32,667, while the median number of commits across the study subjects is 2,544. Next, we report the number of lines of Java code (LOC) in each project, computed with the Unix command `cloc`. In total, the projects have more than 2 M LOC. The two projects with the largest number of lines of code are `CoreNLP` (605,561) and `checkstyle` (342,795), while the median LoC across the projects is 32,965. In the TESTS column we give the number of tests executed by the official `maven-surefire-plugin` in the projects. The median number of tests is 599. The two projects with the most tests are `jimfs` (5,834) and `checkstyle` (3,887).

The last 4 columns of Table 1 provide dependency-specific information. In the column #CD, we provide the number of `compile`-scope dependencies in each project, as resolved by MAVEN. In total, there are 467 `compile`-scope dependencies across the 30 projects, with a median number of 9 CDs and at least 2 CDs in each project. The maximum number of `compile`-scope dependencies is 56, in `Recaf`. The following columns present the number of CLASSES that are written by the developers of the PROJECT, and the number of third-party classes that come from its `compile`-scope dependencies (CD). The bytecode of each of these classes is analyzed by DEPTRIM in order to construct a static call graph of APIs usages between the projects and dependencies, as described in Section 3.2.1. In total, DEPTRIM analyzes the bytecode of 15,594 project classes, and 135,343 classes from third-party dependencies. `CoreNLP` has 3,932 project classes, the maximum in the dataset. The largest number of third-party classes is 17,512, in `OpenPDF`. In the last column of the table, we present the dependency classes to project classes ratio ($\text{RATIO}_{\mathcal{O}}$ in Equation 1).

$$\text{RATIO}_{\mathcal{O}} = \frac{\text{\#CD CLASSES}}{\text{\#PROJECT CLASSES}} \tag{1}$$

We find that, for 27 of these 30 notable projects, most of the code actually belongs to third-party dependencies. In fact, this ratio is as high as 206.7 $\times$ for the project `tablesaw`. Across our dataset, the ratio of the project classes to the dependency classes is 8.7 $\times$.

Recalling the example of `jacop` introduced in Figure 1, the corresponding row in Table 1 reads as follows: we select its latest release for our evaluation (SHA `6ed0cd0`), which has 1,302 commits, 93,170 lines of Java code, 210 tests, and has been starred by 202 users on GitHub. When `jacop` is compiled into a `JAR`, the number of classes from `jacop` is 833. On the other hand, its 9 `compile`-scope dependencies contribute 10.2 $\times$ more classes (*i.e.*, 8,487) in the packaged `JAR` when compared to the classes in the project (*i.e.*, 833).

### 4.2 Protocol for RQ1

With this research question, we quantify the potential for dependency specialization in the 30 projects described in Table 1. In order to do so, we use DEPCLEAN to identify and remove bloated dependencies from each project, ensuring that the project still builds. We report the number of `compile`-scope dependencies that are non-bloated, denoted as NBCD. Next, we present the total number of classes removed through dependency debloating (CLASSES REMOVED), and compute the ratio ($\text{RATIO}_{\mathcal{D}}$) between the remaining dependency classes and the project classes (per Equation 2). This

Table 1: Description of the study subjects considered for the evaluation of DEPTRIM. The table links to the project repository and SHA on GitHub, and lists the number of commits, stars, lines of Java code (LoC), tests, and the original number of `compile`-scope dependencies in the project (#CD). Also indicated are the number of project classes in each study subject, the total number of classes contributed by CDs, as well as the ratio between them (RATIO$_{\mathcal{O}}$).

| PROJECT | MODULE | COMMIT SHA | COMMITS | STARS | LOC | TESTS | #CD | CLASSES | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | PROJECT | CD | RATIO$_{\mathcal{O}}$ |
| checkstyle | - | 6ec5122 | 12,066 | 7,455 | 342,795 | 3,887 | 17 | 863 | 6,493 | 7.5 × |
| Chronicle-Map | - | 63c1c60 | 3,298 | 2,539 | 55,178 | 1,231 | 35 | 375 | 7,595 | 20.3 × |
| classgraph | - | 8a24786 | 5,307 | 2,366 | 32,151 | 170 | 2 | 261 | 224 | 0.9 × |
| commons-validator | - | 33ecc88 | 1,742 | 155 | 16,781 | 576 | 4 | 64 | 780 | 12.2 × |
| CoreNLP | - | 013556a | 17,012 | 8,802 | 605,561 | 1,374 | 32 | 3,932 | 9,121 | 2.3 × |
| flink | flink-java | 1d6e2b7 | 32,667 | 20,488 | 36,455 | 836 | 16 | 277 | 6,175 | 22.3 × |
| graphhopper | core | bff8747 | 6,211 | 3,978 | 66,119 | 2,460 | 18 | 631 | 5,474 | 8.7 × |
| guice | core | 5f73d8a | 2,026 | 11,730 | 49,697 | 979 | 10 | 460 | 2,474 | 5.4 × |
| helidon-io | openapi | 070f2bb | 2,707 | 2,929 | 7,729 | 30 | 36 | 32 | 4,002 | 125.1 × |
| httpcomponents-client | httpclient5 | cb8bdf7 | 3,424 | 1,269 | 42,920 | 669 | 5 | 493 | 1,153 | 2.3 × |
| immutables | gson | 413aa37 | 2,588 | 3,218 | 16,448 | 37 | 2 | 31 | 307 | 9.9 × |
| jacop | - | 6ed0cd0 | 1,302 | 202 | 93,170 | 210 | 9 | 833 | 8,487 | 10.2 × |
| java-faker | - | b0b9e6e | 834 | 3,914 | 8,429 | 579 | 4 | 107 | 503 | 4.7 × |
| jcabi-github | - | 462d724 | 2,764 | 276 | 33,542 | 684 | 20 | 312 | 3,921 | 12.6 × |
| jimfs | jimfs | 9ef38d1 | 508 | 2,234 | 15,558 | 5,834 | 9 | 124 | 3,560 | 28.7 × |
| jooby | jooby | 1c78357 | 4,702 | 1,523 | 20,154 | 122 | 22 | 320 | 6,945 | 21.7 × |
| lettuce | core | fc94fcb | 2,280 | 4,861 | 89,468 | 2,600 | 44 | 1,302 | 10,364 | 8.0 × |
| modelmapper | core | 03663ee | 721 | 2,090 | 21,769 | 618 | 6 | 210 | 2,700 | 12.9 × |
| mybatis-3 | - | 2655970 | 4,436 | 18,065 | 61,849 | 1,699 | 8 | 480 | 1,345 | 2.8 × |
| OpenPDF | - | bd0d458 | 1,296 | 2,573 | 76,397 | 35 | 35 | 484 | 17,512 | 36.2 × |
| pdfbox | pdfbox | af1ff57 | 11,147 | 1,852 | 97,175 | 654 | 7 | 754 | 6,836 | 9.1 × |
| pf4j | - | fd00c63 | 692 | 1,901 | 7,199 | 151 | 3 | 93 | 115 | 1.2 × |
| poi-tl | - | 71b5969 | 732 | 3,063 | 20,882 | 125 | 36 | 255 | 12,143 | 47.6 × |
| Recaf | - | a30dce0 | 2,275 | 4,530 | 31,277 | 274 | 56 | 538 | 10,769 | 20.0 × |
| RxRelay | - | 09428b5 | 81 | 2,473 | 2,405 | 64 | 2 | 16 | 1,758 | 109.9 × |
| scribejava | - | 7a6185b | 1,259 | 5,317 | 5,769 | 82 | 8 | 116 | 1,278 | 11.0 × |
| tablesaw | json | 80d5334 | 2,501 | 3,101 | 508 | 9 | 9 | 7 | 1,447 | 206.7 × |
| tika | tika-core | dd04a3e | 6,823 | 1,584 | 32,388 | 305 | 2 | 435 | 253 | 0.6 × |
| undertow | core | cce54c6 | 5,517 | 3,284 | 106,711 | 682 | 5 | 1,581 | 742 | 0.5 × |
| woodstox | - | 58bd89e | 325 | 180 | 60,476 | 868 | 5 | 208 | 864 | 4.2 × |
| TOTAL | 14 | 30 | 139,243 | 127,952 | 2,056,960 | 27,844 | 467 | 15,594 | 135,343 | 8.7 × |

data provides quantitative insights regarding the impact of dependency debloating to reduce the share of third-party code, and on the opportunity to reduce this share further via dependency specialization.

