# Enhancing the Security of Software Supply Chains: Methods and Practices

Keshani, M.

**DOI**
10.4233/uuid:e0c6d9a9-9249-4890-8f48-374bd5f49b5d

**Publication date**
2024

**Document Version**
Final published version

**Citation (APA)**
Keshani, M. (2024). *Enhancing the Security of Software Supply Chains: Methods and Practices*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:e0c6d9a9-9249-4890-8f48-374bd5f49b5d

# Enhancing the Security of Software Supply Chains: Methods and Practices

# Enhancing the Security of Software Supply Chains: Methods and Practices

**Dissertation**

for the purpose of obtaining the degree of doctor
at Delft University of Technology
by the authority of the Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen
chair of the Board for Doctorates
to be defended publicly on
Monday 14 October 2024 at 17:00 o'clock

by

**Mehdi KESHANI**

Master of Science in Computer Engineering,
Sharif University of Technology, Iran,

This dissertation has been approved by the promotor.

Composition of the doctoral committee:

The work in the thesis has been carried out under the support of the European H2020 project, FASTEN (825328).

*Keywords:* Maven Ecosystem, Software Supply Chain

*Printed by:*

*Style:* TU Delft House Style, with modifications by Moritz Beller
https://github.com/Inventitech/phd-thesis-template

The author set this thesis in LaTeX using the Libertinus and Inconsolata fonts.

*To those whose support carried me through the highs and lows of this PhD,*
*you know who you are, and I am deeply grateful.*

Mehdi.

# Contents

# Summary

Software supply chains include the development, management, and delivery of software products. Software ecosystems are essential components of these supply chains and facilitate software reuse and enhance the efficiency of software development. However, they also introduce unique challenges, such as dependency maintenance and security vulnerabilities. The Maven ecosystem is a popular software ecosystem within the Java realm and is used by millions of developers worldwide. This ecosystem faces issues that threaten its security and effectiveness. In this thesis, we aim to tackle these challenges by proposing novel methods and in-depth analyses of the ecosystem.

First, we propose an automated approach for library reproducibility to enhance library security during the deployment phase. We then develop a scalable call graph generation technique to support various use cases, such as method-level vulnerability analysis and change impact analysis, which help mitigate security challenges within the ecosystem. Utilizing the generated call graphs, we explore the impact of libraries on their users. Finally, through empirical research and mining techniques, we investigate the current state of the Maven ecosystem, identify harmful practices, and propose recommendations to address them.

In this thesis, we investigate the reproducibility of Maven artifacts from their source code and demonstrate that a significant portion of the Maven ecosystem could be made reproducible with automation. We then explore the scalability of method-level analysis in examining library interactions using call graphs, achieving significant performance improvements through a summarization-based call graph generation technique. These enhancements make static analyses more practical. This technique allows us to assess the impact of libraries within Maven. We show a significant concentration of method usage among a small portion of public methods. We also discover that these popular methods experience breaking changes as frequently as methods not utilized by any users, indicating a lack of awareness among library maintainers about their library usage. Lastly, we identify challenges in packaging practices that pose risks to ecosystem users. These challenges highlight the need for improved governance for the ecosystem and developer awareness regarding the security and impact of libraries on users.

# Samenvatting

Software supply chains omvatten de ontwikkeling, het beheer en de levering van softwareproducten. Software-ecosystemen zijn essentiële componenten van deze supply chains en faciliteren het hergebruik van software en verbeteren de efficiëntie van softwareontwikkeling. Ze brengen echter ook unieke uitdagingen met zich mee, zoals het in stand houden van afhankelijkheid en beveiligingsproblemen. Het Maven-ecosysteem is een populair software-ecosysteem binnen het Java-domein en wordt door miljoenen ontwikkelaars wereldwijd gebruikt. Dit ecosysteem wordt geconfronteerd met problemen die de veiligheid en effectiviteit ervan bedreigen. In dit proefschrift willen we deze uitdagingen aanpakken door nieuwe technieken en diepgaande analyses van het ecosysteem voor te stellen.

Ten eerste stellen we een geautomatiseerde aanpak voor bibliotheekreproduceerbaarheid voor om de bibliotheekbeveiliging tijdens de implementatiefase te verbeteren. Vervolgens ontwikkelen we een schaalbare call graph-generatietechniek ter ondersteuning van verschillende gebruiksscenario's, zoals kwetsbaarheidsanalyse op methodeniveau en analyse van de impact van veranderingen, die de beveiligingsuitdagingen binnen het ecosysteem helpen verminderen. Met behulp van de gegenereerde oproepgrafieken onderzoeken we de impact van bibliotheken op hun gebruikers. Ten slotte onderzoeken we door middel van empirisch onderzoek en mijnbouwtechnieken de huidige toestand van het ecosysteem, identificeren we schadelijke praktijken en stellen we aanbevelingen voor om deze aan te pakken.

In dit proefschrift onderzoeken we de reproduceerbaarheid van Maven-artefacten vanuit hun broncode en laten we zien dat een aanzienlijk deel van het Maven-ecosysteem reproduceerbaar kan worden gemaakt met automatisering. Vervolgens onderzoeken we de schaalbaarheid van analyse op methodeniveau bij het onderzoeken van bibliotheekinteracties met behulp van call graphs, waarbij we aanzienlijke prestatieverbeteringen bereiken via een op samenvattingen gebaseerde techniek voor het genereren van call graphs. Deze verbeteringen maken statische analyses praktischer. Met deze techniek kunnen we de impact van bibliotheken binnen Maven beoordelen. We laten een significante concentratie van methodegebruik zien bij een klein deel van de publieke methoden. We ontdekken ook dat deze populaire methoden net zo vaak ingrijpende veranderingen ondergaan als methoden die niet door gebruikers worden gebruikt, wat wijst op een gebrek aan bewustzijn onder bibliotheekbeheerders over hun bibliotheekgebruik. Ten slotte identificeren we uitdagingen in verpakkingspraktijken die risico's opleveren voor ecosysteemgebruikers. Deze uitdagingen benadrukken de noodzaak van een beter beheer van het ecosysteem en het bewustzijn van ontwikkelaars met betrekking tot de beveiliging en de impact van bibliotheken op gebruikers.

# 1 Introduction

## 1.1 Software Supply Chains

Software supply chains involve the creation, management, and delivery of software products. They include a wide range of activities, from the development of software code to the management of dependencies, and the distribution of the final product to the end users. Such chains are necessary for the functioning of modern software and ensure that software products are developed efficiently [1].

Software supply chains are the backbone of software development and delivery processes. Developers can check if software products are built using reliable components, adhere to quality standards, and are delivered securely and efficiently. A well-managed software supply chain minimizes risks associated with security vulnerabilities, code quality issues, and compliance requirements. It also enables faster development cycles and a more robust response to market demands [2].

Security has been a classic challenge in software supply chains. Since the early 2000s, researchers have discussed how software distribution invites attacks [3]. By the 2010s, they were proposing models to help evaluate and mitigate the security risks of software supply chains [4]. With the rise of software ecosystems in the last decade, such challenges are still relevant and even more widespread [5]. In 2021, a vulnerability in the popular library Log4j [6] affected nearly every software organization [7]. Researchers, practitioners, and governments investigated such issues and proposed methods to mitigate them. Ohom et al. [8] collected a set of 174 malicious packages from known package repositories. It was later discussed in the community that organizations with a software bill of materials (SBOM) were able to mitigate vulnerability issues such as the Log4j problem faster than organizations without it [9, 10]. SBOMs [11] are formal records that contain information regarding software components used in programs. In 2022, an executive order mandated federal agencies to include SBOMs [12]. Zahan et al. [7] elaborate on the benefits and challenges of adopting SBOMs. Ladisa et al. [13] propose a taxonomy for attacks on open-source supply chains. Enck et al.[5] report on the top five challenges in software supply chain security. Zahan et al.[14] propose six signals of security weaknesses in a software supply chain. To this day, security challenges are still present and pressing, and the community is actively working on solutions to identify the risks and mitigate them.

## 1.2 Research Objectives and Questions

The goal of this thesis is to enhance the security of software supply chains. To achieve this objective, we aim to answer four research questions.

The first area of improvement addressed in this thesis is related to a security challenge: the reproducibility of libraries. We define the reproducibility of a library as the ability to produce, from the library's sources, a set of (binary) files that are byte-for-byte equivalents. Reproducible libraries are important as they ensure that no malicious content is injected into the released files during the deployment process. While seemingly simple, reproducibility in practice is hard to achieve because the sources of unreproducibility are very diverse, making it difficult to address them automatically. Therefore, our first research question is posed as follows:

- RQ1) To what extent can published libraries be automatically reproduced from code?

Next, we aim to address the existing security issues in the development phase of libraries. Our goal is to enable accurate analysis of libraries to enhance security. Conducting such precise analyses requires a technique for examining the fine-grained interactions among software packages. Call graphs reveal method-level interactions within software systems. For example, by using call graphs, one can investigate whether a software application is utilizing a vulnerable part of a library. This type of analysis requires the generation of call graphs for libraries. However, the existing techniques for generating call graphs are heavyweight and impractical for important use cases like analyzing the entire ecosystem. Therefore, we pose our second research question as follows:

- RQ2) How can we improve the scalability of call graph generation for library analysis?

A lightweight call graph generation technique can be used in various scenarios for analyzing libraries, which can be categorized based on the direction of call graph analysis. The first category, which we explore in this thesis, involves analysis from the library to the clients. This enables library maintainers to examine how their methods are being utilized by clients, thereby understanding the impact of their libraries. The second category, which we do not explore in this thesis, is directed from client applications towards the libraries, enabling client applications to identify which library methods they use directly and transitively [15].

The first scenario forms the next step of this thesis and is referred to as impact analysis. By leveraging the generated call graphs, we aim to investigate the impact of libraries on their clients at the method level. Improving the awareness of library maintainers regarding the impact they have on their users can assist in the development of more secure software. Informing library maintainers that their library is utilized by numerous clients, or that a specific component of their library affects a larger number of people, encourages them to dedicate more time and attention to the quality of that particular library or component. This helps in introducing less vulnerable or low-quality code in the most critical areas. The research question addressed in this study is:

- RQ3) How do libraries impact their users at the method level?

Finally, we focus on studying the packaging practices within software ecosystems. This is crucial for identifying existing packaging practices that pose challenges to secure ecosystems. Additionally, it encourages the development or adoption of more effective practices and techniques within the ecosystem. Therefore, our last question is:

- RQ4) How do the packaging practices of libraries impact the package repositories?

## 1.3 Research Context

We address our research questions in the context of the Java language and its primary software ecosystem, Maven. Software ecosystems contribute to the efficiency of the software supply chain by simplifying and automating many aspects of project building and management. They standardize project structures, manage dependencies, and streamline the build process. Their comprehensive repository and dependency management capabilities are essential parts of modern software development and help developers efficiently add, update, and maintain an extensive set of libraries and frameworks. Package managers such as Maven, npm, and PyPi facilitate the processes of releasing, discovering, and distributing software by using centralized repositories. The investigation of dependencies has been a prominent theme in existing research, which includes investigation of the evolution of dependency networks [16, 17], issues related to bloated dependencies [18, 19], and security vulnerabilities within dependencies [20].

### 1.3.1 Supply Chains for Java

The research questions discussed in Section 1.2 are broad; thus, we specifically focus on the Java context to address them. Java remains one of the most popular programming languages, particularly for developing enterprise systems. Maven stands out as the primary software ecosystem not only for Java but also for other JVM languages. It serves as the core for managing, storing, and disseminating reusable Java open-source libraries. In recent years, Maven has seen substantial growth, hosting more than 38 million indexed artifacts at the time of writing (March 2024) [21]. While existing research has explored package repositories such as NPM [22–26] and PyPi [27], a significant body of literature dedicated to Maven [15–17, 19, 28–37] underscores its importance. However, despite extensive research, critical challenges, especially from a security perspective, remain unaddressed within the Maven ecosystem.

The challenges that we focus on in this thesis are security-related. These challenges arise from a variety of sources such as outdated dependencies, inconsistent maintenance of libraries, complex dependency trees, security vulnerabilities, lack of standardized practices across different projects, and compatibility issues in the development process.

For modern software systems, vulnerabilities found in third-party libraries are a common source of security problems. Research has shown that a considerable number of applications use libraries with security vulnerabilities [38]. Moreover, 37% of the most popular websites incorporate at least one library that is known to be vulnerable [39]. Vulnerabilities, if not properly managed and updated, can propagate across different libraries through transitivity, leading to potential security breaches in the final software products. Identifying and addressing these vulnerabilities is important to maintain the security of the software as well as the entire ecosystem.

Maven users, similar to users of other ecosystems, suffer from security vulnerabilities. Due to the interconnected nature of dependencies a single vulnerable component can compromise the security of an entire program or even a big part of the ecosystem. This highlights the need for effective tools and practices to detect and mitigate vulnerabilities in a timely manner. Several recent studies focused on exploring the effects and spread of security vulnerabilities within software ecosystems [40–43]. Moreover, attackers can manipulate the binaries of libraries while being built or deployed [44]. This poses a significant risk to the security of the software supply chain. This manipulation can lead to the injection of malicious code that compromises the security of the software and the data it processes. Therefore it is important to make sure that the binaries are not manipulated during the build and deployment processes.

User or client awareness is another crucial aspect that can aid library maintainers in developing more secure libraries. The actions of library maintainers have direct consequences for their users. For instance, if a maintainer includes a large file in the library, users are required to download it; similarly, if the maintainer introduces vulnerable code, it endangers the security of the users. Precisely understanding how libraries impact their users on a large scale is necessary to prevent harmful behavior that affects users and to plan beneficial actions. Numerous studies have investigated various aspects of the Application Programming Interfaces (APIs) that libraries provide for their users, including usage [45, 46], evolution [47–50], and stability [37, 51]. However, these studies have not detailed the extent of the method-level impact of libraries on the ecosystem.

### 1.3.2 The FASTEN Project

By using centralized repositories like Maven, developers can simply list the external libraries they need, after which automated tools integrate these libraries into their project's working environment. However, reliance on external library networks has its drawbacks, as highlighted by several incidents: the *XZ* attack [52], which allowed unauthorized access to systems where the compromised library was installed; the SolarWinds compromise [53], which injected malicious code into the SolarWinds Orion software used by many clients, including the U.S. government; the log4shell vulnerability [54], a remote code execution vulnerability in a commonly used Apache logging library; the *left-pad* incident [55], which caused numerous websites to fail; and the *Equifax* security breach [56], that exposed vast numbers of credit card details. These events emphasize the risks that external dependencies can pose to projects. Addressing these issues could significantly improve the productivity and output quality of software development organizations.

Motivated by these challenges, the FASTEN project [57] proposed a novel approach by offering detailed, method-level dependency tracking across existing dependency management systems. Specifically, FASTEN introduced a service of method-level analysis of security vulnerabilities, adherence to licensing requirements, and assessment of dependency risks at the method level.

Existing approaches have used package-level analysis to investigate the relationships between applications and libraries. As shown in Figure 1.1, a package-level analysis only considers the high-level dependency relationships between packages and does not take into account the fine-grained, method-level interactions between them. For example, tools like Dependabot [58] notify users about the vulnerability of `dep3` and inform the developers of `app` that they are using a vulnerable dependency. However, a closer examination of these interactions reveals that the method `a()` from `dep1` is called via the `main()` method of `app`. Method `a()` then calls method `vul()` from `dep3`, which is vulnerable. Nevertheless, if method `a()` were to call `d()` instead, `app` would technically be safe because it would not be calling the vulnerable part of `dep3`. The FASTEN project attempted to leverage this insight by using call graphs to improve the accuracy of dependency analysis.

This thesis was funded by the European H2020 project FASTEN (Grant No. 825328), which formed a team of over 40 experts from both the industrial and academic sectors throughout Europe. FASTEN aimed to design an innovative software package management system to enhance the quality and security of software ecosystems. FASTEN started in January 2019, prior to the start of this PhD. TU Delft led the project, which provided this PhD program with extensive managerial, teamwork, and hands-on experiences. In September 2022, FASTEN successfully ended after presenting its results to a committee of reviewers from the European Union. Our research and development efforts within

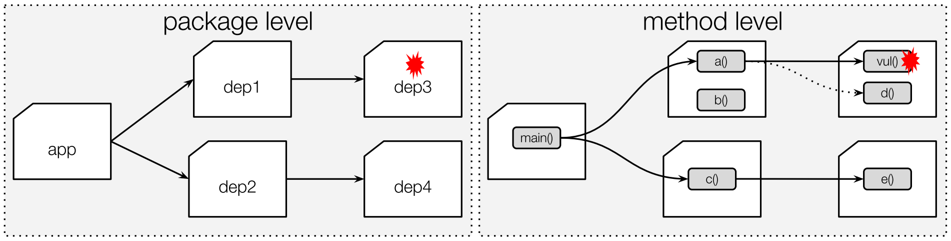


Figure 1.1: Package-level analysis versus method-level analysis.

the project resulted in the creation of novel analysis techniques for C, Java, and Python applications. These techniques tackled issues related to security, risk evaluation, license compliance, and assessing the impact of changes. Furthermore, we developed a backend service and a knowledge base to store the outcomes of these analyses and integrated them into developers' workflows and CI toolchains.

The research conducted for this thesis formed the foundation of FASTEN and was incorporated into FASTEN's open-source tools and deliverables. Techniques similar to those discussed in this thesis were also applied to Python and C components within FASTEN. The FASTEN tooling contains a wide range of functionalities, including, but not limited to the contributions of this thesis. Additionally, our work has been adopted by two industrial companies, Endorlabs [59] and SIG [60], and is presently being utilized and extended to other languages.

Within the scope of the FASTEN project, we generated tens of terabytes of data regarding software ecosystems and open-source libraries. This extensive information was accessible through FASTEN's public services during the project's active period. Upon the project's completion, we stopped maintaining a public service for data access. The data, however, remains stored on TU Delft servers. To support ongoing research, we curated and separately released the critical data necessary for the studies discussed in this thesis. These datasets are available in smaller, more manageable dumps within the replication packages of our studies on GitHub and Zenodo. This approach helps future researchers to replicate our studies, expand upon them, and derive new insights.

## 1.4 RESEARCH METHOD

In this thesis, we employ design science, empirical, and mining methods to draw insights from the MAVEN repository and evaluate our proposed approaches with real-world data. We create a scalable data pipeline that enables us to process the large amount of data that exists on MAVEN efficiently.

Empirical software engineering methods [61], including case studies, surveys, and formal experiments, are used to understand, assess, and enhance the processes and outcomes of software development. These approaches provide a foundation for software engineering practices based on scientific evidence derived from real-world software development activity data. Software engineering scientists have created standards for conducting and reviewing such studies [62] and have specified different categories of empirical methodologies [63]. In this thesis, we utilize benchmarking, engineering research (design science),

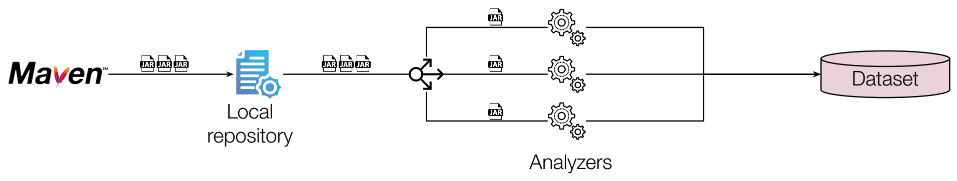


Figure 1.2: Simplified architecture of the data pipeline used in our studies.

and repository mining methodologies.

According to these standards, benchmarking involves assessing a software system using a standard tool (i.e., a benchmark) based on specific characteristics such as performance. We assess our proposed tools using benchmarks and compare them with existing approaches.

Design science methodologies [64] involve developing artifacts to benefit stakeholders and rigorously evaluating their performance within specific contexts using empirical methods. This PhD was conducted within the context of the FASTEN project, as explained in Section 1.3.2. Throughout the PhD, we created various artifacts such as software, documentation, and scientific papers. The primary stakeholders who benefited from these products were the FASTEN partners who utilized these artifacts within the FASTEN project. The software artifacts are still used by one of the industrial partners, SIG [60], as part of their commercial product. On a secondary level, developers can use the open-source software created during this PhD to conduct useful analyses and improve the quality of their products. Scientists can also use these artifacts to conduct follow-up research. The performance of these artifacts was assessed using various techniques, depending on the type of artifact. Software artifacts were reviewed and used in practice by FASTEN collaborators. The scientific publications underwent peer review, documentation was reviewed by peers, and novel technologies were compared against existing baselines.

Moreover, software engineering scientists often identify patterns, trends, and relationships within software projects to derive generalizable knowledge. Mining Software Repositories (MSR) [65] is a research domain focused on extracting and analyzing the abundant data found in software repositories, including version control systems, artifact repositories, and issue tracking systems. MSR applies techniques from data mining, machine learning, and statistics to uncover insights about software development practices and trends. By mining software repositories, researchers and practitioners can gain insights into software evolution, developer behaviors, and software quality. In this thesis, we applied MSR techniques to gain insights about Maven, identify existing challenges, explore the current state, and pinpoint existing shortcomings.

**Ecosystem scale data pipelines** Figure 1.2 illustrates the simplified architecture of the data pipeline we employed for extracting Maven data in our research. It facilitates our empirical research and shapes the resulting data. Given the substantial amount of data on Maven, scalability is a crucial requirement for us. Considering the independence of Maven releases from one another, this infrastructure supports asynchronous data extraction from packages, which enhances scalability. Maven uses a local repository to store libraries on the local machines of Maven users. For instance, when Maven encounters a new library in an application, it downloads all its files and directory structures to this local repository, known as the `.m2` folder. We integrate this local repository into our data pipelines to process Maven libraries. This integration allows us to reuse original Maven implementations in our research and avoid downloading millions of duplicated files. The infrastructure we have developed is modular, facilitating the integration of custom analyzers that can append additional data to the dataset. We establish a reusable execution environment to support both the initial creation and periodic updates of the dataset. We utilized docker-compose to minimize environmental dependencies and simplify the execution process, allowing all experiments to be run with a single command. All analyzers and services, including the database, are dockerized and operate within this framework. Our dataset, stored in a Postgres database, enables data analysis and provides insights into the ecosystem.

**Open Science** The code and data corresponding to each chapter of this thesis are publicly accessible. We have four replication packages: AROMA on Zenodo[1] (Chapter 2), Frankenstein on GitHub[2] (Chapter 3), SemVerVSPopularity on GitHub[3] (Chapter 4), and Maven Packaging on Zenodo[4] (Chapter 5). The papers are available for open access through the TU Delft research repository [66].

## 1.5 Thesis Outline

The main goal of this thesis is to improve the security of software supply chains. In this section, we outline our research design and approach for addressing the research questions introduced in Section 1.2. This thesis contains four publications, each detailed in one chapter. The second chapter of this thesis focuses on enhancing library security through reproducibility, which prevents the injection of malicious code during the deployment of libraries. The third chapter introduces a novel call graph generation technique that facilitates various types of library analyses, including security analysis. The fourth chapter explores a specific use case of these call graphs. It presents a call graph-based analysis to assess the impact of libraries on their clients. This analysis helps library maintainers prioritize the quality and security of the most critical parts of their libraries. The fifth chapter investigates the packaging practices of libraries and the security challenges these practices present to the ecosystem.

