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

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

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 × in the original projects to 4.4 × 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

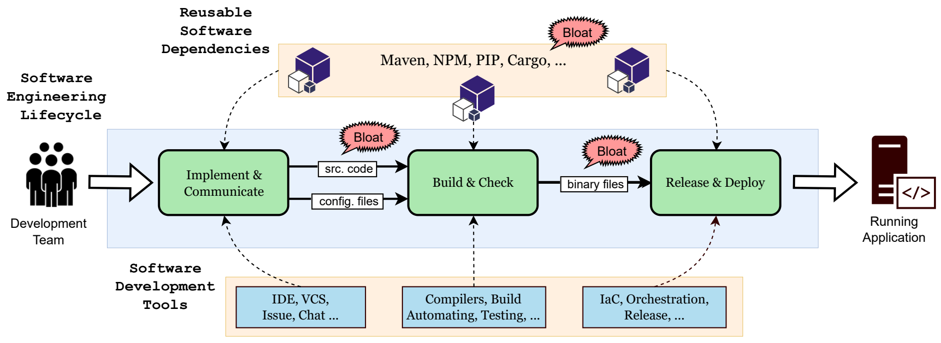
Reusable Software Dependencies
Maven, NPM, PIP, Cargo, ...
Bloat
Software Engineering Lifecycle
Development Team
Implement & Communicate
src. code
config. files
Bloat
Build & Check
binary files
Bloat
Release & Deploy
Running Application
Software Development Tools
IDE, VCS, Issue, Chat ...
Compilers, Build Automating, Testing, ...
IaC, Orchestration, Release, ...

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

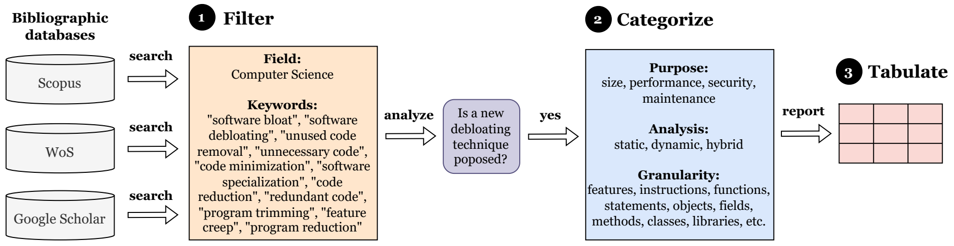
Bibliographic databases
Scopus
WoS
Google Scholar
search
search
search
1 Filter
Field: Computer Science
Keywords: "software bloat", "software debloating", "unused code removal", "unnecessary code", "code minimization", "software specialization", "code reduction", "redundant code", "program trimming", "feature creep", "program reduction"
analyze
Is a new debloating technique poposed?
yes
2 Categorize
Purpose: size, performance, security, maintenance
Analysis: static, dynamic, hybrid
Granularity: features, instructions, functions, statements, objects, fields, methods, classes, libraries, etc.
report
3 Tabulate

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

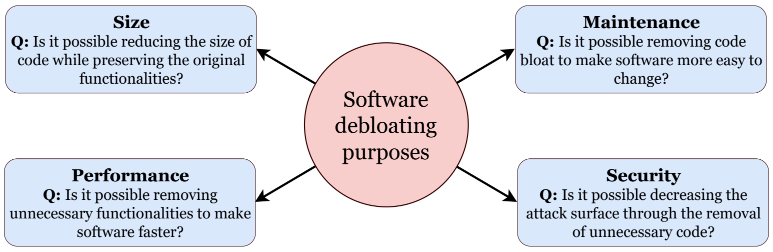
Size
Q: Is it possible reducing the size of code while preserving the original functionalities?
Maintenance
Q: Is it possible removing code bloat to make software more easy to change?
Software debloating purposes
Performance
Q: Is it possible removing unnecessary functionalities to make software faster?
Security
Q: Is it possible decreasing the attack surface through the removal of unnecessary code?