$$\text{RATIO}_{\mathcal{D}} = \frac{\#\text{CD CLASSES} - \#\text{CLASSES REMOVED BY DEPCLEAN}}{\#\text{PROJECT CLASSES}} \quad (2)$$

### 4.3 Protocol for RQ2

In order to answer RQ2, we attach DEPTRIM to the MAVEN build lifecycle of each of our study subjects. DEPTRIM is implemented as a MAVEN plugin, which facilitates this integration, as described in Section 3.3. This means that the non-bloated `compile`-scope dependencies in the dependency tree of each project are resolved, specialized, and deployed to the local MAVEN repository. DEPTRIM then attempts to build the project, *i.e.*, compile it and run its tests, with the goal of preparing a specialized dependency tree with the maximum number of specialized dependencies. Per Algorithm 1, DEPTRIM constructs either a totally specialized tree (`TST`), or a partially specialized tree (`PST`) that includes the largest number of specialized dependencies that preserve the build correctness. For each project, we report whether it builds with a `TST`. If it does not, we report the number of dependencies that DEPTRIM successfully specializes to prepare a `PST` (through the metric NBCD SPECIALIZED). The findings from this research question highlight the applicability of DEPTRIM on real-world MAVEN projects, and its ability to prepare minimal versions of these projects, by removing unused classes within non-bloated dependencies while passing the build.

### 4.4 Protocol for RQ3

After building each project successfully with a `TST` or a `PST` in RQ2, we report the total number of classes that are removed by DEPTRIM through the specialization of its non-bloated `compile`-scope dependencies (as CLASSES REMOVED). We also report the ratio of the remaining number of specialized dependency classes to the number of project classes (RATIO$_{\mathcal{S}}$ in Equation 3). We compare RATIO$_{\mathcal{S}}$ with RATIO$_{\mathcal{O}}$, *i.e.*, we

evaluate the reduction in the original ratio after specialization. This research question demonstrates the practical advantages of dependency specialization with DEPTRIM, specifically the reduction in the original proportion of third-party classes within the compiled project binary.

$$\text{RATIO}_{\mathcal{S}} = \frac{\text{\#NBCD CLASSES} - \text{\#CLASSES REMOVED BY DEPTRIM}}{\text{\#PROJECT CLASSES}} \quad (3)$$

### 4.5 Protocol for RQ4

While processing each project, DEPTRIM records the project build logs after dependency specialization (RQ2), as well as the number of classes removed from each non-bloated dependency (RQ3). We derive the answer for RQ4 by analyzing these logs. In some cases, all the classes in a non-bloated dependency are used by the project, leaving no room for specialization. We refer to such a dependency as a totally used dependency (TUD). We report the number of TUDs for each project, where DEPTRIM is not applicable by design. Another situation where DEPTRIM is not applicable is when a project uses dynamic features to access dependency classes (Section 3.2.1). While computing the `PST` for RQ2, DEPTRIM builds the project multiple times, each time with a single specialized dependency. In case of a failure when building the project with a specialized dependency, we report a compilation error or a test failure. For the assessment of the compilation results, we rely on the official `maven-compiler-plugin`. We consider the execution of the test suite to fail if there is at least one test reported within the sets of `Failures` or `Errors`, as reported by the official `maven-surefire-plugin`. With this research question, we gain insights regarding the existing challenges of dependency specialization with DEPTRIM. More generally, it contributes to the understanding of the limitations of static analysis with respect to specializing dependencies, in view of the dynamic features of Java.

### 4.6 Evaluation Framework

In order to run our experiments, we have designed a fully automated framework that orchestrates the execution of DEPTRIM, the creation of specialized dependency trees, the building of the projects with the specialized dependency trees, as well as the collection and processing of data to answer our research questions. Since DEPTRIM is implemented as a MAVEN plugin, it integrates within the MAVEN build lifecycle and executes during the `package` phase. The execution was performed on a virtual machine running Ubuntu Server with 16 cores of CPU and 32 GB of RAM. It took one week to execute the complete experiment with the 30 study subjects. This execution time is essentially due to multiple executions of the large test suites of our subjects: once with the original project; once after debloating dependencies with DEPCLEAN; once with the `TST` generated by DEPTRIM, and if we generate a `PST` for a project, we run the test suite once with each individually specialized tree and once with the final `PST`. The execution framework is publicly available on GitHub at castor-software/deptrim-experiments, and the raw data obtained from the complete execution is available on Zenodo at 10.5281/zenodo.7613554.

## 5 EXPERIMENTAL RESULTS

This section presents the results from our evaluation of DEPTRIM with the 30 Java projects described in Section 4.1. We evaluate the effectiveness of DEPTRIM in automatically specializing the dependency tree of these projects. The answers to the four RQs are summarized in Table 2.

### 5.1 RQ1: What are the opportunities for dependency specialization in real-world projects?

With this first research question, we set a baseline to assess the impact of dependency specialization regarding the reduction of the number of classes in third-party dependencies. To do so, we report the number of classes removed through state-of-the-art dependency debloating with DEPCLEAN, as described in Section 4.2. We report the ratio of third-party classes remaining after debloating, with respect to the number of classes in each project presented in Table 1.

For our 30 study subjects, the column NBCD in Table 2 denotes the number of `compile`-scope dependencies that remain after identifying and removing bloated dependencies with DEPCLEAN, over the original number of `compile`-scope dependencies in the project (column #CD in Table 1). In total, DEPCLEAN removes 71 bloated dependencies, with a median of 8 dependencies, across the 30 projects. DEPCLEAN removes 23 bloated dependencies from `OpenPDF`, which is the largest number of bloated dependencies for one project in our dataset. In total, DEPCLEAN removes 5,718 third-party classes when considering all the projects' `JAR` files. It is interesting to note that, for all the projects, dependency debloating removes 4.2 % of the total number of classes.

All projects have at least 2 NBCDs, while `Recaf` has the maximum number of NBCDs at 41. In 13 projects, such as `classgraph` and `commons-validator`, all the dependencies are used. Therefore, executing DEPCLEAN does not contribute to the removal of any class on those projects. On the other hand, we find that in 5 projects, the bloated dependencies do not contain `class` files at all, such as in the case of `flink`. This happens when a bloated dependency only contains assets, such as resource files, or is explicitly designed to avoid conflicts with other dependencies [32]. For example, the dependency `com.google.guava:listenablefuture` is present in the dependency tree of 5 projects, and it is intentionally empty to avoid conflicts with `guava` [33]. Another dependency, called `batik-shared-resources`, is included in the dependency tree of 2 projects, and only contains resource files. We investigate the nature of these resource files and find that they are dependency license statements and build-related metadata. Thus, the removal of such dependencies does not result in a build failure within the projects.