**Chapter 2. AROMA: Automatic Reproduction of Maven Artifacts** To address RQ1, Chapter 2 focuses on the reproducibility of Maven libraries [67]. The *SolarWinds* attack [68] in 2020 impacted thousands of organizations, including the US government, by manipulating library binaries before shipping them to users and injecting malicious code. Attacks like this highlight the growing concerns over security. Ensuring the reproducibility of software components is a solution to this challenge [69]. By reproducing a library from its source, we can verify that no malicious content has been injected during the deployment process. This chapter introduces an automated tool, AROMA, a technique that automatically reproduces Maven libraries.

[1]https://doi.org/10.5281/zenodo.8380775
[2]https://github.com/ashkboos/LightWeightCGs/tree/main
[3]https://github.com/ashkboos/semver-vs-popularity
[4]https://zenodo.org/doi/10.5281/zenodo.10143429

The only existing initiative for reproducing Maven libraries is a manual approach, Reproducible Central [70], which curates a list of reproducible packages. Unlike this method, which begins with source code repositories, AROMA targets Maven releases directly, taking into account that many projects, especially legacy ones, might not have been developed with reproducibility in mind. AROMA provides broader coverage and helps reproduce projects that would otherwise be excluded from reproducibility efforts.

Reproducible Central creates a file for each library containing information about its build environment. Each file includes various fields, such as the Java version used by the library maintainers to release the library. Our results show that AROMA could automatically recover up to 99.5% of these build environment fields, which previously required manual effort in Reproducible Central. This shows the effectiveness of AROMA and highlights its potential in repairing inaccuracies within existing manually maintained lists. We also show that AROMA can rebuild 23.4% of the entire ecosystem and reproduce 8% of those packages, which is a substantial improvement over Reproducible Central (less than 1%). We even succeeded in achieving near-perfect reproductions for some binaries, which were not originally intended to be reproducible. This means that the maintainers of these libraries have not configured their pom file [71] for reproducibility. This shows AROMA's capability and is a significant step towards enhancing the security of the ecosystem.

In summary, this thesis chapter presents the AROMA technique and offers practical insights into software reproducibility within the Maven ecosystem. This research enhances the security of Java supply chains by focusing on the security of deployment processes. It represents the first area of improvement proposed in this thesis.

Subsequently, we explore a novel call graph generation technique that facilitates various types of library analyses, including security analysis.

**Chapter 3. Frankenstein: fast and lightweight call graph generation for software builds** To address our second research question, Chapter 3 introduces a novel and scalable call graph-based analysis technique [72] that can be utilized in a variety of use cases regarding dependency management. Call graphs can for example substantially improve the accuracy of vulnerability propagation analysis. Despite their importance, tools like Dependabot [58] operate at the package level and do not utilize call graphs. This is due to practical and scalability challenges associated with call graph generation, which is typically considered a *full program analysis* (application and all of its dependencies). Such an approach results in extensive and computationally expensive call graphs, which is impractical for ecosystem-scale applications where the number of combinations in a full program analysis explodes.

To address these challenges, we propose a summarization-based version of the Class Hierarchy Analysis (CHA) [73] call graph generation algorithm. This summarization-based algorithm generates call graphs on demand by assembling partial call graphs previously extracted from individual libraries. This approach is particularly suitable for large ecosystems like Maven, which hosts millions of software artifacts. It allows for fine-grained analysis to enhance reliability and accuracy in applications like vulnerability propagation. The traditional method of constructing a call graph for a complete application and its dependencies from scratch is inefficient for ecosystems, leading to redundant computations for popular libraries. Our approach improves scalability by eliminating redundant computations from the call graph generation process; that is, for each library, we extract the call graph summarization only once and reuse it multiple times. Our evaluations show that this method scales very well.

The modern practice of Continuous Integration (CI) [74] and delivery in software engineering often faces a trade-off between faster build times and the utility of additional analyses [75]. This is particularly relevant in open-source software development, where resource-limited build services like GitHub Actions [76] or Travis CI [77] are common. For example, Dependabot [58] performs basic analyses at the package level to identify vulnerable dependencies and is accepted by developers for its speed despite less accuracy. This highlights the need for scalable, efficient tools for analyses. Our summarization-based CHA algorithm for call graph generation takes advantage of the fact that dependency sets often remain unchanged between builds. By pre-generating call graph summaries for these dependencies and caching them for subsequent builds, we can merge them using a *stitching* algorithm and reduce resource consumption and processing time. Our evaluation shows that this lightweight method outperforms existing frameworks. It is compatible with the GitHub build environment's memory constraints and achieves speed improvements of up to 38%.

Overall, this chapter introduces a new library summarization technique for call graph generation using a *stitching* algorithm and provides a practical solution for fast, resource-efficient program analysis. We reassess existing methodologies focusing on correctness, scalability, and memory consumption. Our evaluation on a real-world Maven sample shows the feasibility of this approach for CI tools. However, this technique is only the stepping stone for advanced dependency analyses such as vulnerability propagation and impact analysis. In the next chapter of this thesis, we address the change impact analysis use case which helps developers focus on the quality and security of their most critical components.

**Chapter 4. On the relation of method popularity to breaking changes in the Maven ecosystem** To address our third research question, Chapter 4 delves into change impact analysis, with a particular focus on investigating semantic versioning violations within the Maven ecosystem [78, 79]. Semantic Versioning is a versioning system that uses a naming convention for releases to convey meaning about the underlying changes in a release, indicating backward compatibility and the nature of changes made. Our study goes beyond the boundaries of individual libraries and provides an ecosystem perspective. Using a large dataset that includes 13,876 versions of 384 Maven artifacts, we analyze the consequences of semantic versioning violations on 7,190 dependent libraries at the method level, utilizing call graphs.

Our findings reveal that 67% of the artifacts have violated semantic versioning at some point in their history, through either breaking changes or non-compliant API extensions. Our results also show a noticeable centralization regarding the method level usage of libraries: a vast majority (87%) of publicly accessible methods are never used by dependents, and within the subset of used methods, a mere 35% attract half of all unique dependent calls. This usage pattern highlights the criticality of certain methods within the ecosystem.

Our study reveals that there is no correlation between the popularity of a method and its stability. That is, even highly popular methods are not immune to frequent breaking changes. This observation challenges the assumption that more widely-used components may be more stable. Our research highlights the issues with semantic versioning compliance in practice. The high frequency of breaking changes in popular methods shows a lack of awareness about the impact of changes within the ecosystem. We believe that developers would benefit from accessing method popularity data obtained through our approach, enhancing their update strategies, mitigating the effects of breaking changes, and prioritizing the quality and security of the most critical components.

Overall, this chapter presents a detailed analysis of API method changes that violate semantic versioning, along with insights into the popularity distribution of methods in Maven. It contributes to the existing body of knowledge in software reuse and provides practical insights for developers.

In the subsequent chapter, we explore the impact of packaging practices on the ecosystem and its security.

**Chapter 5. Maven Unzipped: Packaging Impacts the Ecosystem** To address the fourth and final research question of this thesis, we explore the overall state of the ecosystem and the impact of packaging practices on it [80]. Despite its popularity among developers worldwide, the Maven ecosystem is not without its share of challenges and vulnerabilities. In this thesis we conduct a comprehensive investigation of the Maven ecosystem, aiming to gain insights into its practices and trends. By inspecting the content of libraries, examining their dependency relationships, and investigating their evolution over time, we find areas that require attention. Maven ecosystem's massive scale and influence make it a central point not just for library developers but also for ecosystem governors and stakeholders who are involved with software security and development.

In Chapter 5, we elaborate on the research methodology of this chapter and the issues identified within the Maven ecosystem. In short, we have created an infrastructure that monitors statistics about Maven packages. Our goal is to analyze these statistics to shed light on the current state of the ecosystem, help make informed decisions about it, and promote best practices. The issues we identify include data inconsistencies, improper archive utilization, and the exponential growth of transitive dependency sets. These issues not only hinder the ecosystem's efficiency but also introduce significant risks and exacerbate security problems for its users. Building upon these findings, we propose practical recommendations to enhance the overall health and security of Maven.

## 1.6 Origin of Chapters

Throughout this PhD, we conducted eight studies, as detailed below. Four of these studies, marked with the 🗎 symbol, are included as chapters in this thesis. Additionally, four studies were conducted that fall outside the scope of this thesis.

- 🗎 *Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch*: Aroma: Automatic Reproduction of Maven Artifacts, Proceedings of the 32st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE), 2024.

🗎 *Mehdi Keshani, Georgios Gousios, and Sebastian Proksch*: Frankenstein: Fast and Lightweight call graph Generation for Software Builds. Empirical Software Engineering (EMSE), 29: pages 1–31, 2024.

🗎 *Mehdi Keshani, Simcha Vos, and Sebastian Proksch*: On the Relation of Method Popularity to Breaking Changes in the Maven Ecosystem. Journal of Systems and Software (JSS), 203: pages 111738, 2023.

🗎 *Mehdi Keshani, Gideon Bot, Priyam Rungta, Maliheh Izadi, Arie Van Deursen, and Sebastian Proksch*: Maven Unzipped: Exploring the Impact of Library Packaging on the Ecosystem. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024.

*Mehdi Keshani*: Scalable call graph Constructor for Maven. IEEE/ACM 43rd International Conference on Software Engineering: ICSE Doctoral Symposium, pages 99–101. IEEE, 2021.

*Amir M. Mir, Mehdi Keshani, and Sebastian Proksch*: On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 201–211. IEEE, 2023.

*Amir M. Mir, Mehdi Keshani, and Sebastian Proksch*: On the Effectiveness of Machine Learning based call graph pruning: An Empirical Study. IEEE/ACM 21st International Conference on Mining Software Repositories (MSR), 2024.

*Amir M. Mir, Mehdi Keshani, and Sebastian Proksch*: OriginPruner: Leveraging Method Origins for Guided Call Graph Pruning. Paper under review, 2024.

# 2 AROMA: Automatic Reproduction of Maven Artifacts

Modern software engineering establishes software supply chains and relies on tools and libraries to improve productivity. However, reusing external software in a project presents a security risk when the source of the component is unknown or the consistency of a component cannot be verified. The SolarWinds attack serves as a popular example in which the injection of malicious code into a library affected thousands of customers and caused a loss of billions of dollars. Reproducible builds present a mitigation strategy, as they can confirm the origin and consistency of reused components. A large reproducibility community has formed for Debian, but the reproducibility of the Maven ecosystem, the backbone of the Java supply chain, remains understudied in comparison. Reproducible Central is an initiative that curates a list of reproducible Maven libraries, but the list is limited and challenging to maintain due to manual efforts. Our research aims to support these efforts in the Maven ecosystem through automation. We investigate the feasibility of automatically finding the source code of a library from its Maven release and recovering information about the original release environment. Our tool, AROMA, can obtain this critical information from the artifact and the source repository through several heuristics and we use the results for reproduction attempts of Maven packages. Overall, our approach achieves an accuracy of up to 99.5% when compared field-by-field to the existing manual approach. In some instances, we even detected flaws in the manually maintained list, such as broken repository links. We reveal that automatic reproducibility is feasible for 23.4% of the Maven packages using AROMA, and 8% of these packages are fully reproducible. We demonstrate our ability to successfully reproduce new packages and have contributed some of them to the Reproducible Central repository. Additionally, we highlight actionable insights, outline future work in this area, and make our dataset and tools available to the public.[1]

[1]This chapter is based on the following paper: Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of maven artifacts. FSE, 2024 [67].

Software ecosystems, such as Maven, host libraries and build plugins in repositories, which form the backbone of many software supply chains. Maven Central is the largest public repository for JVM-based languages, allowing developers to reuse software and release their own packages there.

Relying on a supply chain has the risk of importing vulnerabilities into a project, and it either requires trust in the source or the ability to check the consistency of the packages. Unfortunately, research highlights an increasing number of attacks in recent years [81]. Some of them became popular, like the infamous SolarWinds hack in 2020, which resulted in the shipping of tainted binaries to thousands of customers, including the U.S. government, which was subsequently hacked [68]. The *Backstabber's Knife Collection* by Ohm et al. documents 174 similar attacks [8].

It is obvious that trust alone is insufficient, and checks are necessary. Wheeler et al. [69] were the first to propose a conceptual solution that involves comparing the compilation results of two different environments to detect modified binaries, such as those introduced in a malicious attack. A match is achieved when two software builds produce bit-for-bit identical binaries from the same source code, resulting in the same hash for the binary. If this can be accomplished, the binary is considered to be *reproducible*. This idea inspired the *Reproducible Builds* initiative [82], which recommends practices for improving software reproducibility. Many efforts, such as localizing unreproducibility [83, 84] or automatically addressing them [85], have been recently made.

So far, the community focus was Debian and Linux, which are currently ~90% reproducible [86], and reproducibility efforts for Maven packages are more limited. *Reproducible Central* (RC) represents a notable exception and provides tools for reproducing projects [70]. The project also maintains a list of reproducible Maven projects, however, the reproducibility of the Maven ecosystem is still in its infancy. No official statistics exist, but the RC repository reports rebuild attempts for 2,547 releases of 531 projects. Compared to the 10+M releases of 479K unique `artifactIds` and 64K unique `groupIds` that exist on Maven Central in May 2023, it becomes clear that this invaluable effort only covers a fraction of packages. Furthermore, RC starts their reproducibility attempts at GitHub repositories, which do not necessarily equate to Maven releases. One such repository can have multiple sub-modules and releases. Our analyses also found links to 64K unique GitHub repositories on Maven Central, which indicates that many repositories are currently missed.

Efforts to increase the reproducibility coverage are currently limited by several factors: 1) Reproduction is a manual, high-effort task, making it slow, unscalable, and prone to errors. 2) Starting from source code repositories assumes that developers aim to create reproducible software, which is not the case for many unmaintained legacy projects. 3) Current tooling often assumes the availability of the correct version of the source code for a given release. However, users who wish to reproduce a Maven package must first recover its source code. 4) Reproducibility is a binary metric and requires an *exact, bit-to-bit* match. Many factors influence this comparison, but not all differences are as critical as differences in the generated `.class` files. Even trivialities like file modification times or line endings can prevent a match. We believe that categorizing these cases as simply *unreproducible* is too coarse-grained, and that we need to distinguish reproducibility on a *spectrum*.

In this paper, we explore the possibility of automating the reproduction of large portions of the Maven ecosystem. We take the perspective of the community and try to achieve this goal without relying on the project maintainers. To achieve this goal, we investigate three key research questions:

**RQ$_1$** Can we recreate the link to a source code repository and commit of an artifact?

**RQ$_2$** Can the original build environment be reconstructed after the fact?

**RQ$_3$** To which extent is it possible to automatically reproduce Maven artifacts?

To answer these questions, we created a dataset of metadata for 480k packages. These packages represent one random version from every Maven project (`groupId:artifactId`). This dataset lets us explore the factors that influence release practices and reproducibility on Maven Central and we report on our empirical insights.

Our results are promising and show that an automated approach (AROMA) can recover up to ~99.5% of the information manually maintained by RC. We even identified inconsistencies in the manually maintained list, such as broken repository links. We successfully applied our approach in three practical use cases: reproducing packages that already existed on RC, identifying numerous reproducible libraries not previously listed by RC, and achieving near-perfect reproductions for binaries of some packages that did not aim to be reproducible. So far, we contributed three new packages to RC's repository via *pull requests*, all of which were accepted by their developers.

This paper presents the following main contributions:

1. A comprehensive study of the aspects that impact reproducibility in Maven Central.
2. Heuristic approaches for recovering the repository link, release tag, JDK version, and line endings for a given Maven package.
3. A tool for the Automated Reproduction of Maven Artifacts (AROMA) and its evaluation to automatically reproduce Maven packages in three practical use cases.

The remainder of this paper is structured as follows: We present our Experimental Setup in Section 2.1. In Sections 2.2, 2.3, and 2.4, we elaborate on our research questions, the methodologies that we used to answer them, and the results of our investigation. We then discuss the implications, future directions, and threats to the validity of our work in Section 2.5. Section 2.6 presents the related work of this study. Finally, we summarize our work in Section 2.7. All source code and data for this paper are available in a replication package [87].

## 2.1 Experimental Setup

The primary focus of this study is the automated reproduction of Maven libraries to verify the consistency between source code and binaries. We investigate the feasibility of using only the source code and available metadata for a Maven package, without involving the project maintainers. Throughout this study, we present empirical findings at each step for various reasons. Firstly, these findings justify our data-driven decisions in the methodology. Secondly, the data assists in setting defaults in projects like RC. For instance, by identifying the most popular JDK versions and using them as the default for reproducing packages, we can improve the likelihood of correct identification. Lastly, this approach enhances the transparency and verifiability of our results.

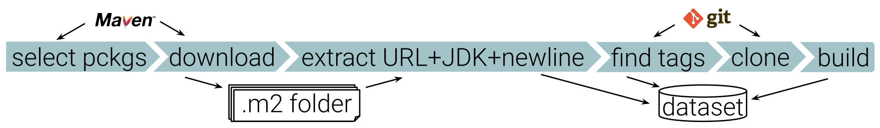


Figure 2.1: Overview of dataset creation

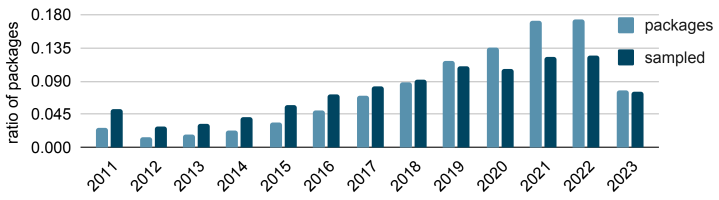


Figure 2.2: Maven libraries and selected versions by release year.

### 2.1.1 Dataset Creation

Our research requires data collection from both Maven and GitHub. Figure 2.1 provides an overview of the methodology we employ for our data processing pipeline. Maven Central publishes an index file weekly, listing all the released packages [88]. We utilized the index files available up to May 17, 2023. As of this date, Maven Central contained 10.3M released versions of 479,915 unique projects (`groupId:artifactId`). To ensure our dataset is representative of Maven Central and to avoid bias towards projects with frequent releases, we sample one random release (version) for each unique project. We do not opt for selecting only the latest versions since our ultimate goal is to reproduce all of Maven, not just the new releases. This includes packages not designed for reproducibility, especially those predating this feature in Maven, thus requiring older versions. Furthermore, as not all projects use the latest versions and some neglect dependency updates, including the older versions becomes important.

We developed a scalable infrastructure for parallel data aggregation of selected releases. The scalability of this process is constrained only by available computational resources and network bandwidth. Leveraging a powerful server with 256 cores, we completed the dataset population in 3.5 hours. Building the projects required an additional 8 hours. The infrastructure is extendable with custom analyzers, which can add supplementary information to the dataset. The system is deployed via docker-compose, simplifying both the recreation of the dataset and periodic updates. The dataset creation uses the local `.m2` repository as a cache, downloading all Maven files to it, including the `pom.xml` files and archives. This creation process also populates a Postgres database, which can be queried for insights about the ecosystem.

**Statistics** Figure 2.2 displays the total number of releases on Maven Central as well as the number of selected releases, grouped by their release year. This figure clearly demonstrates a significant growth in the number of libraries released over time. The plot indicates that our random sampling strategy yields a time-based distribution of releases that mirrors the overall distribution of Maven releases. For our experiments, we successfully downloaded

Table 2.1: Studied dataset stats

| Set | #Packages | % |
|---|---:|---:|
| Indexed packages | 10,333,041 | 100.0 |
| Unique group:artifact combinations | 479,915 | 4.6 |
| Sample size of study | 479,915 | 100.0 |
| Successfully downloaded | 473,352 | 98.6 |
| Have archives | 410,102 | 86.6 |

the majority of the selected packages, with only 1.37% failing to download. A manual inspection revealed that most download issues were related to parent POMs not being hosted on Maven Central. Our sample is detailed in Table 2.1. As this table shows, we successfully downloaded 473,352 packages, out of which 410,102 had archives and were included in our study.

## 2.1.2 Reproducibility

The scope of this study is limited to Maven builds. Reproducing a Maven package presents numerous challenges. The most significant challenge is that build results depend on the build time. For instance, the current time may appear as the change time of the `.class` files or be included in embedded documentation. While adjusting the system time on the build machine can make smaller builds reproducible, this method is fragile. Even minor performance differences in the second build system can lead to varying checksums.

To address this issue, the Maven build plugins support the `project.build.outputTimestamp` property for all time-related output and operations. This property can be specified in the POM file to fix a particular timestamp, facilitating reproducibility. Using this property addresses *most* sources of variability in the default Maven plugins. However, even though the property can be set automatically during a build (e.g., through the `maven-version-plugin` [89]), the vast majority of projects do not define it. In our dataset of ~480K projects, we could only find 9,603 uses. Unfortunately, this property is not a silver bullet; other sources of variability also exist that can hinder successful reproduction.

**Variability Sources** The RC project stands at the forefront of reproducibility efforts in the Maven ecosystem [70]. The project has identified all sources of variability that typically exist in the build environment of Maven packages and has introduced a `.buildspec` format to formalize these build parameters for reproduction. A `.buildspec` requires the following information for reproduction:

- A valid link to the repository and the exact tag used to mark the released commit.
- The JDK version used for the compilation and build execution.
- The type of newline characters used on the build platform (e.g., \n or \r\n).
- The building tool and the specific build command for the release.

The RC project offers advanced tools that utilize the `.buildspec` files to validate the reproducibility of a Maven build. Moreover, the project maintains a manually curated list of reproducible Maven packages and provides their corresponding `.buildspec` files.

**Reproduction** Our study investigates the feasibility of automatically generating `.buildspec` files from the metadata that is available for Maven packages. The following sections introduce our heuristics that we devised to recover the required build environment of a release. After collecting all the information, we generate `.buildspec` files and compare our results with the manually curated ones provided by RC. We also use the RC tooling to attempt an actual reproduction of the release. RC employs Docker containers to build projects, consistently resulting in the same output. We reuse their build scripts and do not modify their logic. This means that when comparing the local binary using RC scripts with the original binary on Maven, the comparison is effectively between the outputs of two different build environments: one from our build and one from the release on Maven, which the maintainers of the library previously built.

## 2.2 RQ1: Can we recreate the link to a source code repository and commit of an artifact?

Maven releases are often distributed with a source archive, however, these archives do not contain the required build files. As such, it is crucial to recover the source code from the original project repository, which can be linked in their POM. However, this linking presents several challenges.

**Non-Existing and Ambiguous Information** Despite being mandatory information for Maven releases [90], not all projects link the source code repository in their POM. The links that do exist suffer from ambiguity, as there are multiple fields in which developers can declare the URL:

**`project.url`** Projects should link to their homepage, but many include the repository URL here.

**`package.scm.url`** A public URL of the source code repository.

**`package.scm.connection`** A link to a read-only version of the source code repository.

**`package.scm.developerConnection`** Similar to `package.scm.connection`, but with write access.

Despite the clear recommendation to specify the repository URL in the `package.scm.url` field, in practice, all fields are used interchangeably. The Maven manual defines the format for each field [71], e.g., the `package.scm.connection` should be `scm:<provider>:<provider-specific>`. However, this format is not enforced and is not strictly adhered to by many.