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
 1 import java.lang.reflect.Method;
 2 import java.util.Scanner;
 3
 4 public class Foo {
 5     public static void main(String[] args) {
 6         Scanner scanner = new Scanner(System.in);
 7         try {
 8             String className = "Foo";
 9             String methodName = "greet";
10             String personName = scanner.next();
11             // Dynamically loading a class
12             Class<?> clazz = Class.forName(className);
13             // Dynamically invoking a method using reflection
14             Method method = clazz.getDeclaredMethod(methodName, String.class);
15             method.invoke(null, personName);
16         } catch (Exception e) {
17             // Catch the exception
18         }
19     }
20
21     // This method is invoked using reflection
22     public static void greetAlice(String name) {
23         if (name.equalsTo("Alice"){
24             System.out.println("Hello, " + name);
25         } else {
26             System.out.println("Sorry, I don't know you");
27         }
28     }
29
30     // This method is never called and could be removed by debloating
31     public static void unusedMethod() {
32         System.out.println("This method is never used.");
33     }
34 }
```

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 tracing 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

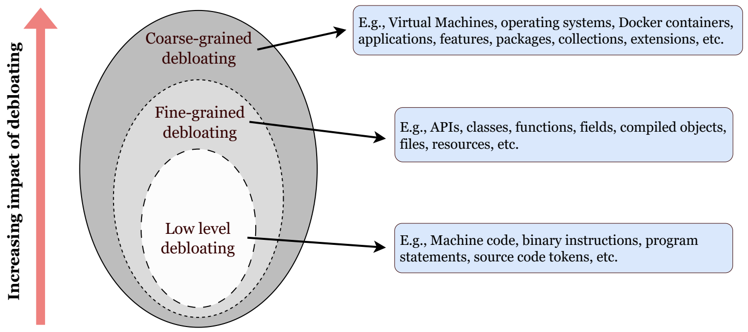
Increasing impact of debloating
Coarse-grained debloating
Fine-grained debloating
Low level debloating
E.g., Virtual Machines, operating systems, Docker containers, applications, features, packages, collections, extensions, etc.
E.g., APIs, classes, functions, fields, compiled objects, files, resources, etc.
E.g., Machine code, binary instructions, program statements, source code tokens, etc.

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.