The column $\text{RATIO}_{\mathcal{D}}$ in Table 2 presents the ratio of the number of classes in the NBCDs to the original number of classes in the project (column PROJECT in Table 1). For 11 of the 30 projects, $\text{RATIO}_{\mathcal{D}}$ is less than $\text{RATIO}_{\mathcal{O}}$ from Table 1. This corresponds to cases where debloating results in fewer third-party classes in the compiled project `JAR`. For example, the removal of the 23 bloated dependencies from `OpenPDF` results in the maximum reduction in the number of classes (2,336). Consequently, $\text{RATIO}_{\mathcal{D}}$ for `OpenPDF` is 31.4 $\times$, which is 4.8 less than its $\text{RATIO}_{\mathcal{O}}$. The project with the highest

Table 2: Results from the evaluation of DEPTRIM with the case studies described in Table 1. For RQ1, the table indicates the number of non-bloated `compile`-scope dependencies (NBCD). These are the dependencies that are identified as used in the project by DEPCLEAN. The number of classes removed via debloating is listed in the column CLASSES REMOVED. $\text{RATIO}_{\mathcal{D}}$ represents the number of remaining third-party classes after debloating over the number of classes in the project. For RQ2, we highlight whether DEPTRIM builds a project with a `TST` or a `PST`, as well as the number of specialized NBCDs contained in the built project (NBCD SPECIALIZED). RQ3 presents the reduction in the number of classes in the NBCDs as a result of specialization with DEPTRIM, and correspondingly, in the $\text{RATIO}_{\mathcal{S}}$ of the third-party classes to project classes. For RQ4, we report the three cases where specialization with DEPTRIM is not applicable: (i) if all the classes in a non-bloated dependency are totally used by the project (TUD); (ii) if specializing a dependency causes a compilation error (COMP. ERROR) during project build; and (iii) if a test fails when building a project with a specialized dependency (TEST FAIL.).

| PROJECT | RQ1 | | | RQ2 | | | RQ3 | | RQ4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | NBCD | CLASSES REMOVED | $\text{RATIO}_{\mathcal{D}}$ | TST | PST | NBCD SPECIALIZED | CLASSES REMOVED | $\text{RATIO}_{\mathcal{S}}$ | TUD | COMP. ERROR | TEST FAIL. |
| `checkstyle` | 15/17 | 2 (0.0 %) | 7.5 × | ✗ | ✓ | 12/15 | 2,330 (35.9 %) | 4.8 × | 0/15 | 2/3 | 1/3 |
| `Chronicle-Map` | 28/35 | 278 (3.7 %) | 19.5 × | ✗ | ✓ | 22/28 | 4,670 (63.8 %) | 7.1 × | 4/28 | 1/2 | 1/2 |
| `classgraph` | 2/2 | 0 (0.0 %) | 0.9 × | ✓ | | 2/2 | 10 (4.5 %) | 0.8 × | 0/2 | – | – |
| `commons-validator` | 4/4 | 0 (0.0 %) | 12.2 × | ✓ | | 4/4 | 625 (80.1 %) | 2.4 × | 0/4 | – | – |
| `CoreNLP` | 30/32 | 364 (4.0 %) | 2.2 × | ✓ | | 29/30 | 3,648 (41.7 %) | 1.3 × | 1/30 | – | – |
| `flink` | 15/16 | 0 (0.0 %) | 22.3 × | ✗ | ✓ | 12/15 | 4,124 (66.8 %) | 7.4 × | 1/15 | 1/2 | 1/2 |
| `graphhopper` | 13/18 | 1,309 (23.9 %) | 6.6 × | ✗ | ✓ | 12/13 | 1,666 (40.0 %) | 4.0 × | 0/13 | 1/1 | 0/1 |
| `guice` | 9/10 | 0 (0.0 %) | 5.4 × | ✓ | | 7/9 | 1,327 (53.6 %) | 2.5 × | 2/9 | – | – |
| `helidon-io` | 34/36 | 38 (0.9 %) | 123.9 × | ✗ | ✓ | 32/34 | 1,040 (26.2 %) | 91.4 × | 1/34 | 1/1 | 0/1 |
| `httpcomponents-client` | 5/5 | 0 (0.0 %) | 2.3 × | ✓ | | 4/5 | 432 (37.5 %) | 1.5 × | 1/5 | – | – |
| `immutables` | 2/2 | 0 (0.0 %) | 9.9 × | ✓ | | 2/2 | 48 (15.6 %) | 8.4 × | 0/2 | – | – |
| `jacop` | 4/9 | 504 (5.9 %) | 9.6 × | ✗ | ✓ | 3/4 | 6,453 (80.8 %) | 1.8 × | 0/4 | 1/1 | 0/1 |
| `java-faker` | 4/4 | 0 (0.0 %) | 4.7 × | ✗ | ✓ | 3/4 | 222 (43.9 %) | 2.7 × | 0/4 | 1/1 | 0/1 |
| `jcabi-github` | 17/20 | 9 (0.2 %) | 12.5 × | ✗ | ✓ | 16/17 | 2,456 (62.8 %) | 4.7 × | 0/17 | 1/1 | 0/1 |
| `jimfs` | 8/9 | 0 (0.0 %) | 28.7 × | ✓ | | 6/8 | 1,741 (48.9 %) | 14.7 × | 2/8 | – | – |
| `jooby` | 20/22 | 0 (0.0 %) | 21.7 × | ✗ | ✓ | 19/20 | 1,349 (19.4 %) | 17.5 × | 0/20 | 1/1 | 0/1 |
| `lettuce` | 39/44 | 0 (0.0 %) | 8.0 × | ✗ | ✓ | 36/39 | 1,866 (18.0 %) | 6.5 × | 2/39 | 0/1 | 1/1 |
| `modelmapper` | 6/6 | 0 (0.0 %) | 12.9 × | ✗ | ✓ | 4/6 | 412 (15.3 %) | 10.9 × | 1/6 | 0/1 | 1/1 |
| `mybatis-3` | 8/8 | 0 (0.0 %) | 2.8 × | ✗ | ✓ | 7/8 | 422 (31.4 %) | 1.9 × | 0/8 | 0/1 | 1/1 |
| `OpenPDF` | 12/35 | 2,336 (13.3 %) | 31.4 × | ✗ | ✓ | 11/12 | 11,468 (75.6 %) | 7.7 × | 0/12 | 1/1 | 0/1 |
| `pdfbox` | 6/7 | 63 (0.9 %) | 9.0 × | ✓ | | 6/6 | 5,070 (74.9 %) | 2.3 × | 0/6 | – | – |
| `pf4j` | 3/3 | 0 (0.0 %) | 1.2 × | ✓ | | 2/3 | 10 (8.7 %) | 1.1 × | 1/3 | – | – |
| `poi-tl` | 33/36 | 258 (2.1 %) | 46.6 × | ✗ | ✓ | 27/33 | 5,192 (43.7 %) | 26.2 × | 5/33 | 0/1 | 1/1 |
| `Recaf` | 49/56 | 518 (4.8 %) | 19.1 × | ✓ | | 41/49 | 2,952 (28.8 %) | 13.6 × | 8/49 | – | – |
| `RxRelay` | 2/2 | 0 (0.0 %) | 109.9 × | ✗ | ✓ | 1/2 | 64 (3.6 %) | 105.9 × | 0/2 | 0/1 | 1/1 |
| `scribejava` | 7/8 | 39 (3.1 %) | 10.7 × | ✓ | | 6/7 | 353 (28.5 %) | 7.6 × | 1/7 | – | – |
| `tablesaw` | 9/9 | 0 (0.0 %) | 206.7 × | ✓ | | 7/9 | 379 (26.2 %) | 152.6 × | 2/9 | – | – |
| `tika` | 2/2 | 0 (0.0 %) | 0.6 × | ✓ | | 2/2 | 187 (73.9 %) | 0.2 × | 0/2 | – | – |
| `undertow` | 5/5 | 0 (0.0 %) | 0.5 × | ✓ | | 5/5 | 224 (30.2 %) | 0.3 × | 0/5 | – | – |
| `woodstox` | 5/5 | 0 (0.0 %) | 4.2 × | ✗ | ✓ | 3/5 | 222 (25.7 %) | 3.1 × | 0/5 | 1/2 | 1/2 |
| TOTAL | 396/467 | 5,718 (4.2 %) | 8.3 × | 14/30 | 16/30 | 343 (86.6 %) | 60,962 (47.0 %) | 4.4 × | 32/396 | 12/21 | 9/21 |

percentage of classes removed is `graphhopper`, for which the removal of 5 bloated dependencies leads to a 23.9 % reduction in the number of third-party classes. $\text{RATIO}_{\mathcal{D}}$ for `graphhopper` is 6.6 ×, down from its original $\text{RATIO}_{\mathcal{O}}$ of 8.7 ×. However, despite debloating dependencies, the total $\text{RATIO}_{\mathcal{D}}$ across the 30 projects is 8.3 × , which is only 0.4 less than the total original $\text{RATIO}_{\mathcal{O}}$.