**Validating links** We need a valid repository link to rebuild releases from their sources. Therefore, we verify the provided repository links, similar to previous work that studied the prevalence of broken links on Stack Overflow [91]. This step is not only crucial as a starting point for our work but it also provides first insights into linking practices and popular hosting platforms.

Our approach is limited to Git, currently the most prevalent version control system [92]. We leverage its `ls-remote` command, which requests references to branches and tags from a remote repository without the need to clone or fetch from it first. Successful execution conclusively demonstrates the validity and accessibility of the queried repository. We validate all four fields separately, storing the results. Links failing this validation are either invalid or associated with different version control systems.

**Release tags** After identifying the repositories, we need to pinpoint the exact commit that has been released for successful reproduction. Using release tags [93] is a common strategy to mark releases in a repository. In Git, tags are unique markers for distinct repository states. They can be used to label pivotal milestones, like major versions or feature releases. As such, we inspect the tags in the source code repository and try to find tags that correspond with the version of the Maven release. Unfortunately, the format is not unified, and a wide range of tagging schemes exists in practice. For instance, a release of version `1.0.0` could be tagged with `v1.0.0`.

### 2.2.1 Where do Maven projects host their source code?

The established repository links allow us to investigate which hosting providers are commonly used for the source code repositories. Combined with the timestamp of each release, we can also analyze an evolution throughout the history of the ecosystem. Such insights highlight the interdependency between Maven, which serves as a binary ecosystem, and the code repositories that function as source code ecosystems.

**Methodology** To examine the market share evolution of each repository host, we parse each URL field, extract the hostname, and group the hostnames by year. For each year, we count the number of repositories for each service and determine the percentage market share of each hosting service by calculating the ratio of its count with the total number of repositories found in that year.

**Results** Overall, 83% of packages with `scm.url` utilize GitHub. Figure 2.3 depicts the market share of all source code repositories over time for this field. Moreover, since other URL fields exhibit similar patterns, our focus here is exclusively on the results pertaining to the `scm.url` field. Given the abundance of repository hosting services, we included only the five most popular ones and grouped the remaining ones into the *others* category. As illustrated in Figure 2.3, GitHub experienced rapid growth from 2011 to 2014. During this period, other repositories, especially SVN, maintained some degree of popularity. However, they were mostly replaced by GitHub in recent years. By 2015, GitHub had secured a market share consistently exceeding 85%. In addition to GitHub's dominance, an emerging trend indicates a rising preference for alternative repository hosts. Among them, *gitee*, *git-wip-us.apache*, and *gitbox.apache* are particularly noteworthy. However, these repositories are mostly exclusive to Apache projects and are often mirrored on GitHub. The figure also captures Apache's shift from the `git-wip-us.apache.org` domain to `gitbox.apache.org` in late 2018 [94].

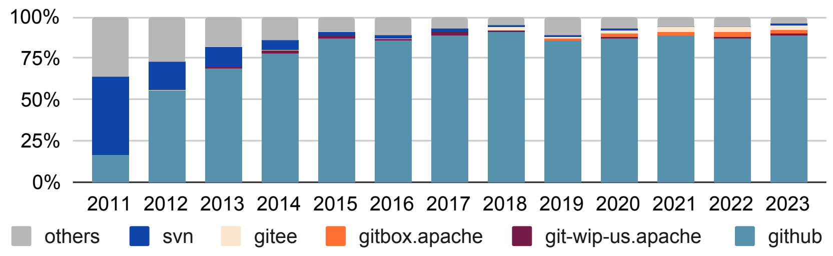


Figure 2.3: Market share of repository hosts per year for package.scm.url field.

### 2.2.2 How reliable are the source code repository links on MAVEN?

It is not enough to extract a seemingly valid URL, it is important for our evaluation that we ensure that these links point to the correct repository. First, if we build a different repository than RC does, the results of our builds will naturally differ, making them incomparable. Second, discovering new Maven reproducible packages is of little value if we are building the incorrect repository.

**Methodology** After obtaining a valid repository link for each package, we compare them to the repository links of RC `.buildspecs` to assess the accuracy of our approach. Since RC is mainly created manually, it serves as a good benchmark for evaluating our method. The manually curated `.buildspec` files are only *one possible way* to reproduce a release; other `.buildspecs` could lead to the same result. Using these files in our evaluation gives us a lower bound, and even a negative match might lead to a reproducible build in practice.

We recognize that several URLs in a `.buildspec` point to mirror repositories and a direct link comparison would result in many mismatches. As such, we decided to rely on the command `git ls-remote url` HEAD for comparison, which returns the commit hash of the default branch. If the hashes of both URLs match for a particular package, we consider the repositories to be identical.

**Results** Out of the 473,352 packages that we downloaded, 442,360 (representing 97.34% of packages with at least one `url`) provided a repository URL in the `scm.url` field. This high percentage is attributable to the mandatory requirements of Maven libraries [95] specified by the ecosystem. Nevertheless, not all developers utilize the `scm.url` field as intended. Some include a URL pointing to the project's homepage rather than the source code repository, which should be specified in the `url` field instead. Conversely, there are instances where developers either place the source code repository URL in both fields or solely in the `url` field. Some even provide the SSH URL for the repository, which might require authentication. However, converting such a link to an HTTPS URL makes it accessible. We automated this conversion process.

Table 2.2 shows the results of our URL validation. The left part lists the percentage of packages that contain each field, while the right part breaks this number down into the percentage of packages for which the identified link is valid. Notably, a significant

Table 2.2: Percentages of URL field usage and valid URLs for each

| | **Found** | | **Valid** | |
|---|---|---|---|---|
| **Metric** | #Packages | % (of Total) | #Packages | % (of Found) |
| Has ≥ 1 URL field | 448,273 | 98.6 | 354,948 | 79.2 |
| Has url | 442,379 | 97.3 | 166,742 | 37.7 |
| Has scm.url | 442,360 | 97.3 | 327,744 | 74.1 |
| Has scm.connection | 425,999 | 93.7 | 224,769 | 52.8 |
| Has scm.developerConnection | 356,982 | 78.6 | 117,912 | 33.0 |

percentage (74.09%) of links in the `scm.url` field were validated, while a considerably lower percentage (37.7%) in the `url` field met the same criterion. Since the `url` field is intended for the project's homepage, this difference is expected; after all, a webpage cannot be verified as a repository. However, the presence of 37.7% valid repository links in the `url` field also indicates that many projects use their repository as their homepage.

Overall, 79.2% of the packages have at least one verifiable repository link. Of the packages with invalid links, 6.87% contain URLs that are unparseable and could not be validated. These cases include malformed links, empty strings, and strings that are not URLs.

In total, our dataset contains 1,517 packages for which RC offers a `buildspec`. 97.5% of the URLs we identified align with the repositories that RC also references. Through manual inspection of the 2.5% non-matching cases, we determined two categories of mismatch:

- Instances where RC provided a URL requiring authentication for access, while the URL we identified is public and matches the package.
- Cases in which RC offers a link that results in a `404` error, but our URL points to a relevant public repository corresponding to the package.

Most importantly, we did not encounter any instances where our method provided a link that contradicted the information in RC.

### 2.2.3 Can we find the corresponding version tags in source code repositories?

Developers of a project can leverage release tags to roll back to a specific version for tasks such as debugging and bug fixing. Yet, from Maven's perspective, tags remain hidden. If the conventions developers use for Maven releases differ from the tags in their codebase, then only a developer's historical knowledge can bridge this gap. In this research question, our objective is to automate the mapping between Maven releases and Git tags.

**Methodology** To discover more version mappings, we do not limit our tag search to exact matches and inspect all tags used in the validated repositories to find recurring tagging practices. From this list, one author has identified patterns, calculated their frequency, and removed their matches until no other repeating patterns existed. We verified each pattern by ensuring that a second author cannot find mismatches in 10 randomly selected matches

Table 2.3: Top 10 tagging patterns

| Pattern | Example coordinate | Tags | %Packages |
|---|---|---|---|
| `v`*«version»* | com.ethlo.dachs:dachs-audit:1.0.0 | v1.0.0 | 48.2 |
| *«version»* | com.eriwen:groovyrtm:2.1.1 | 2.1.1 | 34.9 |
| `artifactId-`*«version»* | com.expedia.tesla:tesla:4.0 | tesla-4.0 | 6.4 |
| `p1-`*«version»* | org.activiti:activiti-form-api:6.0.0 | activiti-6.0.0 | 5.3 |
| `release-`*«version»* | com.senseidb.clue:clue:0.0.2 | release-0.0.2 | 1.5 |
| `p1-p2-`*«version»* | com.aoapps:ao-dbc-book:3.1.1 | ao-dbc-3.1.1 | 1.2 |
| `release/`*«version»* | com.github.dalet-oss:vfs-gcs:2.2.0 | release/2.2.0 | 0.4 |
| `p1-p2-p3-`*«version»* | com.adobe:aio-lib-java-ims:0.0.4 | aio-lib-java-0.0.4 | 0.3 |
| *«version»*`-release` | com.github.machaval:cli_2.12:0.2.0 | 0.2.0-release | 0.3 |
| `v.`*«version»* | dev.struchkov.haiti:haiti-core:1.0.3 | v.1.0.3 | 0.2 |

for each pattern. Overall, we identified 45 common tagging patterns, and our replication package contains the full catalog.

**Results** From a total of 354,948 packages that have at least one valid URL, we successfully identified 234,674 (66.1%) tags using the tagging patterns. Packages that did not align with our patterns either lacked version tags or adhered to a specific convention unique to their project, which differed from those we identified. Table 2.3 presents the prevalence of our top 10 tagging patterns and also showcases examples of tags used in Maven libraries. The most common pattern shown in the table is `v`*«version»*, which indicates that the package version that can be derived from the Maven coordinate is prefixed with the character *v*, leading to tags such as *v1.0.0*. Notably, this pattern is present in 113,243 packages within our dataset. Additionally, some packages incorporate their `artifactId` or parts of it into their tag names. An `artifactId` might consist of several parts, each separated by a dash (`-`). We label these parts as `p`*«number»*. For illustration, the package `com.adobe:aio-lib-java-ims:0.0.4` contains four parts (p1: `aio`, p2: `lib`, p3: `java`, and p4: `ims`). Within our dataset, 639 (0.3%) packages follow the pattern `p1-p2-p3-`*«version»*. The first two patterns listed in the table account for 83% of the cases we identified. Our results suggest that the vast majority of Maven libraries either use the version number as a tag or prepend the character *v*.

Overall, in 93.4% of the cases, the tags that we identified are identical to the ones RC suggests. We manually inspected the remaining 6.6% of tags and found two categories of mismatching tags:

**Commit Hashes** RC references a commit hash instead of a tag label. In some cases, it is the same commit that our identified tag points to. Occasionally, when a branch was merged during the release, RC includes the parent or child commit of our identified tag. In these cases, merge commits were used solely for adjusting the versions, so the difference between RC and our approach is negligible. We found two instances where RC specified a seemingly unrelated commit. In the first instance, a comment in the `.buildspec` notes that the binaries on Maven Central were not built from the tag, which suggests an oversight or poor practice by the library maintainers. In the other instance, the Maven plugin version has changed in the commit of RC. We speculate that the package developers attempted to update a plugin, which made this commit unreproducible. RC might have realized this and instead decided to point to the correct commit instead of the tagged one. However, without better documentation or practices for such cases, it is challenging for others to interpret or automate them.

**Tag Names** RC pointed to a different tag name than what our method identified. In some cases, repositories featured identical releases but with different tag names; one was detected by our method, and the other was found by RC. When comparing the content of both releases, they were identical. Thus, our approach is essentially the same for such cases as RC. In one instance, RC pointed to a tag that did not exist in the repository. However, a branch with the same name was present, enabling a checkout and build. Nevertheless, the tag we identified was also accurate since it pointed to the last commit of the same branch. In another instance, we pinpointed a tag as `1.0.0`, while RC referred to `org.apache.felix.feature-1.0.0`. We noted in the repository of this package that developers versioned each sub-module individually. Thus, `1.0.0` represents the version `1.0.0` of the entire project, whereas `org.apache.felix.feature-1.0.0` pertains to the release of the particular sub-module we were looking for. In this particular example, our approach may not be able to find the accurate release; however, this pattern of versioning is not advised. Consistent versioning across the entire project is the recommended practice. We observed that such cases are exceptions and do not occur frequently.

In conclusion, our observations did not reveal any intrinsic flaws in our approach to identifying tags. However, during the manual inspection, we found that there is a lack of standard practices and procedures when it comes to the reproducibility of packages and releases in general, which makes it even more challenging to automate.

RQ1: Can we recreate the link to a source code repository and commit of an artifact? Yes, for a total of 354,948 packages, we identified at least one valid repository URL. Among these, we successfully pinpointed commit tags for 234,674 using our identified tagging patterns. In summary, for 57.2% of the packages, we can recreate both their link and the commit associated with the artifact.

## 2.3 RQ2: Can the original build environment be reconstructed after the fact?

After obtaining the source code, a reproduction attempt requires building the package. Additional details about the original build environment need to be defined in the `.buildspec` to use the RC tooling for the reproduction. We have devised several heuristics to recover these details from the source code or the released artifacts.

**JDK** Java is the primary language for JVM-based programs that are released on Maven and the development of Java programs requires a *Java Development Kit* (JDK). The chosen Java version notably affects the development and release process of libraries. New Java versions introduce new features, refine existing ones, fix bugs, and address vulnerabilities. As the compiler changes with each version, it also has an important effect on a reproduction attempt for a library from its source code, as the usage of different Java versions for compilation can result in different bytecode.

To collect data about the JDK version used to originally build the libraries, we employ two heuristics. The first heuristic focuses on the archive's metadata file `META-INF/-MANIFEST.MF`. If present, this file usually provides information about the JDK version in the `Build-Jdk-Spec` field; older versions of the Maven archiver [96, 97] have used the `Build-Jdk` field instead. The `Build-Jdk` field typically contains detailed version data, including the minor version. For the purpose of rebuilding the packages, we only need the major versions. As such, we extract only that major part. For instance, from the version `11.0.10`, we only retrieve `11`. It is worth noting, however, that a significant number of archives lack these entries. In such cases, we fall back to the configured compiler version, which can help reduce the *search space* of JDK candidates that could have been used for the release.

**Compiler Configuration** The source code compilation of Maven projects is configured within the POM through the *Apache Maven Compiler Plugin* [98]. While third-party plugins might also support compilation, we focus solely on the *Apache Maven Compiler Plugin* as the main and default compilation tool. It allows specifying the expected version of the Java `source` and the `target` version of the generated bytecode. Knowing about these versions eliminates JDK versions that precede these releases as viable candidates, as only newer JDKs can compile to old versions, but not vice versa. We use the extracted versions as the lower bound for the JDK version and build the project using this specific version and all newer releases with *Long Term Support* (e.g., Java 11, 13, 17) that were available at the release date of the library. With the assistance of the *file-level comparison* (See Section 2.4), we can then identify the best version that maximizes reproducibility. In cases where we cannot find any version for the package, we try all *LTS* versions.

To streamline the information extraction process from the POM, we utilize the built-in model of Maven [99]. This model offers a standard structure for accessing and extracting the required data from the POM. It is important to understand that POM supports numerous features for managing and inheriting properties among different modules. For instance, projects without actual Java code can still possess compiler configurations. These projects are typically shipped as *pom* packages. They act as containers, and sub-modules can inherit configurations and dependencies from them. In this study, we consider such inheritance from the project's parent, configurations of plugins, and *pluginManagement* [100]. We also account for properties and Maven default values.

**Line ending** Line endings or newline characters denote the end of a line in a text file and the beginning of a new one. These characters play an important role in formatting and displaying text across various platforms and editors. The two common line-ending conventions are `lf` (Line Feed) and `crlf` (Carriage Return plus Line Feed). `lf` is primarily used in Unix-based systems, including macOS and Linux, and is represented as \n. Moreover, `crlf`, which is common in Windows environments, is represented as `\r\n`. Inconsistencies in line endings can result in incorrectly displayed text or errors in certain contexts. Line endings can also impact reproducibility, especially in build-generated files like `MANIFEST.MF` and `pom.properties`.

Providing the correct line endings used during the package release in the `.buildspec` is essential, allowing the RC tooling to eliminate variations caused by varying line endings. If the newline is `crlf`, the argument `-Dline.separator=$'\r\n'` is applied when building the project. Additionally, during the git repository fetch, all newlines are converted to `crlf` using the *unix2dos* tool.

To identify the correct line ending, we cannot examine source files from the repository, as committed files were created by a developer on a separate computer. Additionally, files can also get altered by Git on checkout. Instead, we rely on the `pom.properties` file, which is auto-generated during the build and then added as metadata to the artifacts. This file presents a reliable source as its line endings reflect the environment in which Maven built the original release. In cases where we cannot extract line endings from the file, we follow a trial and error approach and use both `lf` and `crlf` to build the package. If one fails, we can repeat the reproduction with the alternative.

**Build Tool** During our project selection, we search for the `pom.properties` file in the archive, which is automatically added during the build process. Build tools that generate the file, usually include the tool name in a comment. Since the scope of this study is limited to Maven builds, we parse the comment and ensure that we only select packages that were generated by Maven. If we cannot locate this file, the project is likely built using another build tool, such as Gradle or sbt.

We follow the default of RC and select Maven version 3.6.3 [101]. The sources of variation in a build that prevent reproducibility are mainly caused by the chosen Maven plugins, so the exact version used is negligible, as long as it is new enough to run all the plugins of a build.

**Build Command** The configuration-based design of Maven makes it possible for the actual build command used to create a release to be reduced to basic instructions. While the details can differ for each project, such as relying on release profiles or certain environment variables, RC has found a meaningful default command that works across a wide range of projects. Their recommendation is to activate the `-DskipTests`, `-Dmaven.javadoc.skip`, and `-Dgpg.skip` flags to skip test execution, the generation of *javadoc*, and the GPG signing process. None of these three steps affect the generated binaries; therefore, they can be safely skipped. Notably, the signatures are stored in separate files, and without knowing the private keys, reproduction would be impossible.

For our reproduction attempts, we adjust this default command of RC with further options. Since our recovered repositories potentially contain multi-module projects, we want to avoid building the whole repository every time, which would slow the build and increase the risk of a build failure. Therefore, we add the `-pl :`*«artId»* parameter, which limits the build to only the specific artifact that we are trying to reproduce. We also include the `-am`/`–also-make` flag to ensure that dependencies to other modules in the multi-module projects get built as well; otherwise, the build would fail. Ultimately, we use the command `mvn clean package -pl :`*«artId»* `-am -DskipTests -Dmaven.-javadoc.skip -Dgpg.skip` to build the packages for reproduction.

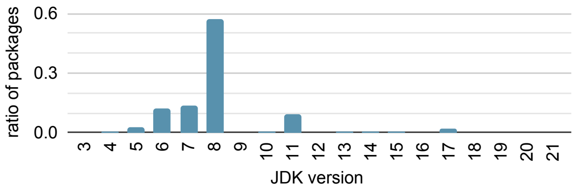


Figure 2.4: JDK versions used by Maven libraries.

### 2.3.1 Which JDKs were used to build Maven libraries?

Finding the correct JDK is not only one of the most fundamental steps in our reproduction efforts, but it also allows us to study the distribution in the ecosystem. This can, for example, help library maintainers choose versions that result in more compatibility, as well as help RC maintainers try the most prevalent versions as their default JDK.

**Methodology** To find the build JDK versions, we analyze the `MANIFEST.MF` files of the packages in our dataset. Out of 410,102 packages with archives, 204,394 (50%) defined the `Build-Jdk` field, and 41,809 (10%) defined the `Build-Jdk-Spec` field in their manifest files. In this section, we report the statistics of analyzing the `Build-Jdk` field as more packages define this field.

**Results** The predominant Java version utilized is Java 8, accounting for a significant 57% (116,607) of the packages. When considering the *Long-Term Support* (LTS) versions, namely Java 8, 11, 17, and 21, they collectively make up about 68.5% of the archives. As the newest JDK version (Java 20) was released only two months before this study was conducted, it is not surprising that we could only find 52 packages that use it. It is noteworthy to mention that older non-LTS versions, specifically Java 5, 6, and 7, have a considerably higher presence compared to more recent non-LTS versions. A comprehensive breakdown by major version can be observed in Figure 2.4. Only one artifact, specifically [102], appears to be compiled in Java 21. Given that Java 21 has not been released at the time of writing this article, it is likely that this artifact is built using an early-access version.

Figure 2.5 illustrates the distribution of JDK versions per year. This figure includes only the top six versions, comprising three LTS versions (8, 11, 17) and three non-LTS versions (5, 6, 7). It is important to note that other versions are excluded from this figure, which is why some bars have values less than one. The figure highlights the relatively rapid adoption of LTS versions (Java 8, 11, and 17). These versions become among the most popular choices within two years of their respective releases. For instance, Java 8 was released in 2014 and became the most popular version by 2016 with a share of 26.6%. Interestingly, Java 8 maintains its dominance in the ecosystem even after newer LTS versions are introduced. The figure also highlights a pattern of quick growth followed by a prolonged decline for non-LTS versions. However, this pattern changes after the introduction of the first LTS version, Java 8. Its dominance in the ecosystem persists much longer than the popularity of preceding versions.

Our analyses have revealed that the artifact compiled in early-access Java 21 is not a unique case; we found several other instances of compilation in previous early-access Java

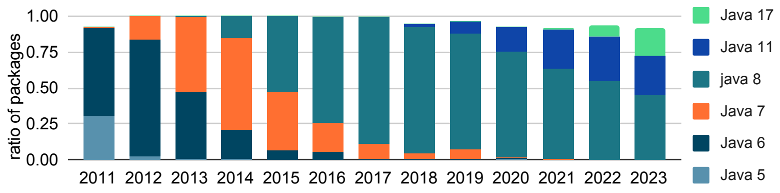


Figure 2.5: JDK versions used by Maven libraries over the years.

versions as well. We can also still find new releases in Java versions that are no longer officially supported. This trend appears in approximately 21K packages. It is important to note that this is only an estimation due to the limited information available regarding end-of-support dates for older versions. Furthermore, if these dates are available, they can vary by multiple years depending on the Java vendor.

The next method to recover the JDK version involves examining the compiler configuration, as discussed in Section 2.3. In many instances, the Maven compiler plugin is not configured, accounting for 45.02% (213,106) of the packages. However, it is configured in 260,246 packages. When the Maven compiler plugin is utilized, both the `source` and `target` versions are specified in 95.89% (249,542) of the cases. On the other hand, neither one is specified in 3.92% (10,202) of the packages. Interestingly, even though it is designed to allow different source and target versions, in almost all cases where both are present, they match. Only a tiny fraction (0.15%/385) specify different versions. Although this feature serves its intended purpose, it is rarely used. Exclusively specifying just one of the two versions is also rare: a mere 376 (0.14%) packages specify only the source version, while 126 (0.05%) packages utilize only the target version. 64% of packages that specified both JDK and source versions use the same version for both. However, 35% use a newer JDK for building than the one specified for the source. In rare cases (0.0012%), there are even source fields that are greater than the compiling JDK. These cases might be mistakes in the manifest files since older versions cannot compile more modern source code.