```
<groupId>org.p</groupId>
<artifactId>p</artifactId>
<version>0.0.1</version>
<packaging>jar</packaging>
. . .
<dependencies>
  <dependency>
    <groupId>org.d1</groupId>
    <artifactId>d1</artifactId>
  </dependency>
  <dependency>
    <groupId>org.d2</groupId>
    <artifactId>d2</artifactId>
  </dependency>
    <dependency>
    <groupId>org.d3</groupId>
    <artifactId>d3</artifactId>
  </dependency>
</dependencies>
. . .
```

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

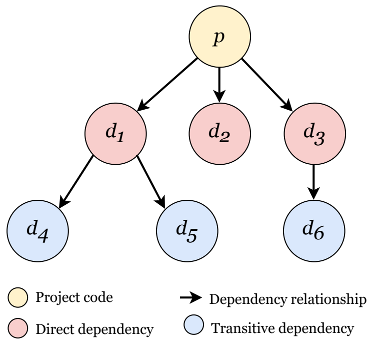

Figure 3.1: Dependency tree from the `pom.xml` file of Listing 3.1. The project p declares the direct dependencies d_1, d_2, and d_3. The dependencies d_4, d_5, and d_6 are transitive dependencies of p.

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 ∇ assigns each edge one of the following six dependency usage types: $\nabla : \mathcal{E} \to \{\text{ud, ui, ut, bd, bi, bt}\}$ 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

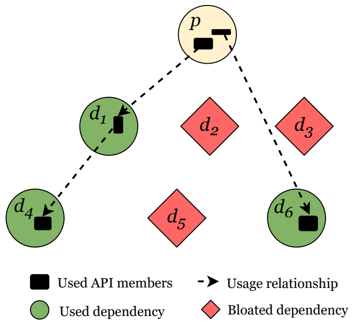

Figure 3.2: Dependency usage tree of used and bloated dependencies corresponding to the dependency tree presented in Figure 3.1.

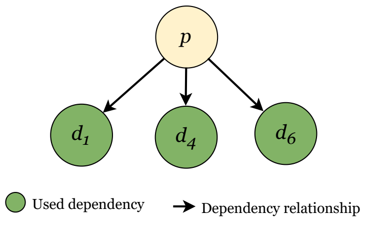

Figure 3.3: Debloated dependency tree after removing bloated dependencies with DEPCLEAN, based on the DUT of Figure 3.2.

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.

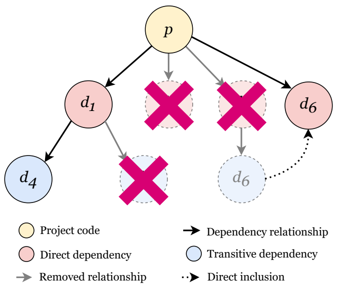

Figure 3.4: Transformations in the dependency tree of p as a result of the changes in the pom.xml file indicated in Listing 3.2.

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 dependencies 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

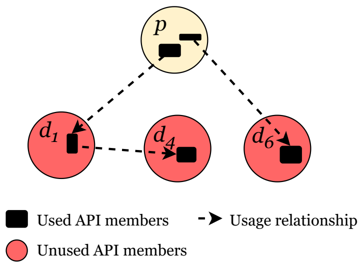

Figure 3.5: Used and unused API members in the debloated dependency tree from Figure 3.3.

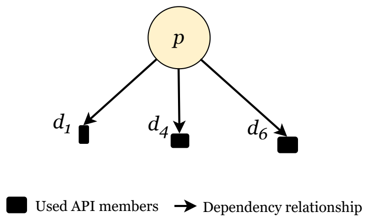

Figure 3.6: Specialized dependency tree after removing unused API members from Figure 3.5.

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

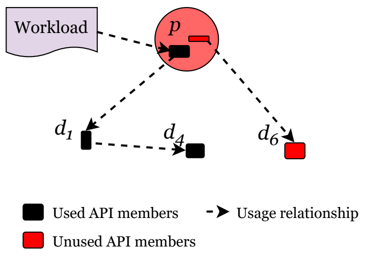

Figure 3.7: Used and unused API members in the dependency tree of Figure 3.6. Note that the usage status is *w.r.t.* the supplied workload.

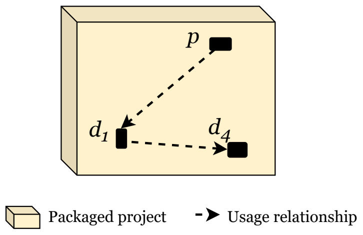

Figure 3.8: Debloated project from Figure 3.7. Only the used API members of the project and its dependencies are packaged.

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

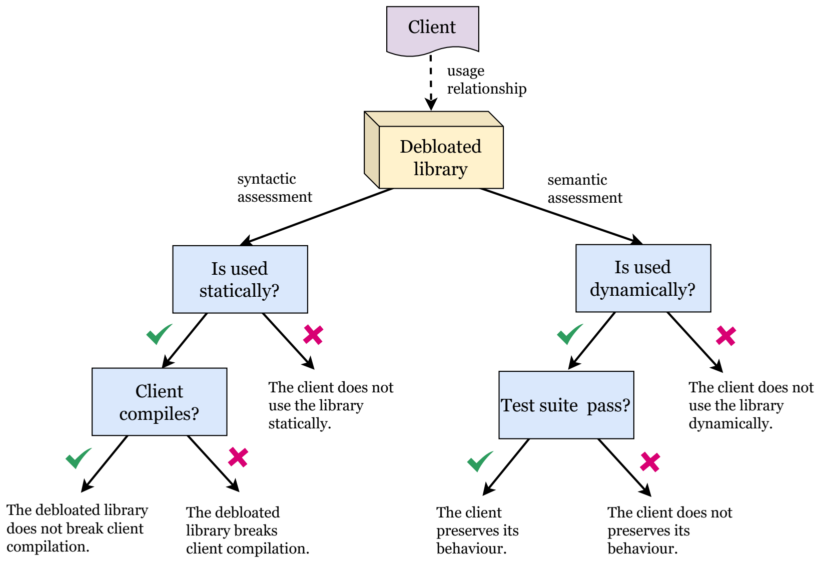
Client
usage relationship
Debloated library
syntactic assessment
semantic assessment
Is used statically?
Is used dynamically?
Client compiles?
The client does not use the library statically.
Test suite pass?
The client does not use the library dynamically.
The debloated library does not break client compilation.
The debloated library breaks client compilation.
The client preserves its behaviour.
The client does not preserves its behaviour.

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