Of the 9 `compile`-scope dependencies in `jacop` (column #CD in Table 1), DEPCLEAN identifies 5 dependencies as bloated and removes them. This leads to the removal of 504 classes, and 4 remaining NBCDs with a total of 7,983 classes. Correspondingly, $\text{RATIO}_{\mathcal{D}}$ reduces to 9.6 ×, down from the original $\text{RATIO}_{\mathcal{O}}$ of 10.2 ×. The 4 NBCDs are the target for specialization with DEPTRIM, which will remove unused classes within these dependencies while ensuring that `jacop` still correctly builds.

**Answer to RQ1:** State-of-the-art dependency debloating with DEPCLEAN contributes to the removal of 71 bloated dependencies from 30 real-world Java projects. This corresponds to the removal of 5,718 (4.2 %) third-party classes in total. Yet, the dependency classes to project classes ratio is reduced by only 0.4 (from 8.7 × to 8.3 × ). This calls for more extensive code removal to reduce the dependencies to the strictly necessary parts.

### 5.2 RQ2: To what extent can all the used dependencies be specialized and the project built correctly?

This research question evaluates the ability of DEPTRIM to perform automatic dependency specialization for the study subjects described in Section 4.1. We consider the specialization procedure to be successful if DEPTRIM produces a valid set of specialized dependencies, with a corresponding specialized dependency tree captured in a *pom.xml*, and for which the project builds correctly. To reach this successful state, the project to be specialized must pass through all the build phases of the MAVEN build lifecycle, *i.e.*, compilation, testing, and packaging, according to the protocol described in Section 4.2.

Columns TST, PST, and NBCD SPECIALIZED in Table 2 present the results obtained. First, we observe that for a total of 14 (46.7 %) projects, DEPTRIM produces a totally specialized tree (TST), *i.e.*, the project builds successfully with a specialized version of all its non-bloated compile-scope dependencies. For these projects, DEPTRIM successfully identifies and removes unused classes within the dependencies. Moreover, DEPTRIM updates the dependency tree of each project by replacing original dependencies with specialized ones. The projects correctly compile, and their original test suite still passes, indicating that their behavior is intact, despite the dependency tree specialization. Overall, these results confirm that dependency specialization is feasible for real-world projects.

We illustrate TSTs with the example of pdfbox, a utility library and tool to manipulate PDF documents. DEPTRIM specializes the 6 NBCDs of pdfbox, and builds its TST successfully. Of these 6 dependencies, 4 are direct: fontbox, commons-logging, pdfbox-io, and bcprov-jdk18on; whereas 2 are transitive: bcutil-jdk18on and bcpkix-jdk18on. Another interesting example is guice, a popular dependency injection framework from Google branded as a "lightweight" alternative to existing libraries, as stated in its official documentation. DEPTRIM builds guice with a TST, thus making it even smaller. The project that builds with a TST and has the largest number of specialized dependencies is Recaf with 41 specialized dependencies. Note that, when specializing transitive dependencies, DEPTRIM keeps all the classes in the direct dependencies that are necessary to access the APIs in transitive dependencies, whether directly or indirectly. For example, the transitive dependency commons-lang3 in Recaf is resolved and used from the direct dependency jphantom, a Java library for program complementation [34]. Thus, DEPTRIM keeps the bytecode in commons-lang3 that is necessary to access the used features provided by jphantom.

On the other hand, projects that do not build with a TST signify cases where at least one compile-scope dependency relies on dynamic Java features that make static analysis unsound. This observation is in line with previous work showing that Java reflection and other dynamic features impose limitations on performing static analysis in the Java ecosystem [35]. However, even for these projects, DEPTRIM successfully builds a partially specialized tree (PST) by targeting dependencies that are safe to specialize and discarding the ones that are unsafe. In total, 16 (53.3 %) of the projects build successfully with a PST. In these cases, DEPTRIM successfully identifies the subset of dependencies that are safe for specialization, and validates that the projects still correctly build with a PST.

Table 3: Dependencies specialized in the OpenPDF project

| DEPENDENCY | TYPE | CLASSES REMOVED |
|---|---|---|
| xmlgraphics-commons | Transitive | 366/375 (97.6 %) |
| bcutil-jdk18on | Transitive | 532/579 (91.9 %) |
| fop-core | Transitive | 2,278/2,547 (89.4 %) |
| bcpkix-jdk18on | Direct | 697/841 (82.9 %) |
| bcprov-jdk18on | Direct | 5,696/7,149 (79.7 %) |
| xml-apis | Transitive | 234/346 (67.6 %) |
| fontbox | Transitive | 100/157 (63.7 %) |
| xalan | Transitive | 942/1,501 (62.8 %) |
| icu4j | Direct | 578/1,555 (37.2 %) |
| serializer | Transitive | 39/108 (36.1 %) |
| commons-logging | Transitive | 6/18 (33.3 %) |
| fop | Transitive | N/A |
| TOTAL | 3 D/9 T | 11,468/15,176 (75.6 %) |

For example, DEPTRIM successfully specializes 4 dependencies in the project modelmapper, an object mapping library that automatically maps objects to each other. DEPTRIM creates a PST with which modelmapper builds successfully. Note that none of the 6 compile-scope dependencies of modelmapper are bloated, and hence debloating the project with DEPCLEAN has no impact on it. However, after executing DEPTRIM, the direct dependencies objenesis and asm-tree are specialized. Moreover, the transitive dependencies asm-commons and asm, resolved from asm-tree are also specialized. This example illustrates the impact of specialization beyond dependency debloating, for projects that build with a PST. Indeed, across our study subjects, there are 8 projects that successfully build with a TST, and 5 that build with a TST, and yet for which no classes are removed through DEPCLEAN.

It is interesting to notice that some of our study subjects share dependencies that are specialized. The three dependencies that are most frequently specialized are slf4j-api (13 projects), jsr305 (9 projects), and commons-io (6 projects). The projects jcabi-github, jooby, and Recaf include all these three dependencies in their dependency tree. After investigating the contents of the specialized versions of slf4j-api prepared by DEPTRIM, we find that there are three sets of variants for which this dependency contains the same number of classes. Thus, deploying multiple specialized versions of slf4j-api to external repositories can contribute to reducing its attack surface for projects that reuse the exact same features. This specialized form of code reuse also increases software diversity. Furthermore, the dependency bcprov-jdk18on, which contains the largest number of classes among the dependencies (3,768), is successfully specialized in 2 projects, OpenPDF and pdfbox. Our findings suggest that specializing dependencies with a large number of classes yields a greater reduction of third-party code. To confirm this hypothesis, additional investigation is required.

Our experiments show that, despite the challenges of specializing the dependency trees of our 30 real-world study subjects, DEPTRIM is capable of specializing 343 of the 396 non-bloated compile-scope dependencies across them. A key aspect of our evaluation is that we validate that each

project builds successfully using its specialized dependency tree. We manually analyze and classify the cases where specialization is not achievable for a dependency, in RQ4. The specialized dependencies contribute to the deployment of smaller project binaries, to reduce their attack surface, and to increase dependency diversity when deployed to external repositories.

> **Answer to RQ2:** DEPTRIM successfully builds 14 real-world projects with a totally specialized dependency tree. For the other 16 projects, DEPTRIM finds the largest subset of specialized dependencies that do not break the build. In total, DEPTRIM specializes 343 (86.6 %) of the non-bloated `compile`-scope dependencies across the projects. This is evidence that a large majority of dependencies in Java projects can be specialized without impacting the project build.