When compared to RC, our identified JDK version matched in 99.5% of the cases, and we manually inspected the very few differences. Determining the exact version that library maintainers used to build the released library is challenging since none of these projects have a manifest file. RC also does not provide details on how they arrived at the versions used in these packages. However, we discovered GitHub workflow files in the corresponding GitHub releases of these packages, and the Java version used in their CI matched one of the LTS versions that we had identified. In contrast, RC assigned non-LTS versions to these packages, which we could not verify within these projects.

### 2.3.2 What line endings do Maven libraries use?

For the reuse of a package, it is irrelevant on which platform it has been built. However, understanding the distribution of build platforms can guide future research on reproducibility.

**Methodology** As explained in the introduction to this section, we extract the information about line-endings from the `pom.properties` file. We calculate basic statistics about the file endings and group the results by year.

**Results** Overall, 63% (258,809) of the Maven libraries with archives possess the `pom.properties` file. Libraries without this file are likely built using tools other than Maven, such as Gradle. Interestingly, 212,340 (82%) of the packages employ `lf` line endings, while only 46,469 (18%) use `crlf`. A historical analysis of the distribution throughout the history of Maven has revealed a consistent ratio of roughly 80/20 in favor of `lf` over the years.

Our identified line ending aligns with RC's results in 98% of the cases. In the few instances in which the results differed, RC employs `crlf-nogit` and `no-auto.crlf` as line endings, while we utilize `crlf`. We were unable to determine the reason RC adopts these line endings, as our search through their documentation and code yielded no answers, but we assume that this hints at the way Git is (not) supposed to take care of the line endings [103]. Upon checking the `pom.properties` of those packages, we found that the releases used `crlf`. Thus, we are confident about our results.

RQ2: Can the original build environment be reconstructed after the fact? We automatically extracted the JDK version and line endings of releases after the fact. When compared to RC, in 99.5% of the cases, our identified version matched, and regarding line endings, our results matched RC's in 98% of the cases.

## 2.4 RQ3: To which extent is it possible to automatically reproduce Maven artifacts?

The final research question investigates the overarching research question of this work and analyzes whether an automated approach for reproduction can be successful. For this effort, we select the subset of packages from our dataset that contains all the required metadata. Out of the 479K packages, 235K (49.0%) have verified links and recovered tags, 118K (50.2%) of these contain a `pom.properties` file in their archives, and 112K (95%) of these packages are built using Maven. To address this research question, we focus exclusively on building these particular packages.

After building the libraries, we record the build's success status and store the compiler output. We then compare this output to the corresponding release on Maven Central. This comparison is facilitated by the RC script. Specifically, it compares the built files with a reference, in this case, the files released on Maven Central. While it outputs various data points, we focus primarily on two pieces of information: the `ok_files`, which represents files that match the reference, and the `ko_files`, indicating files that diverge from the reference. Given that RC builds contain entire repositories, a direct comparison of all generated files would not yield meaningful insights. This is because the package under examination might be just a sub-module of the larger repository. A sub-module build generates fewer files compared to the building of the full project. However, as highlighted in Section 2.3, while building the sub-module we also build its associated dependencies. This results in a more extensive file set than what was released for this specific package on Maven Central, as the set also contains dependency files. To fix this, we curate a set of files directly related to the Maven package in question and limit our comparison to these files. To collect this set, we send a manual request to Maven Central and parse the `html` list of all files linked to the specific package. This list includes files such as JARs, sources, docs, and POM files.

Reproducibility is a binary concept. An executable is either reproducible, resulting in an identical checksum, or it is not, generating a different checksum. Sometimes, reproducibility is compromised by trivial factors like a timestamp in one file. At other times, it can be due to differing content in multiple class files. Perfect reproducibility facilitates automatic checking but builds often are not reproducible. Thus, determining why a certain executable is not reproducible becomes valuable. Tools like diffoscope [104] exist that can compare archives *line-by-line*, but this is too fine-grained for an ecosystem-scale analysis. As such, we focus on analyzing archive contents *file-by-file* to provide a broader perspective on causes of non-reproducibility. Given our automatic rebuild, having an automated way to understand how close we are to achieving a perfectly reproducible build is helpful. To achieve this, we compute the MD5 checksum for each generated file and compare these with the corresponding checksums in the reference archive to pinpoint the differences.

## 2.4.1 Can AROMA reproduce reproducible packages?

AROMA is a tool that automatically extracts build information for a given package. It starts in Maven Central, follows repository links to re-establish a source connection, and extracts the necessary information to generate a `.buildspec` file.

The previous research questions have shown that we can replicate the `.buildspec` files of RC with a high success rate. In this experiment, we will now investigate how our approach compares to RC on an end-to-end basis. This is important to investigate because the main purpose of all previous steps is to automate the rebuild process that RC is conducting manually.

**Methodology** We rebuild packages using both AROMA and RC's `.buildspec` files, then compare the results. We select 100 random packages for which RC provides a `.buildspec`. First, we run the RC script to reproduce these packages using RC's provided `.buildspec` file. Subsequently, we execute the RC script using the `.buildspec` generated by our approach. Finally, we compare the number of packages that each method successfully reproduces.

**Results** Out of the 100 packages we examined in this experiment, our approach successfully built all of them. However, using RC's `buildspecs`, we only managed to build 96 packages. A manual inspection revealed that one of these `buildspecs` contained a broken repository URL, resulting in a `404` error. We noticed that in the RC repository for the newer versions of this package, the link had been updated to match the one we found, but all older versions were still using the broken link. We speculate that the library maintainers changed their repository at some point, and the RC maintainers overlooked updating the links for earlier versions. Nevertheless, when our build was compared to the Maven Central release of this specific package, all files matched. Since we could not build this package using RC's approach, we compared the Maven files with the `ok_files` present in the RC repository for this package. We found one POM file to be missing. In the other three instances where the RC build failed, we noted that the command section of the `buildspec` initiated with a SHELL instruction. This command enforces manual intervention, as it prompts an interactive build process for the packages. Consequently, we excluded these from our experiment since we could not automatically build them. However, the build information we extracted was identical to what was present in RC's `buildspecs` for these packages.

Out of the remaining 96 packages, our approach achieved full reproducibility for 10 of them. Similarly, RC reproduced these packages in their entirety. For 37 packages, we managed to reproduce all released files except for the `sources.jar` files. In contrast, RC was able to reproduce these `sources.jar` files. Upon manual inspection, it became clear that the only difference between our build and the RC build was the *release profiles* specified in the command. These 37 packages would generate the `sources.jar` only when built using a specific release profile. It is worth noting that, from a security standpoint, the reproducibility of binaries is paramount, which both approaches achieve. Regarding the remaining 49 packages:

- In 11 instances, both our approach and RC reproduced all but the `sources.jar` files.
- For six packages, neither our method nor RC was able to reproduce the released files. In two of these cases, the build failed for both methods: one due to a `java.lang.-ClassNotFoundException` and the other due to a dependency issue.
- In our build, two of the packages could not be reproduced, though RC managed to succeed. The primary difference between our method and RC's for these packages was the *release profile*. These two packages specifically required a certain *release profile* for their build.
- For the remaining 30 packages, both methods achieved only partial reproduction. They reproduced some of the released files on Maven, but not all, and this time, the unreproduced files extended beyond just the `sources.jar`. We acknowledge that anything less than perfect reproducibility in binaries could potentially open the door to attacks. Our intention in discussing partial reproducibility is not to settle for less than perfect. Rather, it is to measure how far non-reproducible packages are from achieving perfect reproducibility. This approach helps us understand the underlying reasons for non-reproducibility and can inform future research about the most crucial missing parts.

Overall, our approach achieved very close performance compared to RC when reproducing the packages, despite our method being fully automated. In this experiment, we did not observe any cases where our approach had an inherent limitation.

### 2.4.2 Can AROMA identify new reproducible packages that are not listed on RC?

Manually creating and maintaining a list of reproducible packages can be challenging and prone to errors. One potential application of our approach is to assist RC in expanding and updating this list. In this experiment, we devise a scenario to identify reproducible packages not yet documented by RC.

**Methodology** As noted by RC's records, packages that aim for reproducibility do not always fully succeed, often, they are only partially reproducible. We explore these cases by randomly selecting 100 packages that have the `project.build.outputTimestamp` property in their POM file, indicating their attempt to be reproducible. We ensure that these packages do not exist in the current RC list. We then try to recover a `.buildspec` file and reuse the RC tooling to attempt reproduction.

**Results** We achieved 100% reproducibility for five packages. However, 44 of the packages failed to build. This highlights the challenges of rebuilding projects in the wild and highlights the value of automation in deploying reproducible packages and rebuilding them. For 23 packages, we managed to reproduce all files except the `source.jars`. For three packages, even though they were successfully built, we could not reproduce the files. Upon manual inspection, we discovered that the underlying reason was that these packages utilized *release profiles* for their releases on Maven Central. For the remaining 25 packages, no other files apart from the POM matched the reference.

Overall, the objective of this experiment was to determine whether our approach could be used to expand the RC's list of reproducible Maven packages. We pinpointed several packages suitable for inclusion: some with full reproducibility and others with partial reproducibility. As a result, we contributed a few of these packages to the RC repository through *pull requests*, and these were quickly accepted by the main developers of the RC.

### 2.4.3 Find Candidates: Is there any near miss among packages not tried reproducibility?

As discussed in Section 2, the efforts made so far cover only a limited part of the entire Maven ecosystem. Given this context, automated methods that assist in making existing packages reproducible are important. One application of our approach is to detect opportunities for straightforward reproducibility fixes. Addressing such cases, which can be made reproducible with minimal effort, contributes positively to the overall health and security of the ecosystem.

**Methodology** We select 100 random packages lacking the `project.build.outputTimestamp` property in their POM file, indicating they have not pursued reproducibility. After building these packages, we conduct a *file-level comparison*, as elaborated in Section 2.4, to identify the reasons for non-reproducibility and to search for near-miss that might be fixed with a small manual effort.

**Results** While the build of 39 failed, we were surprised that we managed to partially reproduce some of the other packages, even though they were not designed for reproducibility. For eight of these packages, we successfully reproduced the POM files and almost the binary files. The only difference between the binaries and the references was file metadata, like the file dates in the archive itself, suggesting that the content of these packages was otherwise identical. For another 26 packages, the only culprits for unreproducibility were the `Manifest` and/or `pom.properties` files. This indicates that the majority of the content in these packages aligned with what is available on Maven Central, and it is the files generated by build tools and plugins during the build process that render them unreproducible. For the remaining 27 packages, the unreproducible files also encompassed other types, primarily `.class` files. However, the prevalence of this was relatively limited. The package with the most unreproducible class files had 43 unreproducible class files out of a total of 86. The subsequent cases had ratios of 19 to 53 and 13 to 121, respectively. Apart from these exceptions, all other packages contained fewer than five unreproducible class files. We also observed an occasional presence of other types of unreproducible files, like XML, but these were largely specific to individual projects and did not form a recurring pattern.

We ignored the reproduction of source files and focused primarily on binaries, which are the most critical aspect of reproducibility. Sources are available through the recovered repository links. As we noted, numerous near-miss cases emerged during the experiment. Even though these cases with slight differences might currently not produce the same checksums across different builds, addressing this issue is straightforward. As a result, providing automated recommendations for reproducible releases is feasible in such scenarios. For instance, stripping the timestamp [105] or assigning it a certain value can make a package reproducible. To demonstrate this, we built a package where the timestamp was the only factor causing unreproducibility. Since Maven is immutable (a release cannot be changed after deployment), by manually synchronizing the system's time with that of the released package, we built the package with an identical checksum. Though this technique might appear cumbersome, it highlights the feasibility of automatically repairing the unreproducibility in many packages. This experiment reveals that although the situation on Maven may seem bad at first compared to, for example, Debian, it can be quickly mitigated. Many packages were *almost* reproducible, but achieving high reproducibility requires the attention of the community.

RQ3: To which extent is it possible to automatically reproduce maven artifacts? Overall, automation was possible for 23.4% of the packages, and only 8% of those were fully reproducible. When considering the magnitude of Maven, these percentages represent a significant improvement over a manual approach.

## 2.5 Discussion

The experiments in this paper have touched upon a wide range of areas and resulted in various insights. We use the discussion section to reflect on actionable insights or recommendations for future work.

**Awareness** Setting the `project.build.outputTimestamp` property in a Maven project increases the chance of automated reproducibility substantially. The only explanation for why this property is not widely used is a lack of awareness among developers. We postulate that Maven tools should start to print a warning about a missing definition during a build (similar to a missing definition of the encoding). As it does not hurt to define the property, Maven could even go one step further and set it automatically, when the property has not been set by developers.

Reproducibility should become a quality attribute of a package. Maven could begin warning about non-reproducible dependencies. Once library users start considering reproducibility as a factor in their library selection, library maintainers will have an incentive to put more effort into reproducibility. In the best case, providing `.buildspec` files could become a standard in the community.

On a related note, we found many different release tagging styles in our analysis. Developers need to realize that standardization facilitates automated processing and enhances the transparency of the ecosystem. Instead of reinventing the wheel, we recommend that developers adhere to the most popular community conventions for tagging their releases.

**Better Research** We currently observe that the absence of Maven source archives often leads to the exclusion of packages from studies that require source code analysis. For instance, in the study by Karakoidas et al. [106]. Recovering repositories and release tags for Maven packages not only enhances the likelihood of better reproducibility but can also have a positive impact on these research efforts.

**Sorry State of Maven Central** We find it worrisome to observe how many projects cannot be reproduced simply because the necessary resources are not available or are no longer accessible. Future work should focus on investigating the feasibility of removing broken artifacts from the repository to restore Maven Central to a fully self-contained and reproducible state. Broken packages no longer serve a purpose and impede maintenance efforts.

We believe that the strong separation between hosting binaries and source code, as designed in Maven Central, is the most likely explanation for this state. Perhaps the concept of a central repository is outdated and requires re-evaluation. Current practices on major hosting platforms like GitHub or GitLab highlight the distributed nature of Maven and bring sources and releases closer together in their integrated package registries. However, it is important to note that a fully decentralized system may emphasize the problems with data consistency that we have identified on Maven Central, where, at least in theory, regulations exist to maintain a basic level of repository hygiene.

**Relevance** Our experiments showed that automated reproduction could only be attempted for 112K out of 479K packages (23.4%), and out of the packages we attempted, only 24 out of 300 (8%) were fully reproducible, with 60 out of 300 (20.0%) being at least partially reproducible. While these numbers may seem small, they represent automated results that allow us to focus manual efforts on more challenging packages. If the same fractions were to apply to the 10M existing Maven packages, we would be able to automatically reproduce approximately $23\% \times (8\% + 20\%) \times 10M = 644K$ packages, significantly expanding the RC dataset beyond the existing 42K packages. We believe that AROMA has significant potential, and there are ample opportunities for further improvement in future research.

One direction could be a study on *release profiles* that we found in many projects. They are used, for example, to include or exclude certain files in a release, which obviously changes the outcome. As profiles are often defined in parent POMs, research methodologies must make sure to extract this information properly. Furthermore, profile names can be freely chosen; future work should investigate how to detect the profiles that need activation for a release.

A second direction could be to formulate reproduction as a search problem. The goal is to prune impossible environmental variables and perform a grid search to find a valid `.buildspec`. Even cases that are not or only partially reproducible right now can inform such an automated approach and support the manual creation of a `.buildspec`. Future work should extend AROMA and explore the localization and repair of unreproducible builds, especially cases in which large parts of the release could be matched. This would be invaluable for the ecosystem and could enhance its security.

### 2.5.1 Threats to Validity

The following limitations should be taken into account when interpreting the results.

**Internal Validity** We conducted manual inspections and implemented heuristics in this research, which might introduce human errors or overlook corner cases. To minimize potential issues, we carefully documented all of our steps. Furthermore, for the manual inspections, two authors were involved to mitigate the chances of mistakes. We also compared our results to a manually curated baseline to estimate precision at each step. We reused existing open-source tools for rebuilding packages, which are widely accepted and integrated with Maven. However, there is a possibility of inheriting any flaws that they might have. To mitigate this, we built a flexible infrastructure for others to integrate their tools and compare their results with ours. We also acknowledge that hidden bugs in our implementation could exist. To address this, we conducted multiple manual tests, created an extensive test suite, and made our data and code public for community review.

**Construct Validity** When extracting the JDK version for cases where it was not present in the manifest file, we utilized the Java version specified in the POM. While this version is not necessarily guaranteed to align precisely with the JDK used to build the project, it typically serves as a lower-bound approximation. This approach might not provide the maximum precision but it enables us to build the packages using various versions and then compare the results. This allows for selecting the most probable version that was used in the library release.

**External Validity** In our study, while implementing the heuristics, we focused on common patterns. For example, we concentrated solely on using Maven for building the packages. Similarly, our study was limited to git-based packages, excluding other version control systems like Subversion and Mercurial. We believe that this is only a small limitation that mainly affects releases that are older than a decade, as documented in our analysis of the popularity of hosting platforms. We encourage future work to replicate our study in other ecosystems like NPM or `PyPI`.

## 2.6 Related Work

**Software Ecosystems** Multiple studies explored various aspects of software ecosystems. Kula et al. [36, 107] investigated software updates and library life cycles in Maven. Bavota et al. [47] examined changes within the Apache ecosystem, exploring elements such as dependency graphs, project size, releases, and client behavior during dependency version upgrades. Düsing et al. [108] researched library updates for patching vulnerabilities and found that a considerable number of Maven vulnerabilities were patched before disclosure. Mir et al. [15] studied the propagation of vulnerabilities in the Maven ecosystem. Raemaekers et al. [28] developed the Maven Dependency Dataset, which promoted research into the software evolution of the Maven Repository. Keshani et al. [72] proposed a scalable call graph generation approach for Maven and used these call graphs to investigate the correlation between method popularity and breaking changes [79]. Karakoidas et al. [106] collected comprehensive code metrics associated with object-oriented design, package design, and program size in Maven libraries. Soto-Valero et al. [29] studied the occurrence of bloated dependencies in the Maven ecosystem. Soto-Valero et al. [17] researched library version usage and distribution in the Maven ecosystem. Mitropoulos et al. [30] investigated the Maven Central Repository using the FindBugs tool to establish a link between artifact size and bug count. Abdalkareem et al. [109] studied trivial packages in the npm ecosystem, while Cogo et al. [110] analyzed same-day releases in the npm ecosystem. Kula et al. [111] introduced the Software Universe Graph (SUG) for software ecosystems analysis and compared dependency update behavior between Maven and CRAN. Benelallam et al. [16] constructed an artifact-level model of the Maven Central Repository to identify duplicated artifacts, while Kanda et al. [112] explored the occurrence and duplication of inner JAR files within the Maven Central Repository's JAR files.

**Software Builds** Several studies delved into the software builds. Ma et al.[31] analyzed Maven archetypes, identifying prevalent schema patterns in POM files. Zhang et al.[113] presented BuildSonic, a tool that pinpoints and repairs configuration issues in both Maven and Gradle builds, subsequently enhancing build speeds. Tamaraw et al. [114] attempted to aid in the maintenance of build script files through static analysis. Keshani et al. [115] proposed a lightweight call graph generation approach for software builds. Tufano et al.[116] examined cases of uncompileable snapshots in Java projects that use Maven, discovering that most projects experienced such snapshots due to dependency resolution issues. Gazzillo et al.[117] identified all build configurations using their proposed Kmax. Sotiropoulos et al. [118] introduced BuildFS, which models build executions and identifies faults in build systems. Hassan et al.[119] proposed HireBuild, a history-driven approach for repairing build scripts. HoBuff [120] further refined HireBuild by taking into account both the current project and external resources. Lastly, Lou et al. [121] analyzed 1,000 build-related issues on Stack Overflow and summarized the fix patterns for three build systems: Maven, Ant, and Gradle.

**Reproducibility** In 1984, Ken Thompson reflected on the *Trusting Trust Attacks*. These attacks involve compilers being compromised to embed malicious Trojan horses into the compiled code [44]. Since these alterations are not evident in the source code itself, detecting them is especially challenging. In response to this, Wheeler et al. [69] introduced the concept of diverse double compiling. This method compares the compilation results of the same source code using different compilers to prevent malicious attacks. This research paved the way for the initiative known as reproducible builds [82]. This initiative offers a set of practices to enhance the reproducibility of software packages. Several studies sought to expand on these ideas. Holler et al. [122] explored diverse double compilations in embedded systems. Shi et al. [123] developed a unified build process along with a toolset for verifiable builds. RepLoc [83] and RepTrace [84], aimed to pinpoint the origins of reproducibility issues. RepFix [85] and ConstBin [124] work towards automatically fixing unreproducible builds. Another solution, DetTrace [125], introduces a container abstraction for Linux, ensuring reproducibility. Several studies delved into the state of reproducibility and associated challenges within the open-source ecosystem. Fourne et al.[126] explored the motivations, challenges, and solutions related to reproducibility by interviewing developers. Butler et al.[127] interviewed business managers to understand the adoption of reproducible builds in businesses, shedding light on the technical reasons for embracing these practices. Lamb et al. [81] discussed what it means to build software reproducibly. Carnavalet and Mannan [128] carried out an empirical study on reproducible builds within security-critical software, summarizing the practical challenges of reproducibility.

Despite the significant contributions of these studies, to the best of our knowledge, no current research addresses the issue of automatically reproducing Maven libraries. Our study aims to fill this gap and offer an in-depth understanding of library reproducibility in Maven Central.

## 2.7 Summary

While the software development community has made significant progress in enhancing the reproducibility of libraries, there remains a considerable gap in understanding and addressing this issue within the Maven ecosystem. The existing focus on Debian has left the Maven ecosystem relatively understudied, with RC's list of reproducible Maven libraries being both limited and challenging to maintain. Our research tried to bridge this gap by automatically finding the reproducible Maven packages and their build environments. This enabled us to automatically rebuild the source code of these libraries and compare the results with the files available on Maven. While the automation is still limited to a subset of all releases for which a basic set of features can be recovered, the results of our research are highly promising. Our experiments demonstrate that we can achieve a 99.5% similarity to manually crafted `.buildspec` files. Using our approach we found previously missing reproducible libraries and contributed them to the RC repository. Furthermore, we made our dataset and tools openly accessible to the public.

# 3 Frankenstein: fast and lightweight call graph generation for software builds

Call Graphs are a rich data source and form the foundation for advanced static analyses that can, for example, detect security vulnerabilities or dead code. This information is invaluable when it is immediately available, such as in the output of a build system. Call Graph generation is a whole-program analysis: not just the application, but also all its dependencies are processed together. Recent work has shown that even advanced static analyses can use summarization techniques to substantially improve runtime; however, existing analyses focus on soundness, and as such remain very expensive. When executed in the build system, which typically has limited resources, even powerful servers suffer from slow build times, rendering these analyses impractical in today's fast-paced development. In this paper, we aim to strike a balance between improving static analyses while remaining practical for use cases that require quick results in low-resource environments. We propose a summarization-based implementation of a Class-Hierarchy Analysis algorithm for call graph generation of Java programs. Our approach leverages the fact that dependency sets often do not change between builds: we can generate call graphs for these dependencies, cache their generation for subsequent builds, and using a novel *stitching* algorithm, Frankenstein, merge all partial results into a complete call graph for the whole program. Our evaluation results show that this lightweight approach can substantially outperform existing frameworks. In terms of speed improvements, Frankenstein surpasses the baselines by up to 38%, requiring an average of just 388 Megabytes of memory. This makes the proposed approach practical for build systems with limited memory resources. Despite these optimizations, our generated call graphs maintain a near-identical set of edges when compared to the baselines, achieving an $F_1$ score of up to 0.98. This summarization-based approach for call graph generation paves the way for using extended static analyses in

build processes.[1]

[1]This chapter is based on the following paper: Mehdi Keshani, Georgios Gousios, and Sebastian Proksch. Frankenstein: fast and lightweight call graph generation for software builds. Empirical Software Engineering, 29(1):1–31, 2024 [115].

Continuous integration and delivery have revolutionized modern software engineering. Many tools and analyzers are applied in the build pipelines to enhance developers' productivity (e.g., improving the handling of pull requests [129]). However, this comes at a price: increased build time, which hinders build servers from providing fast feedback. Developers have to decide between faster builds or more helpful analyses. Especially in open-source software, developers often use shared, resource-limited build services like GitHub or TravisCI, which require fast and lightweight tools. For instance, Dependabot [58] conducts relatively basic analysis at the *versioned package* level to identify vulnerable dependencies, which lags behind more advanced state-of-the-art approaches [130, 131]. While this is less accurate, it is still useful and fast, so developers widely accept the tradeoff. We envision leveraging the vast information on centralized infrastructures like GitHub for ecosystem-scale analyses to provide better support for developers, further emphasizing the need for scalable approaches.

Traditionally, *program analysis* aims at soundness as the key property and is performed through *whole-program analyses*, which are more precise but also expensive. Recent results show that even advanced analyses can use summarization techniques without sacrificing soundness [132]. However, despite substantial performance improvements, they are still expensive, which hinders their widespread adoption in practice, especially when high performance is needed. Several thought leaders in the field advocate for relaxed requirements on soundness and favor trading off soundness to make program analyses more relevant in practice [133]. Previous work has shown that it is possible to summarize static analyses and pre-compute certain parts of an analysis [132, 134, 135]. Recent work even demonstrates that pruning analysis results, while being highly unsound, can have positive effects for users, as it results in faster execution and higher precision of two static analyses [136].

Many advanced program analyses, such as the detection of vulnerable call chains or unused code, rely on a *call graph* (CG), an approximation of all call relations between different callables (i.e., methods) of a system. CGs are generated through powerful data and control flow analyses on the whole program, including all of its dependencies. Unfortunately, generating the CG of an average program can easily take minutes, even when only basic algorithms are used. The resulting CG will take up several gigabytes of memory for big programs (e.g., h2o [137]), often even more during the generation. This is neither practical for build systems nor for analyses of large codebases. While the former are limited by their resources, analyses of the latter are usually restricted to static information [138] because advanced analyses are beyond existing approaches due to time and memory requirements. We observe, though, that the largest part of a CG originates from dependencies, both direct and transitive. Research has shown that 81.5% of programs keep outdated dependencies [139], and it is evident that between the small and frequent changes of two subsequent builds, the likelihood of dependency change is even lower. We propose to leverage this and eliminate redundancy to speed up CG generation. The same idea applies in software ecosystems where a few popular libraries are included in many programs, so caching and reusing their analysis results pays off [72].

In this paper, we investigate the idea of pre-computing *Partial CGs* (PCGs) that store minimal information about isolated dependencies (e.g., their declared types and methods) and reducing the CG generation to combine these partial results in three distinct steps.

1) Resolve all (direct and transitive) dependencies of the program at hand. 2) Generate a PCG for each (isolated) dependency and remember the type hierarchies defined in the dependency. Similar to the caching of dependencies in a build job (to prevent re-downloading), pre-computed CGs can be cached across builds. They need to be computed once per dependency and can then be reused whenever this particular dependency is used. 3) Merge the PCGs using our novel *stitching* algorithm.

In contrast to previous work on summarization [132], we are willing to relax soundness guarantees and put a strong focus on investigating the trade-offs necessary for a practical approach. Schubert et al. [132] improve the speed of CG generation using summarization. They evaluate this formally on several projects. However, it is unclear how effective this approach is in practice when analyzing large projects with many or large dependencies. An example of this trade-off is our decision to pick Class-Hierarchy Analysis (CHA) for the CG generation. CHA is a very basic algorithm, but it only needs to preserve the type hierarchy and call site information. This is the least amount of information with which it is possible to generate a CG. The *Reachability Analysis* (RA) algorithm requires even less information, but the usefulness of the resulting CG drastically drops due to high imprecision. This makes CHA the most applicable approach for our use cases.

Advanced approaches, like ModAlyzer, can generate more accurate results and, through summarization, decrease CG generation by 90% (they report a reduction from 4h to 25min) [132]. However, this takes too long for our use case. Moreover, it is infeasible to preserve the required context information for complex algorithms on shared infrastructure, as the points-to-graph and CG quickly grow to enormous sizes. Existing studies found that practical static analyses are often not sound [140] or that pruning static analysis results through machine learning (ML) can result in increased usability [141]. While these are fundamentally unsound solutions, they are promising directions to make static analysis more practical. It is necessary to further investigate the trade-off between soundness, precision, and speed. In this study, we investigate these trade-offs using real-world experiments in great detail.

We demonstrate the validity of our approach in an extensive set of experiments that compare our *stitched* results with the state-of-the-art static analysis frameworks OPAL [142, 143] and WALA [144]. We use them for two tasks: first, we generate PCGs for isolated *versioned packages*. Second, we run them on whole programs to generate a baseline for comparing speed and correctness. Based on our experiments, we have observed that after the cache is filled, our primary use case of performing subsequent builds experiences a significant improvement in execution speed, ranging from an average of 15% to 38%. We further demonstrate that the CGs generated by our approach for the whole program are comparable to those generated by the baselines. A comparison of the resulting sets of callables in the CGs reveals that our approach achieves a precision of 0.93 for OPAL and 0.99 for WALA. Furthermore, it attains a recall of 0.97 for WALA and 0.95 for OPAL. Moreover, we illustrate that, on average, our approach requires between 388 and 432 Megabytes (Mb) of memory to generate a whole-program CG further highlighting its efficiency in real-world applications.

Our manual investigation shows that any deviations in precision and recall are not inherent to our approach and could be fixed with more engineering effort on the partial analysis. We believe that our approach is promising because the minor reduction in recall allows for improvements that make it practical for inclusion in CI tools.

Overall, this paper presents the following main contributions:

- Adopting an existing library summarization technique for CG generation through a novel *stitching* algorithm
- Revisiting existing evaluation methodologies with a focus on correctness, scalability, and memory consumption designed particularly to assess the practicality of our proposed approach
- Extensive evaluation on a real-world Maven sample

We release our tool, the dataset, and the analysis scripts on the artifact page of this paper [145].

## 3.1 Background

In this section, we first introduce the terminology that is utilized throughout this study. Subsequently, we provide an overview of existing studies related to our research.

### 3.1.1 Terminology

To ensure clarity and precision, we present definitions of the key terms used in this article. It is important to note that some of these terms may have different and overlapping meanings in various contexts. Therefore, we aim to establish a shared understanding of these concepts for the rest of this article. The definitions are as follows:

**Program** A *program* refers to a piece of code written in any programming language, regardless of its size or distribution method.

**Artifact** An *artifact* is a *program* distributed to users, offering features related to a core idea. *Artifacts* may evolve over time, addressing existing issues or adding new functionalities. In Maven, this is referred to as `groupId:artifactId`.

**Package** This term is synonymous with *artifact* and can be used interchangeably. *Package* is also commonly used to denote *namespaces* within *programs*. To avoid confusion in this study, we use the term *Java package* when referring to *namespaces*.

**Versioned-package** A *versioned package* is a snapshot of an *artifact* at a specific point in time. *Versioned packages* are released on repositories to be used by others. In Maven, this is referred to as `groupId:artifactId:version`.

**Project** A project is a domain containing multiple packages (`groupId`). While the term *project* may also be used to refer to *artifacts* or *versioned packages* in other contexts, in this study, we consistently use the term *project* to represent the domain holding a `groupId` and containing *packages*.

**Dependency** A *dependency* relationship occurs when one *versioned package*, such as *A*, includes another *versioned package*, like *B*, in order to utilize its functionality. In this scenario, *A* is considered the *dependent* program, while *B* is the *dependency*. In Maven, this relationship is established by adding a *dependency tag* to the *pom.xml* file of *A*.

**Library** This term is synonymous with a *dependency*, and the two are used interchangeably.

**Whole program** A *whole program* encompasses the entire code required for successful compilation, including any *versioned package* listed as a dependency (direct or transitive).

**Dependency set** A *dependency set* consists of the *versioned packages* required for a *program* to compile successfully. The process of determining this set is referred to as *dependency resolution*. The concept of a *dependency set* shares similarities with the *whole-program* notion, as both involve the collection of all necessary components for successful compilation.

**Application** An *application* refers to a *program* that is the current focus of attention. It serves as the root of a dependency set and the starting point for analyses, such as dependency resolution. The *application* is also included in the *dependency set*.

### 3.1.2 Related Work

**CG generation algorithms** There are various algorithms for constructing CGs, including[146–148]. The most basic algorithm is a RA which has multiple variants ranging from only considering method names to also considering the signatures of reachable methods. RA has been used in a variety of use cases in the literature, with pioneers such as Srivastava et al. [149] and Goldberg et al. [150].

Although RA is sound and fast, it significantly reduces precision. To address this Dean et al. [73] proposed *Call Hierarchy Analysis* (CHA) as a more precise extension. This algorithm matches the called method's signature to subtypes of the receiver type and finds implementations of the dynamically dispatched target. While CHA is sound and scalable, it is less precise than other alternatives. It is, however, a good trade-off and the default CG algorithm for most static analyzers. The next step towards greater precision is *Rapid Type Analysis* (RTA), as introduced by Bacon et al. [151]. RTA utilizes a CHA CG and prunes its edges. Hence, this algorithm has a higher performance overhead compared to CHA. RTA tracks allocated types and removes unreachable edges from the CHA CG to make the CG more precise. It is worth noting that we do not propose a new algorithm in this paper. Instead, we reuse the existing algorithms described here and combine them with other techniques to make them practical for CI tools.

**CG accuracy** Several studies have investigated the soundness and precision of CGs. By definition, a CG is considered sound if it covers all edges that are possible at runtime and precise if it does not contain edges that cannot be observed at runtime. Sound analysis is crucial for many use cases, especially security analysis, to avoid missing any positive cases. However, a low precision may result in many false positive alarms, which can hurt the usefulness of such an analysis. Increasing precision while maintaining soundness is a challenging and computationally expensive task.

A *dynamic* analysis can capture a perfect CG for a program execution by instrumenting the program and storing all observed invocations. However, this result is only valid for this particular program execution. In contrast, *statically* generating a perfect CG is an undecidable problem [152–154]. Furthermore, generating a more precise graph often requires a slower CG construction process. Users can choose to trade off precision for soundness and speed. However, some language features, such as reflection, are difficult to approximate in a static analysis, and generating a sound CG can be challenging in these cases. Livshits et al. [133] proposed the term *soundy* to describe an analysis that balances soundness, precision, and scalability. Additionally, Sui et al. [155] investigated the soundness of state-of-the-art CG constructions by comparing static and dynamic CGs. They used the term *recall* instead of soundness because, unlike soundness, it is not a binary term. Reif et al. [156] proposed a benchmark and compared the soundness of existing CG generators in their paper. Their results show that OPAL [142, 143] is faster than WALA [144], SOOT [157], and DOOP [158], but has lower coverage. They did not study other frameworks such as SLAM [159] and Chord [160]. In this paper, we do not compare our approach to these studies. However, we are inspired by the concepts of precision and recall introduced in these studies and adopt them to compare CGs generated by different approaches.

**Improving existing analyses** There is a category of studies that instead of proposing new algorithms focuses on improving the existing ones from different aspects by utilizing various techniques.

For example, some studies have investigated CG generation of applications versus libraries [161, 162]. Ali et al. [163] extensively investigated the effects of the absent parts of a program during analysis. Reif et al. [164] explain how different entry point calculation can affect the results of library CG construction and suggest different configurations for *within-application* and *library* CGs.

Other studies investigate incremental static analysis. Souter et al. [165] created the incremental *Cartesian Product Algorithm* (CPA). Tip et al. [146] improved the scalability of CG generation for large programs. Alexandru et al. [166] incrementally analyzed the historical representation of multi-revisioned software artifacts source code. Their proposed approach removes redundant computation for similar parts of different versions of an artifact. The approach only updates the changed parts of new versions in the CG.

Toman et al. [134] discuss the challenges of static analysis and provide possible solutions to deal with them together with some example articles. Their paper suggests solving scalability through the pre-computation of library summary, which is very relevant to the idea of our paper. However, they do not provide any implemented solution and only discuss ideas that might solve concrete analysis problems. Unfortunately, CG generation goes beyond a simple merge of summaries and is not trivial. Arzt et al. [167] utilize a similar idea for the data flow analysis of Android applications.

Nielsen et al. [135] proposed an approach for combining precomputed modules to compute a full CG. This study is highly relevant to our paper and emphasizes the importance of modular CG construction. However, the focus is on JavaScript, which has vastly different challenges than Java. Multiple studies have investigated the summarization techniques in different domains such as data flow analysis and heap analysis [168–170]. Gopan et al. [171] propose an approach for generating summary information for a library function by analyzing its binary. In this study, the authors do not focus on any particular use case such as CG generation.

Rountev et al. [172, 173] summarize the libraries and use them to do Interprocedural Dataflow Analysis on a main program source code. Dillig et al. present a heap analysis [174] technique based on the summarization of functions. Kulkarni et al. [175] investigated the library summarization idea for CG generation and points-to analysis, which is also closely related to our work. However, that paper has several shortcomings that we address in our paper, including not comparing to any state-of-the-art CG generator, not studying memory consumption, and only using a sample of ten programs. The most related work to this paper has been presented by Schubert et al. [132]. The authors present ModAlyzer, which uses summarization to generate CGs for C and C++ programs. The paper shows that the approach does not have to sacrifice soundness while substantially improving the speed of CG generation. However, the evaluation focuses on correctness and closely inspects results for a limited selection of programs. One case is a large project, for which a runtime of 25 minutes is reported. While this improvement of 90% over the 4-hour baseline is impressive, it is still impractical for our intended use cases. In addition to this, qualitative experiments show for several smaller programs that the approach does not compromise correctness when compared to a whole-program analysis. In our work, we decided to go a different route. First, we employ a much simpler *Class-Hierarchy Analysis* that does not require tracking extensive context information. Second, we decided to trade off soundness for execution speed to achieve scalability. Finally, we believe it is important to understand the applicability of static analysis to *average* programs, so we compile a larger set of real-world *versioned packages* and their dependencies to evaluate our work.

While soundness is an important property of static analysis, relaxing this requirement to achieve faster program analyses has been advocated by thought leaders in the field [133]. Many recent works propose tools that are not sound but useful in practice [155], like the rising area of machine learning that uses ML for increased usability of static analysis results [141]. While these are fundamentally unsound solutions, they are accepted as promising directions to make static analyses more practical. The most recent example of this idea has been proposed by Utture et al. [136]. The idea to prune CGs to make analyses more scalable is highly unsound, but the authors study the effect of the pruning on two example analyses, type casts, and null pointer detection, and show that the observed effect is small. Not all cases can be detected anymore, but the positive effect on performance and actionability that can be achieved through higher precision outweighs the loss in recall.

We are inspired by the studies that we introduced in this category to improve the existing frameworks. To the best of our knowledge, there are no existing studies that focus on the practicality that can be achieved by the summarization of CG generation in real-world scenarios.

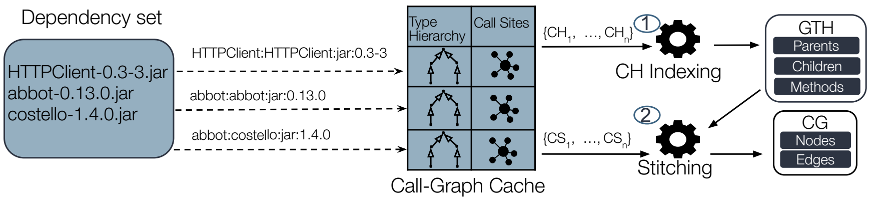


Figure 3.1: Overview of CG construction for a project

## 3.2 Approach

In this paper, we present a novel approach for scalable CG generation of Java programs that enables fast analyses, even in resource-limited environments. We call our approach Frankenstein because it is based on the idea of computing partial results that are stitched together to generate the final result. An overview of the general steps of this approach is shown in Figure 3.1. In the first step, all dependencies need to be resolved to generate the complete dependency set. For each of these dependencies, a partial analysis result is requested from a caching component, or created if it does not yet exist in the cache. This partial result consists of 1) the PCG of the isolated dependency, which includes all static Call Sites (CSs) that we can find in the bytecode, and 2) all types, their parent type, and their methods that are being declared in this particular dependency. The subsequent step, the *stitching*, is the core contribution of our approach. We first build a *global type hierarchy* (GTH) and merge all the individual type information of the partial results. We then perform a basic *class hierarchy analysis* (CHA) and expand the invocation targets found in the bytecode. This is achieved by adding links to all overriding methods within the matching subtypes that override the original targets.

While the underlying idea is straightforward, CG construction remains a complex task. In the remainder of this chapter, we will elaborate on the individual steps and illustrate the particular challenges and our strategies to overcome them.

### 3.2.1 Resolving Dependencies

To generate a CG for an application, it is essential to determine the concrete dependency set for that application through a process known as dependency resolution. In this regard, the widely used build automation tool Maven includes a built-in dependency resolver, simplifying the task for developers. Our proposed tool, Frankenstein, relies on the Maven dependency resolver to accurately compute the dependency set for an application.

It is worth noting that when defining dependencies in Maven, programs can specify version ranges [176]. However, this means that dependency resolution may not always be deterministic. A new release of any dependency, either direct or transitive, could potentially alter the resolution result from the perspective of a particular package.

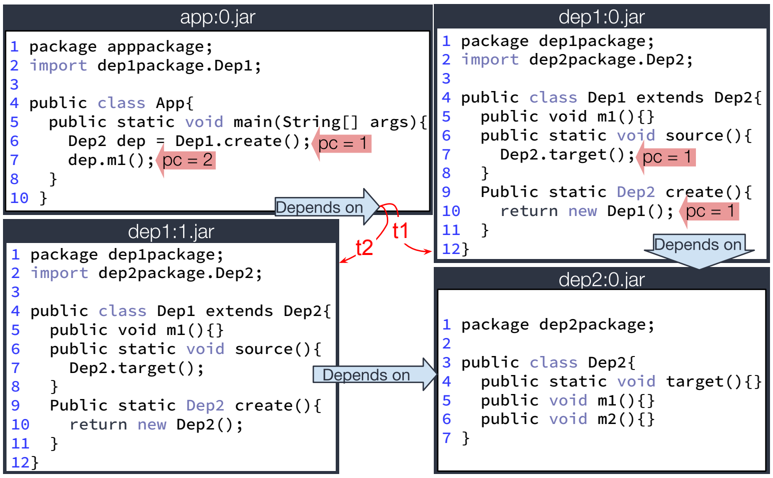
Although the set of available packages may remain stable, the resolution results can vary depending on different contexts. Consider a scenario where a client application $P$ has direct dependencies on two *versioned packages*, namely $A$ and $B$, both of which transitively rely on a library $L$. However, each of these *versioned packages* requires a different version of $L$, leading to a resolution conflict in Maven. To resolve this, a breadth-first search algorithm is used to select the version of *L* declared "closest" in the dependency tree of *P*. This is because having two versions of the same package in a dependency set simultaneously is not possible. Therefore, the closest version strategy prioritizes the most relied-upon version. Thus, depending on whether *P* depends on *A* or not, the resolved dependency set might differ when viewed from the perspective of *B*.

Overall, there are two challenges that make it unfeasible to simply generate a CG for an application and its dependencies. Firstly, as previously mentioned, the utilization of a specific dependency version is contextual. Secondly, the merging of partial results is not a straightforward task and demands a significant level of interaction between the partial results, which may even change based on the chosen dependency versions.

In Figure 3.2, we present an example dependency set and the interactions among its *versioned packages* to illustrate the subsequent steps of our approach. The dependency set comprises three *versioned packages*, and the dependency relations are depicted in the figure. For instance, versions `0` and `1` of `dep1` depend on `dep2:0`. We also demonstrate a time-sensitive dependency resolution in this example, where `app:0` specifies a version range dependency on `com.example:dep1:[0, 1]`. This results in a time-sensitive dependency resolution for `app:0`. For example, if the resolution is performed at time `t1` when the latest released version of `dep1` is `0`, then `dep1:0` would be included in the dependency set. Conversely, if the resolution is done at time `t2` after version `1` of `dep1` has also been released, then the result would include `1`. This time-sensitive dependency resolution highlights the contextuality, the first challenge that we mentioned earlier.

The second challenge occurs at a more granular level, where `Dep1` invokes the `target()` method from `Dep2`. In a more complex scenario, `Dep1` extends `Dep2` and overrides the `m1()` method. In the class App, the object `dep` is used to call `m1()`. Depending on the specific type of variable `dep`, any overridden `m1()` from sub-classes of `Dep2` could be invoked. The exact type of `dep` depends on the time of dependency resolution. For example, if the dependency resolution occurs in `t1`, we use `dep1:0`, and the implemented `create()` method in this version returns a `Dep1` instance, resulting in a call to `Dep1.m1()` in the `App` class. However, in `t2`, when version `1` is used, an instance of `Dep2` is returned, and thus the callee of this call would be `Dep2.m1()`. In this example, the type `Dep2` is used to declare the variable `dep`, which is referred to as the "receiver type". Determining the exact type of the receiver object is a challenging problem that different algorithms attempt to narrow down to varying extents. The CHA algorithm generates a sound CG by including all potential edges from all subtypes, such as `Dep2` and its subtypes, including `Dep1`.

To generate a whole-program CG from partial results the only information that is fixed for a *versioned package* can be stored and it becomes necessary to preserve enough context information about the *versioned package* to allow merging the partial results later in the process. For CG generation this minimal information is the PCG for the isolated dependency and all its declared types and methods. For simplicity, we refer to this data as a PCG. Whenever a whole-program CG should be generated, Frankenstein uses Maven for dependency resolution. The resolved dependency set is then used for the subsequent steps in our approach.



```
package apppackage;
import dep1package.Dep1;

public class App{
  public static void main(String[] args){
    Dep2 dep = Dep1.create();
    dep.m1();
  }
}
```

```
package dep1package;
import dep2package.Dep2;

public class Dep1 extends Dep2{
  public void m1(){}
  public static void source(){
    Dep2.target();
  }
  Public static Dep2 create(){
    return new Dep1();
  }
}
```

```
package dep1package;
import dep2package.Dep2;

public class Dep1 extends Dep2{
  public void m1(){}
  public static void source(){
    Dep2.target();
  }
  Public static Dep2 create(){
    return new Dep2();
  }
}
```

```

package dep2package;

public class Dep2{
  public static void target(){}
  public void m1(){}
  public void m2(){}
}
```

Figure 3.2: Example of a dependency set

### 3.2.2 Requesting or Creating PCGs

The next step in generating a whole-program CG is collecting the PCGs for all direct or transitive dependencies of the project in question. We have this dependency set from the previous step and we maintain a basic in-memory key-value store to manage access to the individual results and act as a cache. The key of each PCG is a Maven *coordinate* that is composed of a `groupId`, `artifactId`, and a `version`, which uniquely identifies a package within the whole Maven ecosystem. If the desired PCG is cached, it can be directly returned, which eliminates redundant processing. However, if the PCG is not yet available in our database, it has to be created first and then added to the cache for future use.