### 5.3 RQ3: What is the reduction in the number of classes in the dependency tree of the project after specialization?

To answer our third research question, we count the number of classes removed by DEPTRIM in the 343 successfully specialized dependencies. The goal is to evaluate the effectiveness of DEPTRIM in removing unused `class` files through specialization, as described in Section 4.3. We also report the impact of this reduction on the ratio of third-party classes to project classes, *i.e.*, $\text{RATIO}_{\mathcal{S}}$.

The column CLASSES REMOVED in Table 2 shows the number of classes removed by DEPTRIM from the NBCDs of each project, in order to build its `TST` or `PST`. DEPTRIM removes a total of 60,962 classes, with a median removal of 1,184 third-party classes for each project. This represents 47.0 % of the total number of classes in the third-party dependencies for all the projects (*i.e.*, 135,343 per Table 1). For example, the project `tika` has 2 dependencies specialized: `slf4j-api` with 9/52 (17.3 %) classes removed, and `commons-io` with 178/201 (88.6 %) classes removed. This represents a removal of 187 (73.9 %) third-party classes in `tika`, as a result of which its $\text{RATIO}_{\mathcal{S}}$ is $0.2\times$. Thus, the ratio of dependency classes to project classes in `tika` decreased by 0.4 compared to $\text{RATIO}_{\mathcal{D}}$ (*i.e.*, $0.6\times$).

The project with the highest percentage of dependency classes removed is `jacop` with 80.8 %, *i.e.*, 6,453 of the 8,487 original third-party classes. This drastic reduction is a consequence of removing unused classes from large `compile`-scope dependencies such as `scala-compiler`, `scala-library`, and `scala-reflect`, which embed the standard library, compiler, and language reflection features of the Scala programming language. DEPTRIM removes 2,982, 1,371, and 1,351 classes in these dependencies, respectively, as well as 749 classes from `mockito-all`. In fact, DEPTRIM identifies that only 2 of the 2,984 classes in `scala-compiler` are required by `jacop`. These classes are `SourceReader` and its nested class `SourceReader$`, which provide functionalities to read and decode Scala source files. The 2,982 classes removed from this dependency provide tools for reflection, type-checking, or transformation, which are not necessary for `jacop`.

The project with the largest number of classes removed (11,468) is `OpenPDF`. Table 3 shows the dependencies specialized in `OpenPDF`, of which 3 are direct and 9 are transitive. DEPTRIM builds `OpenPDF` with a `PST`, excluding the dependency `fop` from the specialized dependency tree. Looking at the 11 successfully specialized dependencies, we observe that `OpenPDF` depends transitively on a family of dependencies from the Apache XML Graphics Project, including `fop-core` and `xmlgraphics-commons`, from which DEPTRIM removes 2,278 (89.4 %) and 366 (97.6 %) unused classes, respectively. `OpenPDF` also depends directly on `bcpkix-jdk18on` and `bcprov-jdk18on`, which are dependencies from the Bouncy Castle family of cryptographic libraries, from which 697 (82.9 %) and 5,696 (79.7 %) classes are removed, respectively. Moreover, DEPTRIM systematically identifies functionalities that are used transitively through direct dependencies. For example, two classes within `OpenPDF`, called `PdfPKCS7` and `TSAClientBouncyCastle`, use classes from the direct dependency `bcpkix-jdk18on`. In turn, these classes of `bcpkix-jdk18on` depend on 4 classes within `bcutil-jdk18on` that are responsible for supporting the encoding of the Time Stamp Protocol. Therefore, DEPTRIM marks these 4 classes within the transitive dependency as necessary for `OpenPDF`, and does not remove them. Note that `OpenPDF` built with a specialized dependency tree may be deployed to an external repository, which reduces the attack surface of the clients that rely on the features that are provided by `OpenPDF` when used as a library.

We observe that significant removal of unnecessary bytecode within dependency `JAR` files through specialization is achievable and also beneficial for the projects. Smaller binaries reduce overhead when the `JAR` files are deployed and shipped over the network. Specialized dependencies reduce the build time, which increases productivity and reduces maintenance efforts. Furthermore, the reduction in the number of third-party classes has a positive impact on minimizing the attack surface of the projects.

> **Answer to RQ3:** DEPTRIM reduces the number of classes in the dependency tree of the 30 projects by 47.0 %. The dependency classes to project classes ratio is almost halved (from $8.3\times$ to $4.4\times$). This result confirms the relevance of this dependency specialization approach in drastically reducing the share of third-party bytecode in Java projects.

### 5.4 RQ4: In what contexts is static dependency specialization not applicable?

With this research question we report on the cases where there is no scope for specialization in a non-bloated dependency, as well as cases where projects do not build successfully with a specialized dependency in the dependency tree.

First, we observe that 14 projects include at least one dependency that is totally used. A total of 32 dependencies are totally used by their respective client projects, as presented in the column TUD in Table 2. A dependency is a TUD for a project if all its `class` files are exercised by the project. Consequently, there is no scope for the specialization of a TUD. TUDs represent 8.1 % of the non-bloated dependencies. `Recaf` has 8 TUDs, the largest number in the study subjects. Note that a project with TUDs in its dependency tree can still successfully build with a `TST`, as is true for 8 projects, including `Recaf`. We observe that the dependencies `asm-tree`,

Table 4: Number of unique failing tests, and the specialized dependency that causes these failures, in the 9 projects with tests failures

| PROJECT | DEPENDENCY | # TEST FAIL |
|---|---:|---:|
| checkstyle | Saxon-HE | 1/3,887 (0.0 %) |
| Chronicle-Map | chronicle-wire | 3/1,231 (0.2 %) |
| flink | commons-math3 | 1/820 (0.1 %) |
| lettuce-core | micrometer-core | 6/2,600 (0.2 %) |
| modelmapper | byte-buddy-dep | 4/618 (0.6 %) |
| mybatis-3 | slf4j-api | 1/1,699 (0.1 %) |
| poi-tl | commons-io | 107/125 (85.6 %) |
| RxRelay | rxjava | 6/64 (9.4 %) |
| woodstox | msv-core | 1/868 (0.1 %) |
| TOTAL | 9 | 130/11,912 (1.1 %) |

failureaccess, and minimal-json are TUDs in 2 projects. For example, minimal-json is totally used by both tablesaw and by Recaf, which is evidence of a minimal API that is completely used by these projects. As far as we know, this is the first time in the literature that totally used dependencies are identified and quantified.

DEPTRIM builds 16 projects with a PST. For example, DEPTRIM marks 1 of the 4 non-bloated compile-scope dependencies in java-faker as unsafe for specialization. This is because building java-faker with the specialized version of org.yaml:snakeyaml prevents the compilation of the project, which includes the Lebowski class. Consequently, org.yaml:snakeyaml is excluded from the specialized dependency tree of java-faker and DEPTRIM outputs a partially specialized tree that successfully builds java-faker and includes three specialized dependencies.

Column COMP. ERROR in Table 2 shows the number of specialized dependencies for which the build fails due to compilation errors. This occurs for 11 projects and 12 dependencies. We investigate the causes of compilation errors by manually analyzing the logs of the maven-compiler-plugin. We find the following 4 causes for compilation to fail:

- Some classes are not found during compilation. For example, 2 PSTs of checkstyle fail due to the missing classes, BasicDynaBean and ClassPath, from dependencies commons-beanutils and guava, respectively. Both classes enable dynamic scanning and loading of classes at runtime.
- The project has a plugin that fails at compile time. For example, the plugin snakeyaml-codegen-maven-plugin in the project helidon adds code to the project's compiled sources automatically [36], and fails when building with the specialized dependency smallrye-open-api-core because the specialization process changes the expected dependency bytecode.
- The project has a plugin that checks the integrity of the specialized dependency. For example, the dependency commons-io in project jcabi-github uses the maven-enforcer-plugin to check for certain constraints, including checksums, on the dependency bytecode.
- The specialized dependency is not found in the local repository. For example, the specialized dependency snakeyaml in project java-faker is not deployed correctly due to a known issue in this dependency when using the android MAVEN tag classifier [37].