To create the PCGs, we first download the binary (i.e., `.jar` file) of the dependency in question from the Maven repository. We then use an existing static analyzer such as OPAL [142, 143] to build a CG for this isolated *versioned package* and transform it into a PCG. Different static analyzers may have different configurations and options that allow users to adjust the coverage, accuracy, etc. OPAL for example is highly configurable. We have used it to generate PCGs and, for that, we configure it for library mode, which uses all public methods as entry points in the analysis. OPAL considers that non-private classes, fields, and methods are accessible from outside, non-final classes are extendable and non-final methods are overridable. We have also enabled rewriting of `invokedynamic` instructions to make them easier to analyze. All other configuration options are left at their default values. The complete configuration can be found in our open-source repository to help others reproduce our results.

Using existing frameworks for generating the PCGs has substantial benefits. We do not need to work directly with bytecode and can rely on existing tools. This also enables users to use the framework of their choice as the approach is not dependent on any specific framework. Since existing tools allow for the extraction of the required information for PCG construction, users can use the static analyzer they already have in their CI without adding another dependency to their program.

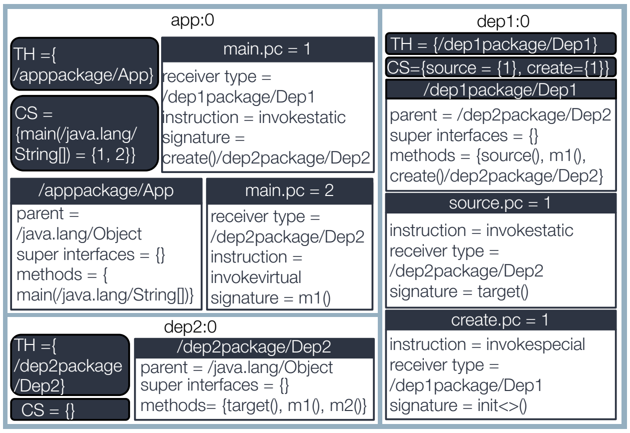

app:0
TH ={ /apppackage/App}
CS = {main(/java.lang/ String[]) = {1, 2}}
main.pc = 1
receiver type = /dep1package/Dep1
instruction = invokestatic
signature = create()/dep2package/Dep2
/apppackage/App
parent = /java.lang/Object
super interfaces = {}
methods = { main(/java.lang/String[])}
main.pc = 2
receiver type = /dep2package/Dep2
instruction = invokevirtual
signature = m1()
dep2:0
TH ={ /dep2package /Dep2}
CS = {}
/dep2package/Dep2
parent = /java.lang/Object
super interfaces = {}
methods= {target(), m1(), m2()}
dep1:0
TH = {/dep1package/Dep1}
CS={source = {1}, create={1}}
/dep1package/Dep1
parent = /dep2package/Dep2
super interfaces = {}
methods = {source(), m1(), create()/dep2package/Dep2}
source.pc = 1
instruction = invokestatic
receiver type = /dep2package/Dep2
signature = target()
create.pc = 1
instruction = invokespecial
receiver type = /dep1package/Dep1
signature = init<>()

Figure 3.3: PCGs of example *versioned packages*

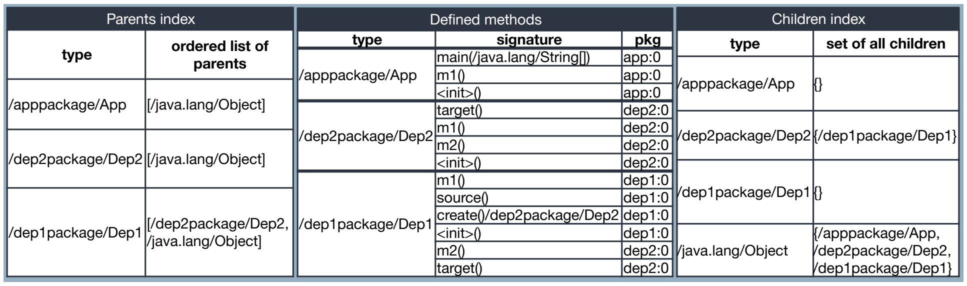
As touched upon before, PCGs contain two different types of information: a snapshot of the (incomplete) type hierarchy that is declared within the *versioned package*, including the declared methods, as well as the information about CS within the *versioned package*. Figure 3.3 shows simplified examples of PCGs for *versioned packages* in the example dependency set (resolved in t1).

**Type Hierarchy** PCGs store the Type Hierarchy (TH) that is defined in a *versioned package*. We use a naming convention similar to Java bytecode to identify types. For example, the type `App` in the *Java package* `apppackage` would be referred to as `/apppackage/App`. We do not store parameters of generic types because due to the type erasure [177] of Java, it is not possible to reason about them. We differentiate between types that have been declared inside the current *versioned package* (internal) and those in dependencies (external).

For every stored type, we preserve the list of declared methods. As Java supports virtual methods, we can use this later to infer all override relations. We store the method signatures and preserve their name, list of parameters (including types), and the return type. For example, the method signature of the well-known *equals* method `Object.equals(Object)` would be stored as `equals(/java.lang/Object)/java.lang/BooleanType`. For conciseness in Figure 3.3 we eliminated the `/java.lang/VoidType` from the methods with no return type.

Our data model allows for the storage of arbitrary meta-data as key-value pairs to cater to the needs of future use cases. We use this, for example, to mark abstract methods without implementation. These arbitrary metadata fields are also removed from the example in Figure 3.3 for the sake of brevity.

**Call Sites** The minimal information that is required from a CG for later reconstruction is a list of all *CS* that can be found in the PCG. A CS is an instruction in the bytecode that results in a method call. For each CS, we identify the surrounding (*source*) method and the



| Parents index | |
|---|---|
| type | ordered list of parents |
| /apppackage/App | [/java.lang/Object] |
| /dep2package/Dep2 | [/java.lang/Object] |
| /dep1package/Dep1 | [/dep2package/Dep2, /java.lang/Object] |

| Defined methods | | |
|---|---|---|
| type | signature | pkg |
| /apppackage/App | main(/java.lang/String[]) | app:0 |
| | m1() | app:0 |
| | <init>() | app:0 |
| /dep2package/Dep2 | target() | dep2:0 |
| | m1() | dep2:0 |
| | m2() | dep2:0 |
| | <init>() | dep2:0 |
| /dep1package/Dep1 | m1() | dep1:0 |
| | source() | dep1:0 |
| | create()/dep2package/Dep2 | dep1:0 |
| | <init>() | dep1:0 |
| | m2() | dep2:0 |
| | target() | dep2:0 |

| Children index | |
|---|---|
| type | set of all children |
| /apppackage/App | {} |
| /dep2package/Dep2 | {/dep1package/Dep1} |
| /dep1package/Dep1 | {} |
| /java.lang/Object | {/apppackage/App, /dep2package/Dep2, /dep1package/Dep1} |

Figure 3.4: Global Type Hierarchy of example *versioned packages*

*target* method that is being called, and we store this pair as one call relation. For each call relation, we store the bytecode instruction type, e.g., static invocation.

As Figure 3.3 shows, CSs of each source method are indexed by their program counter (*pc*). Since *pc* is unique for each invocation site, we use it as a key. This is helpful for tracing the results; however, it is not necessary for the approach, and one can use any unique key as an index for each CS. For instance, in the case of the *App* class, there are two CS in the main method. In the first one, `dep` is declared, and in the second one, it is used to call method `m1()`. The receiver type of the first CS is `/dep1package/Dep1` and it is used to call the `create()` method. In the second CS, `/dep2package/Dep2` is the receiver type since it is the type of the variable `dep`. The first call uses `static` invocation while the second one uses `virtual` invocation. For each CS, we also store the signature of the target method. In the previous example, the first call's signature is `create()/dep2package/Dep2` and the second one is `m1()`. As mentioned before, we use existing tools, more specifically already implemented CHA algorithms to build the PCGs. We only need static information, and existing frameworks and algorithms preserve this information and can be used in this phase. Only receiver type, invocation instruction, and target signature are required, which are available in the bytecode itself without any additional analysis. One could even read the bytecode and build the PCGs directly without using any third-party tool. However, this is beyond the scope of this study. We aim for a lightweight approach that can be easily used on top of existing tools.

Our database fits in regular machines' memory, so we can keep it instantiated for our evaluation. The data model is simple, making it easy to add an efficient, binary disk serialization. Generating the PCGs is expensive, and it is worthwhile to preserve partial results across different analysis executions for any actual use case. For instance, build server integration could preserve partial results from build to build for a substantial speed-up, similar to the caching of Maven downloads. Alternatively, a central server can host an in-memory storage of frequently used dependencies across many projects.

### 3.2.3 Inferring Global Type Hierarchy

After Frankenstein has acquired all PCGs for the different packages in the resolved dependency set, the next step is to merge the (incomplete) typing information of the individual *versioned packages* into a (complete) global type hierarchy. This creates the full picture of the type-system that is used when executing the whole program. We call this a *Global Type*

*Hierarchy (GTH).*

Assuming that the dependency set is complete, merging the individual type hierarchies can be reduced to joining the sets of internal types stored in the individual *THs*. As an example, consider the example dependency set in Figure 3.2 (resolved in `t1`). This set is $DepSet = \{\texttt{app:0}, \texttt{dep1:0}, \texttt{dep2:0}\}$ hence $GTH = TH_{app:0} \cup TH_{dep1:0} \cup TH_{dep2:0}$. All unresolved external types contained in the *THs* will appear as an internal type in one of the other *versioned packages*. For convenience and efficient traversal of type hierarchies, we transform this information into multiple index tables. These index tables are shown in Figure 3.4 for the example *versioned packages*. Every type is indexed using its full name since this name should be unique in the classpath of the program otherwise it cannot be compiled. We create three different index tables. One table stores the parents of each type based on the inheritance order. For example, the class `/dep1package/Dep1` directly extends `/dep2package/Dep2` therefore the first parent that appears in the parent list of `/dep1package/Dep1` is `/dep2package/Dep2`. This sequence continues for each type until we reach the `/java.lang/Object` class. After the list of all parent classes, we also append a set of all interfaces that a type or any of its parents implement. This is because Java always gives precedence to classes. Also, the order of super interfaces that are appended to the list of parents does not matter because there cannot be two interfaces with default implementations of the same signature in the parent list of a type. The parent index table is not directly used in the stitching phase. It is used to facilitate creating the children index and *defined methods index*. Another index that we create is a list of all children of a type. We identify all types that extend or implement a given type, including indirect relationships through inheritance. Indirect relationships occur when a type's ancestor extends/implements the given type, such as a grandparent. This set also does not keep any order. The final index in the *GTH* is the list of methods that each type defines or inherits. The signatures of these methods are then used as another index to find in which *versioned package* they are implemented in. I.e. if a method is not implemented in the current type itself we refer to its first parent that implements it. We use the ordered list of parents in the parent index to retrieve this information efficiently. For example, as shown in Figure 3.4 `/dep1package/Dep1` inherits method `m2()` from its dependency `dep2:0`. Note that since `/dep1package/Dep1` overrides `m1()` we point to `dep1:0` as the defining *versioned package* of this signature.

## 3.2.4 Stitching the Final CG

Once the PCGs are ready and the global type hierarchy has been established for the complete dependency set, Frankenstein can move to the most crucial part of the approach, the *stitching*. Similar to a compiler that resolves symbols in an abstract syntax tree, we need to connect the CSs that we found in the bytecode with all potential method implementations that could be reached by the corresponding invoke instruction.

The algorithm to achieve this is sketched in Figure 3.5. After creating the *GTH* (3.2.3), the algorithm processes all CSs in all PCGs. The *Java virtual machine* (JVM) supports five different invocation types that require specific handling (`invokestatic`, `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokedynamic`). Depending on the invocation type, Frankenstein selects the correct resolution strategy for the call when processing the CSs. While different invocation types are handled differently within JVM we treat them

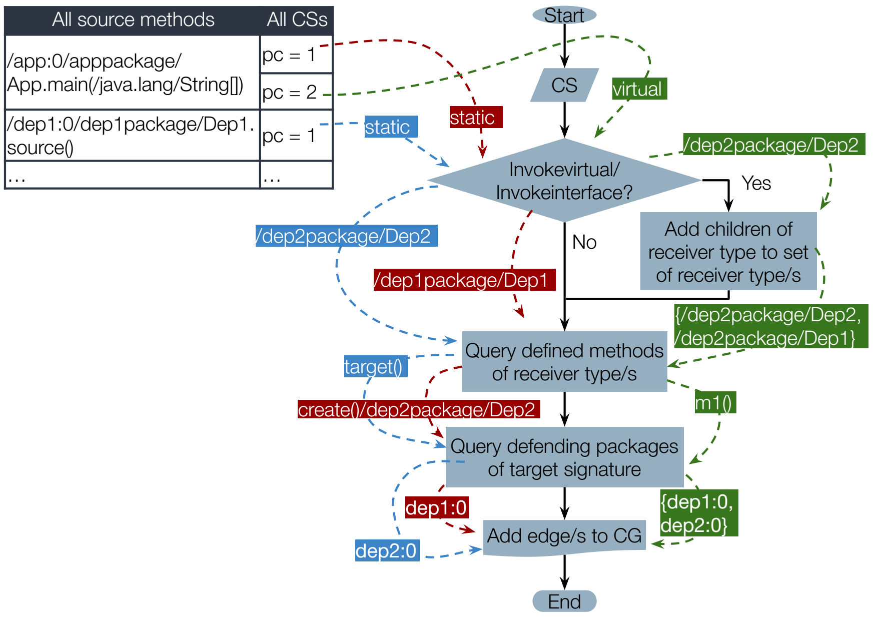

All source methods
All CSs
/app:0/apppackage/App.main(/java.lang/String[])
pc = 1
pc = 2
/dep1:0/dep1package/Dep1.source()
pc = 1
...
...
Start
CS
virtual
static
static
Invokevirtual/Invokeinterface?
/dep2package/Dep2
Yes
Add children of receiver type to set of receiver type/s
No
/dep2package/Dep2
/dep1package/Dep1
{/dep2package/Dep2, /dep2package/Dep1}
Query defined methods of receiver type/s
target()
create()/dep2package/Dep2
m1()
Query defending packages of target signature
dep1:0
{dep1:0, dep2:0}
dep2:0
Add edge/s to CG
End

Figure 3.5: Resolving edges of CSs

in two categories of *dynamic dispatch* (`invokevirtual`, `invokeinterface`) and *regular dispatch* (`invokestatic`, `invokespecial`) calls in our algorithm. In the following, we briefly explain these invocation types and how we handle them.

**Invokestatic** `invokestatic` is used for regular static method calls. This case can be directly resolved by adding an edge from the surrounding method to the static target method from the instruction. As long as the dependency set is a valid resolution, the target type will exist in the GTH and it will always have a matching method declaration signature either implemented in the type itself or one of its parents. A static call `T1.m()` has to have an implementation of `m` in `T1` or one of its parents. If this is not the case, the resolved dependency set is invalid and not executable.

**invokespecial** The `invokespecial` instruction is used for calling non-overridable methods, like private methods, super methods, and constructors. Similar to static invocation, the target type of the CS is in the GTH and can look up the method with a matching signature to draw an edge. For example, to resolve the constructor call `T2.<init>()`, we lookup `T2` and search for a parameterless constructor.

**invokevirtual** Instructions that require dynamic dispatch during runtime, `invokevirtual` and `invokeinterface`, are the most challenging to resolve. In Java, the corresponding CSs point to a base type or an interface, but during runtime, the receiving instance could have any possible subtype of this base type. For example, even if the bytecode contains an invocation of the `Object.hashCode()` method, the receiver type might be any type in the GTH that overrides the `hashCode` method. Similarly, while the bytecode might contain a reference to `T3.hashCode()`, `T3` might not override the method, but inherits it from its superclass, in which case the correct target of the invocation is the method declared in the superclass. Frankenstein supports both cases and searches for matching method signatures in all subtypes of the target and draws edges to those that can be called. In case there is none, it finds the first supertype that implements this signature.

**invokeinterface** This invocation is handled in the same way as `invokevirtual` in the JVM. Some optimizations are performed by JVM based on which instruction is used in the bytecode but this does not affect our approach. These optimizations are done on the virtual method table of JVM which we do not use in our approach. Moreover, for `invokeinterface`, it is possible to encounter target methods that are defined outside of the hierarchy of the target type. For illustration, consider a base type `B` that implements `m` and an interface `I` that defines `m` as well. A subtype `S` implements `I` and extends `B`. As `m` is inherited, it is not necessary to implement it. For a CG generator, this is a challenging case to handle, as the `invokeinterface` instruction points to `I`, while the correct target is `B.m`, despite `B` having no relation with `I`. Frankenstein can handle these cases.

**Invokedynamic** Recent versions of the JVM have introduced the `invokedynamic` instruction to support alternate programming languages for the JVM that might use more advanced invocation logic. Resolving such invocations is the most challenging part of CG generation tools. Existing frameworks usually have limited support for these invocations since they require very expensive operation due to their highly dynamic nature. Some frameworks such as OPAL can rewrite these invocations using other invocation instructions. We do not have special handling for these invocations in the *stitching* algorithm. However, if the framework that we use for PCG construction has the feature to rewrite them we utilize it to handle them automatically without special reasoning. As an illustrative example, consider a scenario in which the OPAL framework employs the `invokevirtual` instruction to transform a particular CS that was initially an `invokedynamic` CS. During the bytecode analysis for PCG creation, we store the modified version of the CS. Subsequently, in the *stitching* phase, we utilize the dynamic dispatch handling technique that was previously explained.

Figure 3.5 shows how we use CS information and GTH to resolve each CS of the example dependency set. Consider the second CS of `App.main` with `pc = 2`. This invocation uses virtual instruction; hence, we need to use the dynamic dispatch category of handling for it. For that, we first query the children index of the GTH to retrieve all children of `dep2package/Dep2`. The result of this query is `/dep1package/Dep1`. Thus we add this child type to the list of receivers. In the next step we query the defined methods within the `/dep1package/Dep1` and `/dep2package/Dep2` using the defined methods index of GTH. Having this information we can easily ask for the exact location of the target method. More specifically we query the result of the previous step for the method signature of `m1()`. This will identify the two places where `m1()` is defined and are thus potential targets of this call. One target is the implementation within `dep1:0` inside `Dep1` class and the other one is implemented in `dep2:0` in the `Dep2` class. The next CSs will be also processed similarly (see Figure 3.5). After all, PCGs have been processed, the resulting CG is ready and can be used for further static analyses.

## 3.3 Evaluation

The approach that we discussed in this paper tries to improve the speed of CG generation for consecutive software builds by removing redundant computations. In this section, we investigate how successful the proposed approach is in achieving its goal. We also investigate the effects of the approach on the soundness of CGs and whether it adds any overhead to the memory requirements of existing static analyzers. In a quantitative evaluation, we compare Frankenstein with the current state-of-the-art static analysis frameworks. The evaluation consists of four Research Questions (RQs):

- RQ1: How accurate are Frankenstein's CGs?
- RQ2: How Fast is Frankenstein?
- RQ3: Is Frankenstein generalizable?
- RQ4: How much memory does Frankenstein require?

### 3.3.1 Creating a Representative Dataset

Our evaluation strategy for library summarization drew inspiration from previous work by Kulkarni et al. [175]. They evaluated their approach using 10 programs. To improve upon this, we expand the scope of our evaluation by a factor of 5x, using 50 programs in total.

We randomly select 50 packages (i.e., `groupId` and `artifactId`) from the Maven repository and then randomly select one version for each package. We use Shrinkwrap [178] to resolve the complete dependency sets for the selected *versioned packages*. Shrinkwrap is a Java library that re-implements Maven's built-in dependency resolution. During the dependency resolution process, we face "missing artifact on Maven" errors in some cases. In some other cases, dependency resolution results in an empty dependency set, which either means that they were *POM projects* and do not contain code, or they do not include any dependencies. When we face such cases, we discard them and select another random *versioned package* until we have 50 fully resolved dependency sets. These dependency sets include a total of 1044 *versioned packages* (906 unique *versioned packages*). More characteristics of these *versioned packages* are shown in Figure 3.6. On average, the selected *versioned packages* have 20 dependencies and consist of 113 files (6051 including dependencies).

In the first two RQs of this paper, we compare Frankenstein and OPAL[2]. To mimic a representative environment, we decided to limit all of our experiments to 7GB of memory, which is the available memory size for `Linux` build jobs on GitHub.[3] To achieve fairness when comparing two approaches, we remove *versioned packages* for which one of the approaches failed due to OutOfMemoryException. All statistics that we report in

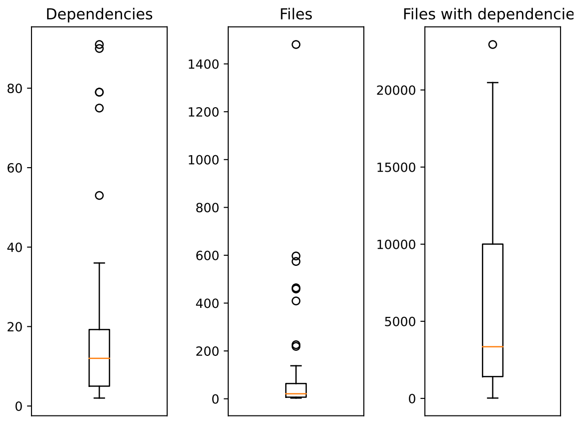


Figure 3.6: Selected *versioned packages*' Characteristics

the research questions are therefore generated for the dependency sets for which both approaches successfully generated CGs.

### 3.3.2 RQ$_1$: How accurate are Frankenstein's CGs?

The most important assessment that we need to do is to understand whether or not Frankenstein affects the quality of the generated CGs. We are interested in a comparison between the accuracy of CGs generated by Frankenstein and OPAL on its own. Hence, we answer the question of how Frankenstein affects the accuracy of the CGs in the form of two sub-questions:

- How does *stitching* affect the precision of the CGs?
- How does *stitching* affect the recall of the CGs?

**Methodology** To investigate the accuracy of the generated CGs by the proposed approach, we compare the generated CGs with OPAL. We take the CGs that are generated by OPAL as the ground truth and do not further investigate their correctness. Previous studies [156] have already compared OPAL with other well-known existing frameworks like Wala [144] or Soot [157].

Existing studies[140, 156] measure the soundness of CGs or compare different frameworks. However, they do not aim to compare the edges of real-world programs. Additionally, they do not follow our assumption that OPAL's CGs build the upper bound for our results. We cannot get more accurate than OPAL, since we use it as a base framework for generating the PCGs. Ideally, our approach should be as sound and precise as OPAL. Therefore, we propose an edge-by-edge comparison by calculating the precision and recall of our CGs compared to OPAL's results.