We now discuss the number of specialized dependencies for which the build reports test failures (column TEST FAIL. in Table 2). For 9 projects, one specialized dependency has at least one test failure. DEPTRIM preserves the original behavior (*i.e.*, all the tests pass) of 356 (97.8 %) specialized, non-bloated compile-scope dependencies. This high rate of test success is a fundamental result to ensure that the specialized version of the dependency tree preserves the behavior of the project.

In total, we execute 27,844 unique tests across all projects (per Table 1). Of these, 130 do not pass. DEPTRIM produces specialized dependency trees that break a few test cases. These cases reveal the challenges of dependency specialization concerning static analysis. For example, DEPTRIM can miss some used classes, resulting in the removal of bytecode that is necessary at runtime. This is a general constraint for static analysis tools when processing Java applications that rely on dynamic features to load and execute code at runtime. As a result of the absence of bytecode from a specialized dependency, 9 projects report test failures, *e.g.*, an unreachable class loaded at runtime causing a failing test that stops the execution of the build.

Table 4 shows the number of unique test failures (column #TEST FAIL.) in the 9 projects that have at least one PST with test failures, as well as the specialized dependency that causes the failure (column DEPENDENCY). For example, the project Chronicle-Map has 3 tests that fail when specializing the dependency chronicle-wire, from a total of 1,231 executed tests, which represents 0.2 % of the total. The project with the largest number of test failures is poi-tl, with 107 (85.6 %) tests failures when specializing its dependency commons-io. Overall, the number of test failures accounts for 1.1 % of the total tests executed in the 9 projects, and only 0.5 % across the 30 projects.

We further investigate the causes of the failures. To do so, we manually analyze the logs of the tests, as reported by the maven-surefire-plugin. We find the following 3 causes:

- The tests load dependency classes dynamically. For example, poi-tl relies on the method byte[] in class IOUtils of commons-io to check the size of a file. This method is loaded via reflection through an external configuration file and causes the failure of 107 tests.
- Some tests rely on Java serialization to manipulate objects at runtime, and the input stream is not closed properly because DEPTRIM removes a class responsible for closing the input stream. For example, the project Chronicle-Map uses the dependency chronicle-wire for serialization, and 3 tests fail due to a ClosedIORuntimeException.
- The project has tests that rely on dependencies that use Java Native Interfaces (JNI) to execute machine code at runtime. For example, the test TestWsdlValidation in project woodstox relies on dependency msv-core which uses JNI to validate XML schemas. DEPTRIM's static analysis is limited to Java bytecode, and therefore native code executed in third-party dependencies is not considered as used when building the call graph.

Our results reveal the challenges of dependency specialization based on static analysis (see Section 3.2.1) for real-

world Java projects. Handling these cases to achieve 100 % correctness requires specific domain knowledge of the project, and of the reachable code in the dependencies that exercise some form of dynamic Java features. To facilitate this task, we provide a dedicated parameter `ignoreDependencies` in DEPTRIM so that developers can declare a list of dependency coordinates to be ignored by DEPTRIM during the call graph analysis. Nevertheless, we recommend always checking that the build passes to avoid semantic errors when performing bytecode removal transformations.

**Answer to RQ4:** Of the 396 dependencies that are targets for specialization, 32 are not specialized because they are totally used, 12 (3.3 %) dependencies cause a compilation failure when specialized, and 9 (2.5 %) lead to a failure at runtime. For the latter, the test failures represent only 0.5 % of the total number of tests executed. This behavioral assessment of DEPTRIM demonstrates that the specialized dependency trees preserve a large majority of syntactic and semantic correctness for the 30 projects.

## 6 DISCUSSION

In this section, we discuss the state-of-the-art and the current challenges of code specialization in Java, as well as the implications of specialization for software integrity. We also discuss the threats to the validity of our results.

### 6.1 Specialization in the Modern Java Ecosystem

The Java community is currently making substantial efforts to reduce the amount of unnecessary third-party code that ends up being deployed in production, as a consequence of dependencies. The GraalVM native image compiler [38] is perceived by many as an important step in this direction. GraalVM relies on static analysis to build a native executable image that only includes the elements reachable from an application entry point and its third-party dependencies [39]. To do so, GraalVM operates with a closed-world assumption [40]. This means that all the bytecode in the application, and their dependencies that can be called at runtime, must be known at build time, *i.e.*, when the native-image tool in GraalVM is building the standalone executable [41]. While building the image, GraalVM performs a set of aggressive optimizations such as the elimination of unused code from third-party dependencies. Consequently, the self-contained native executable image only includes code that is actually necessary to build and execute a Java project. This reduces the size of container images, making them more performant. The resulting images are ideal for the cloud, making Java applications easy to ship and deploy directly in a containerized environment, as microservices for example.

On the other hand, the reachability of some bytecode elements (such as classes, methods, or fields) may not be identified due to the Java dynamic features, *e.g.*, reflection, resource access, dynamic proxies, and serialization [42], [43], [35], [44]. For example, the popular dependency `netty`, an asynchronous event-driven framework, heavily relies on dynamic Java features to perform blocking and non-blocking sockets between servers and clients. As in DEPTRIM, the closed-world constraint of GraalVM imposes strict limits on the natural dynamism of Java, particularly on the run-time reflection and class-loading features, upon which so many existing libraries and frameworks such as `netty` depend. There is a risk of violating the close world assumption if at least one of the dependencies in the dependency tree of a project relies on some dynamic Java feature.

The community is creating new versions of libraries that adhere to the closed-world assumption. Rather than embracing the closed-world constraint in its entirety, the Java community instead pursues a gradual, incremental approach along this spectrum of limitations. By starting with small and simple modifications to Java Platform Specification, the community aims to establish a strong grasp of the necessary changes while maintaining key values of the Java programming language, such as readability, compatibility, and generality. In the long run, Java will likely embrace the full closed-world constraint in order to produce fully-static images. Between now and then, however, the community works on developing and delivering incremental improvements which developers can use sooner rather than later. A notable effort is the GraalVM Reachability Metadata Repository [45], which enables native image users to share and reuse metadata for common libraries and frameworks in the Java ecosystem, and thus simplify the maintenance of third-party dependencies. The repository is integrated with GraalVM Native Build Tools beginning with version `0.9.13`, and integrates with notable Java web frameworks such as Spring Boot 3.0, Quarkus, and Micronaut [46]. Despite the widespread popularity of Java libraries, there remains inadequate support for them. DEPTRIM offers a solution for projects that have dependencies potentially conflicting with the closed-world assumption by specializing their dependency tree. With the creation of a partially specialized tree (PST), DEPTRIM effectively achieves dependency specialization without jeopardizing the success of the build, making it a practical option.

### 6.2 Specialization and Software Integrity

The integrity of software supply chains is a timely research topic [47], [48], [49]. Ensuring the integrity of dependencies involves checking that their code has not been tampered with between the moment they are fetched from a repository and the moment they are packaged in the project. Checksums, such as the SHA family of cryptographic functions, are commonly used to verify the integrity of software dependencies. For example, when a software dependency is deployed, a SHA checksum is generated for the dependency, which is a unique representation of its binary content. Then, the clients of the dependency can recompute the checksum at build time to check the integrity of the dependency they are packaging in their dependency tree. A modern project may also use a software bill of materials (SBOM) that lists all the components that compose it, including open-source libraries, frameworks, and tools [50]. A comprehensive, well-maintained SBOM can help ensure software integrity by enabling organizations to identify and track potential security vulnerabilities in their software components and take appropriate action to address them, while also complying with regulations and standards.

The specialization of third-party dependencies modifies the bytecode of the target dependencies, which can break the

integrity-checking process. This is because the checksum of the original bytecode, which was used to verify the integrity of the dependency, will no longer match the checksum of the changed bytecode. For example, Listing 1 shows a JSON file reporting the checksum of the original dependency `commons-io` in one of our study subjects, `jcabi-github`, when using the SHA-256 hashing algorithm. DEPTRIM specializes `commons-io` by removing unused classes, which constitutes a change in its bytecode, and hence in its checksum, as presented in Listing 2. Therefore, the checksum of the changed bytecode after specialization no longer matches the expected checksum, and the integrity checks fail, as discussed in Section 4.5.