[2]In all of our experiments, we use OPAL version 4.0.0-SNAPSHOT which is included in our artifact page.
[3]https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners#supported-runners-and-hardware-resources, Accessed: 2022-01-15

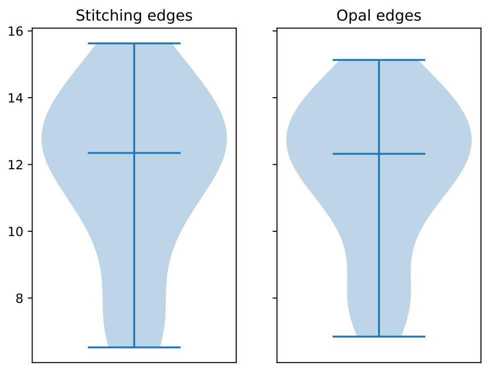


Figure 3.7: Edge comparison of Frankenstein and OPAL

To calculate the precision and recall for each dependency set's CG, we first group all edges of the CG by their source method for both OPAL and Frankenstein. All resulting sets can be identified by type and method name, and we can calculate the precision and recall of Frankenstein CGs compared to OPAL. For the calculation, we construct the intersection between the two results for each source. Assuming that $i$ is the intersection set, $f$ is the set of target methods that Frankenstein found for a particular source, and $o$ is the set of target methods identified by OPAL, we calculate precision and recall as follows:

$$precision = \frac{|i|}{|f|} \qquad recall = \frac{|i|}{|o|}$$

For example, let's assume that for the source method `A.src()`, OPAL finds three targets: `B.t1()`, `C.t2()`, and `D.t3()`, while Frankenstein only finds two targets: `B.t1()` and `X.x()`. The intersection set for `A.src()` only contains `B.t1()`; hence, the precision is $\frac{1}{2}$ and the recall is $\frac{1}{3}$.

**Results** To investigate the soundness and precision of Frankenstein, we first compare the number of edges generated by OPAL and *stitching*. For 50 dependency sets, OPAL found a total of 25M edges, while Frankenstein found 36M. Figure 3.7 shows the violin plot of the number of edges. The numbers in this plot are logarithmic. However, these numbers only provide an initial comparison, and it is necessary to further investigate the overlap between the generated edges. We consider two edges identical if the source methods and target methods are identical. We then calculate precision and recall for the targets of each source method as discussed in 3.3.2. A perfect precision of 1.0 means that all stitched targets found for a source are also present in OPAL's CG, while a perfect recall of 1.0 means that all targets in OPAL's CG have also been identified by Frankenstein.

The precision and recall of all source methods in 50 dependency sets are shown in Table 3.1. In terms of precision, we achieve an average precision of 92.7% using Frankenstein, indicating that most stitched edges can also be found in OPAL's CGs. In terms of recall,

Table 3.1: Precision and recall of Frankenstein

| | Mean | Std. Deviation | Median |
|---|---|---|---|
| **Precision** | 92.7% | 24.2% | 100% |
| **Recall** | 94.7% | 21.4% | 100% |

there are some cases with a recall of less than 100%, resulting in an average of recall 94.7%. However, the median of precision and recall is still 100%.

These decreases in the recall can be explained by missing CSs in PCGs, which are caused by OPAL not supporting all corner cases in partial analysis or having low coverage. For language features that are not very common, OPAL may not produce complete information in the partial analysis.

To further investigate these problematic corner cases of partial analysis, we manually compared the CGs generated by OPAL and Frankenstein for a benchmark that has been proposed in previous work [156]. This benchmark consists of a set of annotated test cases that cover numerous possible ways of calling a function in Java. The annotations in these test cases are written by several experts in the field to indicate the expected edges that a sound CG must include. After comparing the two resulting CGs edge by edge, we categorized the CGs of the test cases into four categories:

- $\square_{OF}$: both OPAL and Frankenstein can generate a *sound* CG for the test case.
- $\square_{O}$: only OPAL can generate a *sound* CG for the test case.
- $\square$: neither OPAL nor Frankenstein can generate a *sound* CG for the test case.
- ⊠: none of the two approaches was able to generate a CG for the test case.

Table 3.2 shows the results of our manual analysis for the various test cases. The abbreviations refer to different language features, and we refer to the original work for details [156]. Some language features have multiple test cases to test different ways of calling with that particular Java language feature. In the table, we reported the number of test cases that we analyzed per language feature in the first column. For example, out of the five test cases that cover the language feature *JVM Calls* (JVMC), four are $\square$, and one is $\square_{OF}$.

In the table, the overall number of test cases in which $\square_{OF}$ approaches generated sound CGs is 24. We also have 52 test cases that ⊠ of the approaches could generate CGs for. This is because we use the library mode of OPAL without providing specific entry points for it due to the nature of our analysis, hence some of these small test cases are not suitable for library analysis because of functions not being reachable from the outside. 15 out of 109 test cases are $\square$ meaning that none of the approaches could generate a sound CG. Since our PCG generation is dependent on OPAL, it is inevitable not to be sound in cases where OPAL itself cannot be sound. The most important cases for our study, however, are the $\square_{O}$ test cases because they can reveal the reason for the reported 94.7% recall. We further investigate the root cause of these cases to understand whether it is a shortcoming of the approach or not. We have 18 test cases that only OPAL generated sound CGs for. These test cases are distributed among five different language features. The common reason for all missing edges in these test cases is that in the PCG generation phase OPAL does not provide us with all possible CS. This also means that for all CSs that we could extract in

Table 3.2: The results of manual analysis of Stitched CGs.

| **LF** | **# Cases** | $\square_{OF}$ | $\square$ | $\boxtimes$ | $\square_{O}$ |
|---|---|---|---|---|---|
| CFNE | 4 | 2 | 0 | 0 | 2 |
| CL | 1 | 0 | 1 | 0 | 0 |
| CSR | 4 | 0 | 1 | 3 | 0 |
| DP | 1 | 0 | 1 | 0 | 0 |
| ExtSer | 3 | 3 | 0 | 0 | 0 |
| J8DIM | 6 | 0 | 0 | 6 | 0 |
| J8SIM | 1 | 0 | 0 | 1 | 0 |
| JVMC | 5 | 1 | 4 | 0 | 0 |
| Lambda | 4 | 4 | 0 | 0 | 0 |
| LIB | 5 | 1 | 1 | 2 | 1 |
| LRR | 3 | 0 | 0 | 3 | 0 |
| MR | 7 | 7 | 0 | 0 | 0 |
| NVC | 5 | 1 | 0 | 3 | 1 |
| Ser | 9 | 2 | 4 | 3 | 0 |
| Serlam | 2 | 0 | 2 | 0 | 0 |
| SI | 8 | 3 | 0 | 0 | 5 |
| SPM | 7 | 0 | 0 | 7 | 0 |
| TC | 6 | 0 | 0 | 6 | 0 |
| TMR | 8 | 0 | 0 | 8 | 0 |
| TR | 9 | 0 | 1 | 6 | 2 |
| Unsafe | 7 | 0 | 0 | 0 | 7 |
| VC | 4 | 0 | 0 | 4 | 0 |
| Sum | 109 | 24 | 15 | 52 | 18 |

the PCG creation phase, we generated the correct edges. Listing 3.1 shows an example of a reflective call for which OPAL generates a sound CG. This occurs when the whole program is present during the analysis i.e. the Target class is included in the analysis. However, in this example, during partial analysis, the Target class is not present, and OPAL does not provide us with enough information about the *target.target()* CS in the PCG. Since we use OPAL for our partial analysis, this affects the soundness of our approach. Similar to other successful cases, if OPAL had provided the CS information in the PCG, we have could *stitched* it to the Target class. That is, we could not find any cases where the proposed approach inherently causes any unsoundness in these test cases. Nevertheless, the missing CS problem can be fixed with engineering efforts, either on OPAL or by additional handling in the PCG creation phase. We have documented these cases and discussed them with the OPAL developers for future improvements of both tools.

Example 3.1: Reflective method call

```java
public static void main(String[] args) throws Exception {
    Demo demo = new Demo();
    demo.field = new CallTarget();

    Field field = Demo.class.getDeclaredField("field");
    Target target = (Target) field.get(demo);
    target.target();
}
```

### 3.3.3 RQ$_2$: How Fast is Frankenstein?

The main benefit of summarization of CG generation is the speedup it provides due to less computation. We need to evaluate whether removing redundant library generation from the process affects the speed of CG generation. It is also important to investigate the extent of any improvements, if any.

**Methodology** We design and implement a tool that creates the CG Cache for randomly selected dependency sets. Therefore, we will be able to test the performance of our approach and compare it with the baseline (OPAL).

For this experiment, we need to simulate consecutive builds in software programs. Maven hosts a large number of packages with different versions. We select random *versioned packages* from Maven to build their CGs. Hence, we need to resolve the dependency sets of selected *versioned packages*. We use a dependency resolver library called *Shrinkwrap* because it is open source and works well with Maven *coordinates*. In some cases, this tool may not be perfect and could result in missing dependencies in the resolved sets. However, we still use it to ease the dependency resolution process, as it is not the main focus of our paper. We ensure that this does not cause any unfairness by always using the same set of dependencies for our approach and baselines.

After selecting the dependency sets, we perform the following steps on each dependency set D:

1. Generate full CGs using Frankenstein:
    - For each *versioned package* in D, we generate a CG using OPAL and parse it into a PCG.

Table 3.3: Time of CG generation different phases (in seconds)

| | Mean | Std. Deviation | Median |
|---|---|---|---|
| **Frankenstein** *initial* | 18.1 | 21.2 | 10.2 |
| **Frankenstein** *cached* | 5.0 | 7.4 | 2.2 |
| **OPAL** | 6.0 | 6.6 | 3.2 |

- We create the CG Cache using the PCGs.
- For each *versioned package* in D, we fetch PCGs from the CG Cache and then stitch them.

2. We generate a full CG for D using OPAL.

3. Finally, we measure time information and CG statistics of the previous steps to find out how much speed Frankenstein can add in the consecutive generation of the CGs.

**Results** As shown in Table 3.3, for these 50 dependency sets on average OPAL takes 6s to generate a CG for each dependency set, whereas Frankenstein takes 5s. The average time for the first round of Frankenstein generation is 18.1s (as shown in Frankenstein *initial*). This round includes PCG generation and caching for the entire set of dependencies, as well as the *stitching* process. However, subsequent rounds benefit from a significant speed improvement. Note that Frankenstein *cached* represents subsequent rounds of generation after caching and includes the time for generating the PCG of the current version of the application but not the rest of the dependency set. This is because the application's logic usually changes between CI builds, requiring a rebuild of its PCG. All three rows of this table show large standard deviations. This is due to the variety of CG sizes. The fact that multiple dependency sets in the selected sets are considerably larger than others causes longer generation times and hence bigger standard deviations. Generating a PCG for a dependency *versioned package* is a one-time process that needs to be done once and only once. After that, the cached dependencies can be used in different builds. For the first time, Frankenstein may take longer than OPAL due to the combination of caching PCGs and *stitching*. However, in consecutive software builds, Frankenstein saves a lot of time by pre-computing the common parts. Over time, the enhanced speed results in significant time savings, which accumulate with repeated use and translate into considerable overall efficiency gains.

Figure 3.8 shows the violin plot of the generation time for OPAL, Frankenstein initial generation, and Frankenstein with cached dependencies for 50 dependency sets. The numbers are logarithmic for the sake of better presentation, and median lines are visible for all plots. Frankenstein with a median of 2.2s outperforms OPAL with 3.2s. It is worth mentioning that Frankenstein's first round of generation with a median of 20.2s is slower than OPAL. This slowdown is because we use OPAL for PCG generation and on top of that we extract the necessary information for PCGs. However, because it is a one-time process the generation time will improve from the second CG generation onwards.

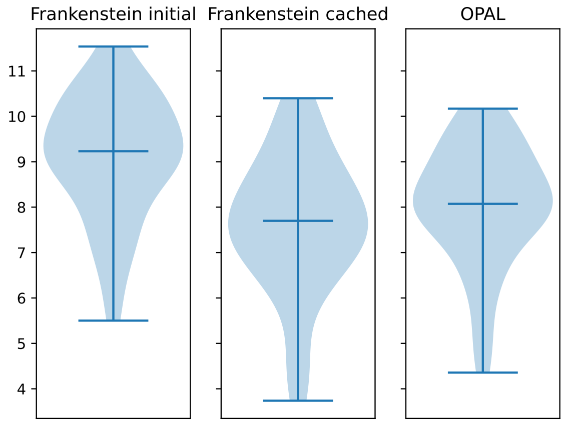


Figure 3.8: Time comparison of Frankenstein and OPAL

Table 3.4: Precision and recall of Frankenstein when generating with WALA

| | **Mean** | **Std. Deviation** | **Median** |
|---|---|---|---|
| **Precision** | 98.9% | 0.07% | 100% |
| **Recall** | 97.5% | 11.0% | 100% |

### 3.3.4 RQ$_3$: Is Frankenstein generalizable?

In the previous sections, we presented the benefits of Frankenstein in terms of the speed of CG generation. We also investigated its effects on the accuracy of the CGs. However, since the proposed Frankenstein aims to make CG generation practical for software builds, it is vital to consider the limitations that build servers have to validate the practicality of Frankenstein. On the one hand, Frankenstein requires memory to cache partial results. On the other hand, build servers often have limited memory available. Therefore, we need to investigate how much memory Frankenstein requires.

**Methodology** To investigate whether Frankenstein is generalizable, we slightly adapt our setup and use WALA as the static analysis backend for generating CGs. Apart from this adjustment, we use the same methodology as in RQ1 and RQ2 to compare the speed and accuracy of the generated CGs. Overall, we compare the CGs generated by WALA and Frankenstein for 50 randomly selected *versioned packages*. We use WALA both for the whole-program analysis and the PCG generation, configured to run the CHA algorithm, and using all methods of all classes as entry points. To measure the accuracy of the *stitched* CGs, we use the *precision* and *recall* metrics defined in Section 3.3.2.

**Results** Similar to the evaluation on OPAL, the goal of this experiment is to understand the *speed* and *accuracy* of Frankenstein compared to a plain WALA approach. Table 3.4

shows the results for precision and recall of the generated CGs. The results confirm the same effects for WALA CGs that we have seen before in Table 3.1. On average, Frankenstein achieves 99% precision, meaning that virtually all identified edges also exist in the WALA CGs. Similar to the OPAL experiment, Frankenstein also misses some edges in comparison to WALA, resulting in a recall of 97.5%. When we investigated these cases manually, we found some corner cases that require special handling during CG generation. For example, WALA contains hard-coded information about Java-related classes (such as Object) and functions (such as Object.hashcode()), resulting in additional edges that Frankenstein does not have. Since we did not provide any additional information to Frankenstein about Java-related classes, it is not surprising that it cannot reproduce these edges. These limitations can be addressed in the future by adding special handling for said Java classes or by providing such classes in the dependency set. A second explanation that we found for missing edges is that existing static analyzers are not perfect and might create incorrect edges. In our manual analysis, we also found a case in which classes were copied from a dependency and were then modified locally. When ignoring the origin of a class file, the name of the class is the same locally and in the dependency. As a result, WALA cannot distinguish the classes and adds additional nodes and edges to the dependency class. Since the including .jar file is part of Frankenstein's fully-qualified names, we can avoid these edges. Across the manually inspected cases, we could not find any differences indicating any conceptual limitations, and all deviations could be addressed through more engineering effort. Most importantly, all limitations come at our own cost, and we believe that the results present a fair comparison. Overall, Frankenstein achieves 98.2% F1 compared to WALA.

Regarding our performance investigation of Frankenstein, we observed similar improvements for WALA (Table 3.5) and OPAL (Table 3.3). Both tables show substantial speed improvement once the PCGs are cached. On average, WALA generates a CG in 16.6s, and Frankenstein generates it in 10.2s. This means that caching the PCGs speeds up the process by 38% on average. The caching mechanism applies only to dependencies, not to the application itself. This means that the PCG generation time for the application *versioned package* is included in the Frankenstein $_{cached}$. It is important to realize that the first round of generation is slower due to additional analysis to create the PCGs (119.8 seconds). However, this slowdown pays off in the next rounds of CG generation by saving a lot of time and resources. Compared to the OPAL results, the numbers are generally a bit higher, which can be mostly explained by WALA generating a larger CG that has more nodes and edges. However, the relative comparison and performance increments are comparable.

Overall, our results show that Frankenstein is a generic solution. Users can choose a static analysis framework that fulfills their needs, such as OPAL or WALA. Using Frankenstein then allows for substantial speedup of CG generation through pre-computing and caching PCGs with minimal side effects on accuracy.

### 3.3.5 RQ$_4$: How much memory does Frankenstein require?

In the previous sections, we presented the benefits of Frankenstein in terms of the speed of CG generation. We also investigated its effects on the accuracy of the CGs. However, since the proposed Frankenstein aims to make CG generation practical for software builds, it is vital to consider the limitations that build servers have to validate the practicality of

Table 3.5: Time of CG generation different phases (in seconds) with WALA

| | Mean | Std. Deviation | Median |
|---|---|---|---|
| **Frankenstein** $_{initial}$ | 119.8 | 147.8 | 65.0 |
| **Frankenstein** $_{cached}$ | 10.2 | 3.3 | 9.8 |
| **WALA** | 16.6 | 11.4 | 12.7 |

Frankenstein.

**Methodology** To answer the third research question we investigated memory usage from two perspectives.

First, we imposed memory constraints on all of our experiments. In the CG generation steps detailed in Section 3.3.3 (specifically the first and second steps), we restricted the memory available to the JVM. This meant that both our baselines and the CGs generated by Frankenstein for the chosen *versioned packages* (as described in Section 3.3.1) operated within a limited-resource environment, simulating a build server. Following the generation steps, we compared their success rates and the frequency of OutOfMemoryException. This allowed us to examine the impact of memory limitations in real-world scenarios on our proposed method.

Second, we employed the open JDK profiling tool JOL [179] to measure the space occupied by Frankenstein within the JVM heap. Specifically, we determined the sizes of the PCGs and index tables (refer to Section 3.2.3) utilized in the *stitching* process.

**Results** After running the experiment, we realized that out of 50 dependency sets, both OPAL-based Frankenstein and OPAL encountered memory errors for 3 *versioned packages*. The fact that they had the same number of memory exceptions indicates that Frankenstein does not introduce any additional memory requirements or add memory overhead to the base framework OPAL. When generating with WALA, however, WALA-based Frankenstein faced fewer memory exceptions compared to WALA itself. WALA failed in 8 instances, while Frankenstein failed in 4 cases due to OutOfMemoryException. This lower memory demand may be attributed to Frankenstein holding just enough information about each *versioned package* to perform a basic CHA algorithm, whereas WALA, as a general static analysis framework, needs to capture a more comprehensive view of the entire dependency set's bytecode.

To assess the memory requirements of Frankenstein for generating a complete program CG, we examined all data necessary for the *stitching* process. For each dependency set, Frankenstein initially caches all dependency PCGs, followed by the creation of *GTH* each time *stitching* is executed. Table 3.6 displays the memory usage in megabytes for these Data Structures (DSs) in both OPAL-based and WALA-based implementations of Frankenstein. The *PCGs* row is determined by summing the sizes of all PCGs for each dependency set. We then calculated the mean, median, and standard deviations for the selected 50 dependency sets, as shown in the table. Likewise, the *GTH* row is based on the 50 *stitching* operations performed on the selected dependency sets, with the mean, median, and standard deviations reported. In summary, on average, OPAL-based Frankenstein necessitates a total of 432 Mb

Table 3.6: Memory required by Frankenstein (in Mb) to *stitch* a whole program

| | **DS** | **Mean** | **Std. Deviation** | **Median** |
|---|---|---|---|---|
| OPAL-based | **PCGs** | 285 | 299 | 159 |
| | **GTH** | 152 | 146 | 93 |
| WALA-based | **PCGs** | 272 | 301 | 173 |
| | **GTH** | 116 | 102 | 88 |

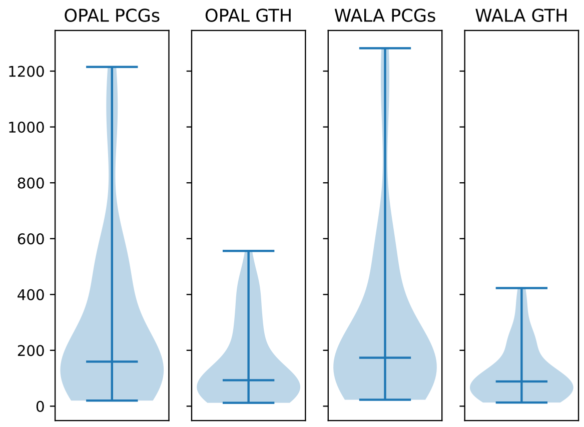


Figure 3.9: Distribution of Data Structure sizes (in Mb) required by Frankenstein

of memory, while WALA-based Frankenstein demands 388 Mb of memory to generate a whole-program CG. Figure 3.9 further illustrates the distribution of this data.

An interesting observation from this table is that both WALA-based and OPAL-based figures are within the same order of magnitude. This outcome is somewhat expected since the general approach is similar. However, differences may arise from two factors. One reason is the variation in successfully generated dependency sets. As previously mentioned, some dependency sets did not have a CG generated due to memory exceptions. This implies that the sets of projects successfully generated for WALA-based and OPAL-based Frankenstein are not identical. Another reason for the discrepancies is that the outputs of different frameworks are not identical, even for the same operations. For instance, as we discussed in previous sections the CGs generated by WALA are generally larger than those generated by OPAL due to the different coverages of these tools. This may result from numerous differences in the implementations of the various frameworks.

## 3.4 Discussion

As manual inspection has revealed that the lack of recall is mainly due to missing CS information during PCG construction or the internal handling of special cases within WALA or OPAL such as built-in handling of Java-related classes. Nevertheless, we are confident that this limitation can be fixed through engineering work. We strongly believe that the proposed approach has the potential to be adopted by engineers, software companies, and researchers to implement fast and lightweight CG-based analyses that are practical for software development pipelines such as CI tools. Moreover, our proof of concept Frankenstein shows that summarization works in practice. Thus we highly recommend that developers of existing static analysis frameworks adopt the summarization technique that we propose. These tools can benefit from the speed up while reusing their handling of special cases. This potentially can remove or mitigate the accuracy penalties that we reported in this paper.

Our results confirm previous findings in the field. First, as pointed out by Toman et al. [134], library pre-computation can effectively solve the scalability problems of the analysis. Our study validates this idea, particularly for CG generation. However, applying this technique to different static analysis tasks can have similar positive effects as examined by Arzt et al. [167] for data flow analysis.

We believe that the concept of summarization will strongly impact the design of future static analysis tools. Although configurability and modularity are essential requirements for static analyzers it is important to consider them not only for program design but also for data flow design. Currently, these tools can be highly configurable and different modules can be combined in a pipeline based on specific requirements. For a given configuration, this flexible pipeline of modules is created and executed on top of a single input data, producing a single output. We propose that by modularizing the intermediate data, analysis modules can save the current state at specific points and restart the task from the point of interruption. This approach would make the summarization of different static analysis tasks straightforward. We believe that this will make static analysis more practical for daily usage such as in CI tools.