A way to ensure the integrity of specialized dependencies is by deploying them to external repositories at build time. For example, in the previous example, the project `jcabi-github` could deploy the specialized variant of `commons-io` to Maven Central with a custom MAVEN `groupId`, while updating the checksum in its SBOM accordingly. This way, it could check the integrity of this dependency against the SHA of the specialized variant. This approach provides the benefits of specialization while preserving software integrity. As far as we know, there is currently no tool that implements this technique. Preserving integrity in the light of specialization is currently a challenge for hardening the software supply chain, and a promising research direction.

```
{
  "groupId": "commons-io",
  "artifactId": "commons-io",
  "version": "2.11.0",
  "checksumAlgorithm": "SHA-256",
  "checksum": "961b2f6d87dbacc5d54abf45ab7a6e2495f89b755989
      ↪ 62d8c723cea9bc210908"
}
```

Listing 1: SHA checksum of the original dependency `commons-io` in the project `jcabi-github`

```
{
  "groupId": "se.kth.castor.deptrim.spl",
  "artifactId": "commons-io",
  "version": "2.11.0",
  "checksumAlgorithm": "SHA-256",
  "checksum": "c84eaef6b629729c71a70a2513584e7ccacf70cb4df1
      ↪ 3e38b731bb6193c60e73"
}
```

Listing 2: SHA checksum of the specialized dependency `commons-io` in the project `jcabi-github`

### 6.3 Threats to Validity

We now discuss the threats to the validity of the evaluation of DEPTRIM, and how we address them.

***Internal validity.*** The first internal threat relates to the usage of static analysis to determine which parts of the dependency bytecode are reachable from the project. We mitigate this threat, by relying on DEPCLEAN, the state-of-the-art tool for debloating Java dependencies [25]. Another threat lies in the thoroughness of the test suite. The test suite may not capture all the dependency API behaviors that can be exercised by the project. To mitigate this threat, we curate a set of study subjects that are mature and contain tests (see Table 1). DEPTRIM is a MAVEN plugin that modifies the *pom.xml* on-the-fly during the build process. It might introduce conflicts between plugins, causing the build to fail. For example, `maven-enforcer-plugin` or `license-maven-plugin` check the *pom.xml* to ensure that it meets specific requirements and follows the best practices. However, since our approach only modifies the code within the entry `dependencies` in the *pom.xml*, the failures due to misconfigurations are minimized.

***External validity.*** Our results are representative of the Java ecosystem, and our findings are valid for software projects with these particular characteristics. Moreover, our bytecode removal results are influenced by the number of dependencies of these projects. To address this, we found our evaluation on 30 real-world, well-known projects, derived from sound data sources, as described in Section 4.1. Furthermore, the selected projects cover a variety of application domains (*e.g.*, dependency injection, database handling, machine learning, encryption, IO utilities, faking, meta-programming, networking, etc). To the best of our knowledge, this is the largest set of study subjects used in software specialization experiments.

***Construct validity.*** The threats to construct validity relate to the accuracy and soundness of the results. Our results may not be reproducible if the projects are compiled with a different Java version or have flaky tests. To mitigate this threat, we choose the latest Java version and build the original projects two times in order to avoid including projects with flaky tests. Furthermore, for all RQs, we include logs and automated analysis scripts in our replication package for reproducibility as described in Section 4.6.

## 7 RELATED WORK

In this section, we position the contribution of our dependency specialization technique with respect to previous work that aims at reducing the size of applications composed of multiple third-party dependencies.

Several previous works focus on reducing the size of Java applications. While all techniques perform code analysis based on the construction of a call graph, they vary in the way they look for code that can be removed: dead-code removal, inlining, and class hierarchy removal [51]; identification and removal of unused optional concerns with respect to a specific installation context [52], unbundling user-facing application features [53] or tailoring the Java standard library [54], [55]. In contrast to these efforts that aim at reducing the size of a packaged application, DEPTRIM targets reduction while keeping the modular structure of the project and its third-party dependencies. Our technique focuses on reducing each dependency while keeping an explicit dependency tree in the form of a specialized *pom.xml* file as well as maintaining specialized dependencies as distinct, deployable `JAR` files.

Bruce *et al.* [56] propose JSHRINK, augmenting static reachability analysis with dynamic reachability analysis. They rely on test cases to find dynamic features, including methods and fields, invoked at runtime, adding them back to amend the imprecision of static call graphs. DEPTRIM differs from JSHRINK, as it does not aim to refine reachability

analysis to create smaller JAR files of the target project. Instead, DEPTRIM focuses on specializing the dependency tree of a Java project by removing unused code in third-party dependencies independently, such that each dependency can be deployed to external repositories.

In our previous work, we proposed DEPCLEAN, a tool that identifies and removes unused dependencies in the dependency tree [57], [26]. DEPCLEAN constructs a call graph of the bytecode class members by capturing annotations, fields, and methods, and accounts for a limited number of dynamic features such as class literals. DEPCLEAN produces a variant of the dependency tree without bloated dependencies. DEPTRIM pushes forward the field of dependency debloating through the removal of unused bytecode from individual dependencies, thereby yielding smaller packaged artifacts.

Closely related to DEPTRIM is the work on code specialization. Mishra and Polychronakis propose SHREDDER [22], a defense-in-depth exploit mitigation tool that protects closed-source applications against code reuse attacks. They also build SAFFIRE [58] which creates specialized and hardened replicas of critical functions with restricted interfaces to prevent code reuse attacks. These tools target C++ API implementations. They eliminate arguments with static values and restrict the acceptable values of arguments. A key feature of these techniques is to replace the code of API members by a stub function so that, at runtime, only specialized versions of critical API functions are exposed, while any invocation that violates the enforced policy is blocked. Focusing on JavaScript applications, Turcotte *et al.* [59] propose STUBBIFIER, which replaces unreachable code, identified through static and dynamic call graphs. DEPTRIM does not remove unused code from the project but rather replaces dependencies that are partially used by the project with smaller and specialized versions.

Previous specialization techniques mitigate the risk of removing code that might be needed for a specific execution, by replacing this code by small stub functions. With DEPTRIM we address the challenges of dynamic language features with another strategy. We specialize each dependency and then assess whether the completely specialized dependency tree still passes the build. If it does not, we search for a partially specialized tree that does not include the dependencies that rely on the dynamic features of Java. To the best of our knowledge, prior research on software specialization has not addressed the customization of third-party dependencies or the provision of build configuration files to enable the construction of specialized dependency trees. This represents a novel contribution of our work, differentiating it from previous studies in this area.

As part of our experiments with DEPTRIM, we contribute novel observations to the body of knowledge about library and API usage. Recent work in this area includes the following studies. Huang *et al.* [60] study the usage intensity from Java projects to libraries. They find that the number of libraries adopted by a project is correlated to the project size. However, their study does not provide a more fine-grained analysis of the used components. Hejderup *et al.* [21] investigate the extent to which Rust projects use the third-party packages in their dependency tree. They propose PRÄZI, a call-based dependency network for CRATES.IO that operates at the function level.

Some studies examine the benefits of debloating from a security standpoint. For instance, Azad *et al.* [61] report that debloating significantly reduces the number of vulnerabilities in web applications, while also making it more difficult for attackers to exploit the remaining ones. Agadakos *et al.* [62] propose NIBBLER to erase unused functions within the binaries of shared libraries at the binary level. This enhances existing software defenses, such as continuous code re-randomization and control-flow integrity, without incurring additional run-time overhead. Although DEPTRIM's primary function is to specialize dependency trees and enhance their reusability, it is important to note that the removal of third-party code can lead to a reduction in the potential attack surface.

# 8 CONCLUSION

In this paper, we propose DEPTRIM, a fully automated technique to specialize third-party packages in the dependency tree of a Java project. DEPTRIM systematically identifies and removes unused classes within each reachable dependency, repackages the used classes into a specialized dependency, and replaces the original dependency tree of a project with a specialized version. The goal of DEPTRIM is to build a minimal project binary, which only contains code that is useful for the project.

Our evaluation with $30$ real-world java projects that build with MAVEN demonstrates the capabilities of DEPTRIM to produce minimal versions of the dependencies in these projects while keeping the original build successful. In particular, DEPTRIM builds totally specialized trees for $14$ projects and builds the other $16$ with the largest number of specialized dependencies such that the project still builds. The ratio of dependency classes to project classes decreases from $8.7\times$ in the original project to $4.4\times$ in the specialized project, which represents a reduction of $47\,\%$ in the total third-party classes. Our findings reveal that dependency specialization is an effective means of reducing the share of third-party code in Java projects.

An important direction for future research is to investigate the application of dependency specialization to increase the diversity of software supply chains. DEPTRIM currently generates one specialized dependency tree for each project. However, there exists a multitude of possibilities within the realm of partially specialized trees, which we have yet to explore. As future work, we will venture into the forest of trees in between, with diverse combinations of specialized dependencies. By doing so, we aim to achieve an effective moving target defense against software supply chain attacks.


# ACKNOWLEDGEMENTS

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, as well as by the TrustFull and the Chains projects funded by the Swedish Foundation for Strategic Research.

## REFERENCES


[1] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[2] R. Cox, "Surviving software dependencies," *Communications of the ACM*, vol. 62, no. 9, pp. 36–43, 2019.

[3] T. Gustavsson, "Managing the open source dependency," *IEEE Computer*, vol. 53, no. 2, pp. 83–87, 2020.

[4] N. Harutyunyan, "Managing your open source supply chain-why and how?," *IEEE Computer*, vol. 53, no. 6, pp. 77–81, 2020.

[5] P. Ombredanne, "Free and open source software license compliance: tools for software composition analysis," *IEEE Computer*, vol. 53, no. 10, pp. 105–109, 2020.

[6] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110653, 2021.

[7] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *Proc. of ICSME*, pp. 404–414, 2018.

[8] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transacions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, 2022.

[9] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proc. of ACM CCS*, pp. 380–394, 2018.

[10] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?," *IEEE Transacions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, 2022.

[11] A. Dann, H. Plate, B. Hermann, S. E. Ponta, and E. Bodden, "Identifying challenges for OSS vulnerability scanners - A study & test suite," *IEEE Transacions on Software Engineering*, vol. 48, no. 9, pp. 3613–3625, 2022.

[12] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transacions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2022.

[13] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proc. of ESEM*, pp. 1–11, 2021.

[14] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *Proc. of MSR*, pp. 254–265, 2021.

[15] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, "The go programming language and environment," *Communications of the ACM*, vol. 65, no. 5, pp. 70–78, 2022.

[16] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: software signing for everybody," in *Proc. of ACM CCS*, pp. 2353–2367, 2022.

[17] P. Pashakhanloo, A. Machiry, H. Choi, A. Canino, K. Heo, I. Lee, and M. Naik, "Pacjam: Securing dependencies continuously via package-oriented debloating," in *Proc. of ACM CCS*, pp. 903–916, 2022.

[18] S. E. Ponta, W. Fischer, H. Plate, and A. Sabetta, "The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application," in *Proc. of ICSME*, pp. 555–558, IEEE, 2021.

[19] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in NPM," in *Proc. of ASE*, pp. 1 – 12, 2022.

[20] A. A. Sawant and A. Bacchelli, "fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1348–1371, 2017.

[21] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–42, 2022.

[22] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *Proc. of ACSAC*, pp. 1–16, 2018.

[23] A. S. Foundation, "Apache Maven," January 2023. Available at https://maven.apache.org.

[24] Apache Maven, "Introduction to the dependency mechanism," January 2023. Available at https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[25] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021.

[26] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proc. of ESEC/FSE*, pp. 1021–1031, 2021.

[27] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?," in *Proc. of ESEC/FSE*, pp. 319–330, 2018.

[28] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proc. of ICSE*, pp. 1049–1060, 2020.

[29] "Depclean." https://github.com/castor-software/depclean/tree/master/depclean-core. Accessed: 2023-05-18.

[30] T. Durieux, C. Soto-Valero, and B. Baudry, "Duets: A dataset of reproducible pairs of java library-clients," in *Proc. of MSR*, pp. 545–549, 2021.

[31] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[32] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, 2021.

[33] Google, "Guava listenablefuture," January 2023. Available at https://mvnrepository.com/artifact/com.google.guava/listenablefuture/9999.0-empty-to-avoid-conflict-with-guava.

[34] G. Balatsouras and Y. Smaragdakis, "Class hierarchy complementation: soundly completing a partial type graph," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 515–532, 2013.

[35] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 2, pp. 1–50, 2019.

[36] Helidon, "Helidon snakeyaml helper maven plugin," January 2023. Available at https://github.com/helidon-io/helidon-build-tools/tree/master/maven-plugins/snakeyaml-codegen-maven-plugin.

[37] GitHub, "Issue 327," October 2018. Available at https://github.com/DiUS/java-faker/issues/327.

[38] O. C. and/or its affiliates, "Project leyden: Beginnings," May 2022. Available at https://openjdk.org/projects/leyden/notes/01-beginnings.

[39] GraalVM, "The graalvm native image," January 2023. Available at https://www.graalvm.org/native-image.

[40] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: Application initialization at build time," *Proc. of OOPSLA*, 2019.

[41] C. Wimmer, "Graalvm native image: large-scale static analysis for java (keynote)," in *Proc. of Workshop on Virtual Machines and Intermediate Languages*, pp. 3–3, 2021.

[42] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir, "On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation," in *Proc. of APLAS*, pp. 69–88, 2018.

[43] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *Proc. of ICSE*, pp. 507–518, 2017.

[44] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proc. of ISSTA*, pp. 251–261, 2019.

[45] Oracle, "The graalvm reachability metadata repository," January 2023. Available at https://github.com/oracle/graalvm-reachability-metadata.

[46] M. Šipek, D. Muharemagić, B. Mihaljević, and A. Radovan, "Enhancing performance of cloud-based software applications with graalvm and quarkus," in *Proc. of MIPRO*, pp. 1746–1751, 2020.

[47] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.

[48] M. Ahmadvand, A. Pretschner, and F. Kelbert, "A taxonomy of software integrity protection techniques," in *Advances in Computers*, vol. 112, pp. 413–486, Elsevier, 2019.

[49] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.

[50] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "Proc. of icse," *arXiv preprint arXiv:2301.05362*, 2023.

[51] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for java," in *Proc. of OOPSLA*, pp. 292–305, 1999.

[52] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 745–764, 2013.

[53] J. B. F. Filho, M. Acher, and O. Barais, "Software unbundling: Challenges and perspectives," *LNCS Trans. Modul. Compos.*, vol. 1, pp. 224–237, 2016.

[54] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *Proc. of COMPSAC)*, vol. 1, pp. 12–21, IEEE, 2016.

[55] D. Rayside and K. Kontogiannis, "Extracting java library subsets for deployment on embedded systems," *Science of Computer Programming*, vol. 45, no. 2-3, pp. 245–270, 2002.

[56] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: In-depth investigation into debloating modern java applications," in *Proc. of ESEC/FSE*, pp. 135–146, 2020.

[57] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The Emergence of Software Diversity in Maven Central," in *Proc. of MSR*, pp. 1–10, 2019.

[58] S. Mishra and M. Polychronakis, "Saffire: Context-sensitive function specialization against code reuse attacks," in *Proc. of (EuroS&P)*, pp. 17–33, IEEE, 2020.

[59] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: debloating dynamic server-side javascript applications," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–36, 2022.

[60] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[61] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *Proc. of USENIX Security*, pp. 1697–1714, 2019.

[62] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proc. of ACSAC*, pp. 70–83, 2019.