In this paper, we acknowledge the tradeoff between memory, speed, and cache utilization when using Frankenstein. The caching mechanism significantly improves speed by reusing precomputed PCGs but requires memory for cached PCGs. Our experimental results illustrate these tradeoffs, providing a practical evaluation of Frankenstein's real-world performance improvements and its memory usage. This information will help developers make informed decisions based on the benefits and costs of the tool.

## 3.5 Threats to Validity

**Internal validity** WALA and specially OPAL offer a plethora of configuration options, all of which may affect our results. To mitigate this we attempted to utilize the maximum coverage that these frameworks provide, such as using the most comprehensive entry point strategies or the assumption of library openness [164]. Additionally, we provide public access to our code including an aggregated configuration file on our artifact page to ensure a fixed and transparent configuration throughout the study.

Our approach supports all Java invocations except `invokedynamic`. We use OPAL's `invokedynamic` rewrite feature, which rewrites this type of method invocation in the form of other invocations. However, some frameworks may not support the `invokedyna-mic` rewrite feature, requiring the implementation of heuristics to support `invokedyna-mic` when adopting our approach. Nonetheless, this and other corner cases that we reported earlier in this paper result in minor accuracy penalties that can be mitigated if developers of existing frameworks adopt the Frankenstein solution within their frameworks. They can reuse the same treatment that they already have in their toolbox for these cases while benefiting from the advantages of Frankenstein.

Since we have implemented multiple tools and conducted various experiments for this study any hidden bug in our implementation can affect our results. To mitigate this threat, we ran our tools on a large data set of real-world *versioned packages*, addressing bugs as we discovered them. Additionally, we implemented an extensive test suite for our tools and programs.

**External validity** In this study, we rely on the WALA and OPAL static analysis frameworks. We chose these tools because they are widely used, actively maintained, and we could contact their maintainers. However, this choice may result in inheriting their potential flaws. To mitigate this, we support more than one framework in our approach and designed our intermediate representation (PCGs) in a generic way. Thus, they can be extracted from the output of other frameworks as well. The information required in PCGs is present in the bytecode and does not require further analysis. Therefore, one could even construct them directly from the bytecode.

## 3.6 Future Work

This paper has introduced a summarization-based algorithm for CG generation. In this study, we used WALA and OPAL as the basis of our approach and as a baseline. During this study, we observed that the difference between different frameworks is significant in practice, to the extent that even for the most basic algorithm their outputs differ significantly. Therefore, we believe that it is valuable to investigate these differences and their effects on Frankenstein as future work. For example, we can explore the size of the difference between OPAL and WALA, and how the *stitched* version of these two differs.

The improvements in the scalability of CG generation also affect downstream tasks. In this study, we found evidence of imperfections in existing frameworks themselves. One direction for future work is to evaluate Frankenstein by comparing the effects of CGs generated by Frankenstein and existing frameworks on downstream tasks. For example, we could investigate whether the vulnerable call chains that Frankenstein can find differ from those found by WALA. Using Frankenstein it is also possible to perform more extensive analyses that require CGs. Hence as a next step, we plan to use this approach to generate a large number of CGs and conduct an ecosystem-wide analysis. It is valuable to understand the current state of vulnerability propagation or change impact in Maven Central. Another direction for future work is to study other languages and investigate whether our proposed approach can be applied to them. We plan to explore the differences and specific challenges that other languages may present for summarization.

## 3.7 Summary

Generating CGs is essential for many advanced program analyses. The inherent complexity of CG generation and the high memory requirements challenge scalability, making certain static analysis tasks impractical in resource-limited environments such as hosted build servers. In this paper, we propose a summarization-based approach Frankenstein that makes CG construction fast and lightweight. Instead of performing a whole-program analysis, we introduce a summarization-based algorithm that precomputes PCGs for all dependencies of an application and caches the results. For whole program CG generation, an algorithm stitches all PCGs on demand. In our extensive evaluation, we investigate the impact of Frankenstein on accuracy, speed, and memory usage. Our results are very promising when compared to the state-of-the-art approaches. Our results demonstrate that Frankenstein has a minimal impact on the soundness of the generated CGs and can achieve an F1 score of up to 98%. Additionally, we show a significant improvement in performance, with up to 38% lower execution time on average and memory usage of only 388 megabytes for the whole program analysis. We believe that the proposed approach in this paper addresses the current key challenges that limit the practicality of static analyzers. Our results show that pre-computing partial static analysis results has a high potential to improve CG generation. We propose a novel algorithm that is the first step towards scalable static analysis in CI tools.

# 4 On the relation of method popularity to breaking changes in the Maven ecosystem

Software reuse is a common practice in modern software engineering to save time and energy while accelerating software delivery. Dependency managers like Maven offer a large ecosystem of reusable libraries that build the backbone of software reuse. Breaking changes, i.e., when an update to a library introduces incompatible changes that break existing client programs, are troublesome barriers to this library reuse. *Semantic Versioning* has been proposed as a practice to make it easier for the users to find *safe* updates by encoding the change impact in the version number. While this practice is widely studied from the framework perspective, no detailed insights exist yet into the ecosystem perspective. In this work, we study violations of semantic versioning in the Maven ecosystem for 13,876 versions of 384 artifacts to better understand the impact these violations have on the 7,190 dependent *versioned packages*. We found that 67% of the artifacts introduce at least one type of semantic versioning violation, either a breaking change or an illegal API extension in their history. An impact analysis on breaking methods that (direct or transitive) dependents reference revealed strong centralization: 87% of publicly accessible methods are never used by dependents and among methods with at least one usage, half of the unique calls from dependents concentrate on only 35% of the defined methods. We also studied method popularity and could not find an indication that popularity affects stability: even popular methods break frequently. Overall, we confirm the previous result that *Semantic Versioning* is violated repeatedly in practice. Our results suggest that the frequency of breaking changes might be a sign of insufficient change-impact awareness on the ecosystem and we believe that developers require more adequate information, like method popularity, to

improve their update strategies.[1]

[1]This chapter is based on the following paper: Mehdi Keshani, Simcha Vos, and Sebastian Proksch. On the relation of method popularity to breaking changes in the maven ecosystem. Journal of Systems and Software, 2023 [79].

Software reuse is a pillar of modern software engineering. The availability of mature and powerful open-source libraries has boosted the productivity of developers both in open-source and industry. For easy release, discovery, and distribution, package managers like Maven, npm, or PyPi usually rely on centralized repositories. These repositories contain vast numbers of inter-dependent packages that build self-contained, ever-growing software ecosystems [180]. The use of dependencies has already been the focus of a rich body of previous works that studied, for example, the evolution of dependency networks [16, 17], bloated dependencies [18, 19], and vulnerable dependencies [20].

Depending on a third-party library means that a project accepts a coupling to the API of said library. However, libraries might introduce *breaking changes* in subsequent versions that change the API in an incompatible way, for example, by removing a method or by changing its arguments. Updates are often necessary for consumers to fix bugs or vulnerabilities, so such a breaking change to the API is often an unwelcome surprise for the developers of the depending project, which forces them to adapt their code. *Semantic versioning* has emerged as a best practice to signal compatibility of a change to the previous version, but it is voluntary and not enforced in the ecosystem.

Numerous studies already investigated various aspects such as usage [45, 46], evolution [47–50], and stability [51] of APIs. However, the closest related work is presented by Raemaekers et al. [37] who studied the relation of *semantic versioning* and *breaking changes* in Maven. The paper finds evidence that many libraries break versioning rules and introduce breaking changes even in non-major upgrades. However, they did not provide any details about the extent of the method-level impact on the ecosystem which is the main goal of our study.

In this paper, we advance the perspective on breaking changes from the library creator to the ecosystem. Instead of treating a breaking change to a library method as a single incident, we consider the methods' popularity in the ecosystem to weigh the severity of a breaking change. This is achieved by identifying all dependents of a library in question through a global dependency graph [17]. This global dependency graph includes a large part of the recent releases in the ecosystem. It is important to note that we can query this graph for all possible dependents of any given library that is released recently. All dependents start using a version after its release date. Therefore, if we select a library that was released x months ago, all of its dependents are at least x months old hence our dependent graph includes them. After generating the call graphs (CGs) for all dependents, we can identify all references to method definitions in the original library (through calls or inheritance). A breaking change can then be identified by comparing the list of extracted references to the available methods in the next version of the library. This methodology not only allows us to replicate previous work and identify breaking changes in the library but also helps us to reason about the severity of a breaking change for the ecosystem. We expect that widely-used methods are more damaging to break because they affect more users. In this study, we aim to answer three research questions: 1) How often do semantic versioning violations occur? 2) How is popularity distributed among library methods? and 3) Is there a relation between popularity and breaking changes?

We have created a sample of 384 artifacts and 19,639 unique versions of them. From each of these artifacts, we picked one version. For the picked versions we identified a total of 7,190 unique dependents on Maven Central. We used these dependents to infer method popularity. Our results confirm previous work by showing that 67% of artifacts violate semantic versioning. To better understand the effects that these violations have on the ecosystem, we have further investigated the methods' popularity. We found that 87% of publicly defined methods are never called by another library. From the remaining 13% only 35% receive half of all calls to the respective library.

In this study, we show that maintainers introduce breaking changes in popular methods as often as less popular methods. One possible explanation for this is that library maintainers may not always be aware of the popularity of their methods. It is important to note that the adoption of a library is typically in the best interest of its maintainers, and the lack of upgradability may sometimes hinder this goal. Providing sufficient information to library maintainers about the popularity of their methods has the potential to help them enhance the upgradability of their library. While some breaking changes may be inevitable, there are cases where the severity of certain breaking changes may be underestimated. In situations where a breaking change affects a widely used method, maintainers may decide to notify users with a major release.

The contributions of this study are as follows:

- A quantitative analysis of API method extensions and contractions that violate semantic versioning.
- An empirical study of the popularity distribution in typical APIs of Maven libraries.
- An investigation of the extent of user breakage that semantic versioning incompatibilities cause on Maven.

## 4.1 Related Work

We found three areas of research to be closely related to this paper: software ecosystems, dependency management, and APIs and breaking changes.

**Software Ecosystems** Multiple studies investigated the software ecosystems from different aspects. Decan et al. [180] found that dependency networks are growing over time by analyzing the evolution of different package managers. Moreover, they realized that a minority of packages are used by a majority of the network. Several studies [16, 17, 181] modeled software ecosystems using *graphs* with package versions as *nodes*. They used these graphs to study Maven Central and the CRAN. Their findings show that Maven ecosystem has a more conservative approach to updating dependencies than CRAN. They also studied *activity*, *popularity*, and *timeliness* of more than 1*M* artifacts in the Maven Central. However, these studies only consider package-level relations. Raemaekers et al. [33] presented the dataset of Maven containing information about 148K Java libraries and their CGs. The authors only provided a dataset of metrics and CGs but contrary to our work they do conduct any ecosystem analysis using this data. Hejderup et al. [130] proposed a fine-grained dependency network that uses CGs to model function interactions in the ecosystem. The authors present a methodology to construct and analyze this network. This study also does not provide insights into the method-level relations of the ecosystems.

**Dependency Management** Various studies studied different aspects of dependency management such as their updates, trends, and adoption. Several other studies [18, 19] studied the use of *bloated* (i.e., unused) dependencies. They showed that once a package becomes bloated in a project it is likely to stay bloated. They also found that bloated dependencies are mostly the result of transitive dependencies. Zapata et al. [20] studied developer reactions to known vulnerable dependencies. This study shows that 73.3% of clients using vulnerable dependencies are not running vulnerable code. Hence they confirm that analysis at the library level is an overestimation and function-level analysis is needed. Alrubaye et al. [182] automated library migration to save time and reduce the knowledge requirements for engineers. They use method-level changes in programs that already migrated and automate the procedure for others that are interested in migration. Mileva et al. [183] presented an approach to support developers in their decision to upgrade a dependency using *wisdom of the crowds*. Macho et al. [35] analyzed trends of changes in Maven build files. Kula et al. [107] presented a way to decide when a library needs to be updated based on its usage level. Kula et al. [139] conducted an empirical study that covers over 4,600 GitHub software projects and 2,700 library dependencies. The findings of this study show that 81.5% of the studied systems keep outdated dependencies. Kula et al. [36] studied the adoption habits and trust of maintainers towards new releases of an existing library. The study concludes that maintainers are becoming more trusting of new releases and becoming inclined to update their existing systems to the latest release.

The aforementioned studies inspect dependency-related topics. However, none of them empirically studies the API usages of the Maven ecosystem. Most of them only considered package-level relationships between packages. In this study, we will consider API usage and empirically study method-level networks. We leverage the *CGs* of libraries to provide insights about the *APIs* of the Maven with method-level precision.

**APIs and Breaking Changes** Some studies specifically targeted the library APIs and investigated their patterns, stability, usage, etc. Qiu et al. [45] studied API usage of 5k open-source projects. Their findings show that the API usage obeys a Zipf distribution and deprecated APIs are still widely used. Bavota et al. [47] studied the evolution of a set of projects. They show that when releases contain major changes a large amount of them are bug fixes. They also show that developers are more reluctant to upgrade when APIs are removed or altered. Wang et al. [46] empirically investigated the usage and updates of packages in Java projects. They also provide a prototype of a tool that alerts developers about updating packages. Xavier et al. [184] studied the frequency of *breaking changes*, the behavior of these changes over time, their impact on users, and the characteristics of libraries with a high frequency of breaking changes. Lima et al. [185] categorized APIs into *popular*, *ordinary*, and *unpopular*. They found that popular APIs often have more public methods, more lines of code, a higher complexity, higher relative complexity per method, change more frequently, and have more contributors. However, they also found that there is no change in the relative line of code, method name length, or the number of parameters of popular APIs and they are often used early in the development cycle and are often more unstable. Harrand et al. [34] performed a large scale study on Maven. They discovered that with sufficient users, all APIs seem to be used, but there is a concentration of usage on a small set of APIs. Meaning developers could focus on a smaller portion of

APIs and save time. Hora et al. [48] proposed a tool to extract rules by monitoring API invocation changes in the code history. This then can be used to keep track of the evolution of an API. Kim et al. [186] performed a large analysis of API refactorings and bug fixes. Their findings revealed that the number of bug fixes increased after refactorings, while the time required to address these bugs decreased. Kocci et al. [49] investigated changes that happen to APIs and classified them to gain a bigger picture of API evolution. Nguyen et al. [187] presented LibSync, a tool for developers who want to upgrade their dependencies. It suggests to users a way of adapting their API usage by learning from clients that have already migrated to a new library version. Hora et al. [188] performed an exploratory study to observe API evolution and its impact on the Pharo software ecosystem. Lamothe et al. [50] reviewed the literature on APIs and API evolution. They conclude that the main challenges are identifying factors that drive API changes, creating a uniform benchmark for research evaluation, and the impact of API evolution on various programming languages. Raemaekers et al. [51] proposed four different metrics for the stability of the libraries. Raemaekers et al. [37] studied how new releases of a library impact the client libraries and their semantic versioning. They found that on average one in three releases introduce breaking changes that produce compilation errors that need to be addressed. The above-mentioned studies investigated APIs from different aspects. However, none of them focuses on an analysis of public APIs from the consumer perspective at the ecosystem level.

## 4.2 Experimental Setup

Our approach contains multiple components that enable us to perform the desired analyses. In this section, we first provide a brief overview of these components and the overall methodology. Then we elaborate on different components for example the Sampler component that is used for the data selection.

### 4.2.1 Overview

Within Maven Central, all packages are uniquely identified by a triplet consisting of `groupId:artifactId:version`. In this paper, we refer to such an identifier as a *Versioned Package* (VP). We also use the term *artifact* to refer to a package but not a

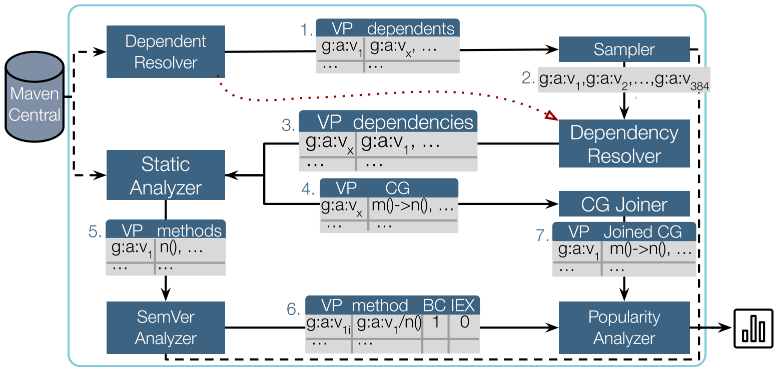


Figure 4.1: Overview of the Methodology

particular version, i.e. `groupId:artifactId`.

Figure 4.1 illustrates an overview of the methodology of the paper. We resolve dependents of all *versioned packages* released in a particular time frame on Maven. For this, we use our Dependent resolver component which is described in Section 4.2.2. This component internally uses Dependency Resolver (red arrow in the overview figure). The next step is selecting a subset of *versioned packages* to analyze. We select 384 *versioned packages* for the analysis using the Sampler component. This component is described in detail in Section 4.2.3. For the *dependents* of any selected *versioned package*, we resolve their *dependencies*. We describe our dependency resolution in Section 4.2.4. Note that, the list of dependencies of each dependent contains the original *versioned package* that was selected as a target for the analysis.

In the rest of the paper, we call these *versioned packages* target *versioned packages* and their corresponding artifacts target artifacts.

In the next step, all dependency sets are transmitted to the Static Analyzer component (see Section 4.2.5). This component performs two crucial tasks. Firstly, it generates a Call Graph (CG) for each dependency set and transmits it to the CG Joiner. Secondly, it forwards the *method definitions* of the target artifacts and their respective versions to the SemVer analyzer (explained in Section 4.2.6). The CG Joiner (refer to Section 4.2.5) combines these Call Graphs into a single joined CG for each target *versioned package*.

These CGs contain all method calls from dependent *versioned packages* to the target *versioned packages*. All other edges such as internal calls of these libraries are filtered. Finally, we analyze method declarations to find violations of semantic versioning in SemVer Analyzer resulting in *violation information*. Popularity Analyzer then uses this information together with joined CGs to find if the violations affect the most popular methods.

Figure 4.1 also shows the data that goes through this pipeline alongside an example. Consider $g : a : v_1$ as a *versioned package* that is released within our target time frame. $g : a : v_x$ is one of its dependents resolved by Dependent Resolver. In the second step Sampler selects $g : a : v_1$ as a target *versioned package*. Afterward, Dependency Resolver includes $g : a : v_1$ among the dependencies of $g : a : v_x$. In the fourth step, Static Analyzer generates a CG for all dependency sets including $g : a : v_x$'s. This component also uses Maven to find all the versions of selected artifacts and extracts their method definitions such as method `n()` in $g : a : v_1$.

After acquiring all the versions of the selected artifacts, the SemVer Analyzer computes any breaking changes and illegal API extensions, such as the removal of the method `n()` in $g : a : v_{1i}$, which is the subsequent minor release after $g : a : v_1$. Meanwhile, the CG Joiner joins the CGs it receives, resulting in a unified CG that contains an edge from method `m()`, defined in $g : a : v_x$, to method `n()`, defined in $g : a : v_1$. Using all the calls to the methods defined in the target *versioned packages*, the Popularity Analyzer calculates the popularity values for methods. For instance, the popularity value of `n()` is greater than zero because it is utilized at least once by $g : a : v_x$.

By removal of `n()` method `m()` would break should developers of $g : a : v_x$ decide to update to $g : a : v_{1i}$. We use popularity values, breaking changes, and illegal API extension information for the reports and figures of this study.

## 4.2.2 Dependent resolver

*Dependent resolution* is the process of identifying *versioned packages* that refer to a particular target *versioned package*, either directly or transitively. One needs to first perform dependency resolution for all *versioned packages* on the ecosystem. Using this information, one can create a so-called dependency graph. This graph determines which *versioned packages* are dependent on a given *versioned package* by retrieving the incoming edges of the given node. We replicated the approach presented by Benelallam et al. [16] to create this graph.

We created this graph by including releases from 1st of October 2021 to 31st of March 2022. This contains 537K *versioned packages* that are released in this time frame and we add them as initial nodes of this graph.

Maven index [88] lists all *versioned packages* that are being released. We use this index to include releases of our target time frame. We also include all dependencies of each *versioned package* by analyzing their *POM* files. We do not resolve the exact versions of these dependencies in this phase and only store the information available in the *POM* file. Most dependency definitions specify the exact versions. For these cases, we simply add a dependency edge between two *versioned packages*. However, some *POMs* define version range dependencies. This is less common than exact versions dependency definitions. For these cases, we also store the range information separately. Consider *versioned package* g : a : 0 and g : a : 1 that both define the dependency g : d : [ 1 , 5 ] in their *POM* files. This dependency refers to all releases between version 1 and 5. We use this example in the rest of this paragraph to show how Dependent Resolver functions. While adding g : d as a dependency we store the range information [ 1 , 5 ]. We resolve the exact versions only on demand once we receive queries. For each query about the dependents of a given *versioned package*, we first return all potential dependents that exist in the whole graph. This includes both direct and transitive dependents. For example, when A depends on B and B depends on C we also count A as a dependent of C. Next, we use the dependency resolver (see Section 4.2.4) to resolve the dependencies of each potential dependent at the current moment. After that, we check whether or not the given *versioned package* is present in their dependency sets. For example, when we receive a query about the dependents of g : d : 1 we return both g : a : 0 and g : a : 1. We then resolve dependencies of g : a : 0 and g : a : 1 to validate whether or not g : d : 1 is among their dependencies. Assuming that g : a : 0's dependency set does not contain g : d : 1 and g : a : 1's does, we keep g : a : 1 and eliminate g : a : 0 from the list of its dependents. We opted for analyzing a particular time frame because it enables us to find the complete set of dependents for any *versioned package* that is released within or after our target time frame. Users can depend on each *versioned package* only after its release, not before, so this approach provides a comprehensive view of the dependents existing on Maven.

## 4.2.3 Sampler

We have continuously collected data from Maven central and created a dataset that represents the current state of the ecosystem. This dataset contains all *versioned packages* that are released between the 1st of October 2021 and 31st of March 2022. We call this time frame our sampling frame. The sampler component is responsible for selecting a set of *versioned packages* that is representative of this dataset. This sampling is done because CG

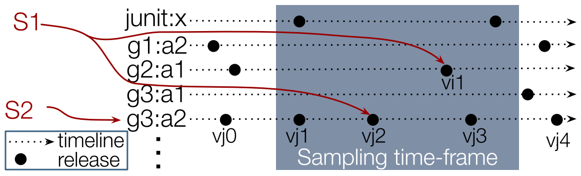


Figure 4.2: Random selection example

generation is a highly expensive task and not feasible to perform for all existing *versioned packages*. Figure 4.2 shows an example of this sampling. In the remainder of the Section, we use this example to describe the steps we take in our data selection. In the sampling time frame, some artifacts may have only one release (`g2:a1`), some may have multiple releases (`junit:x, g3:a2`), and some may have no release (`g1:a2, g3:a1`). As the first step of our selection, we filter the artifacts without any release in the sampling frame (`g3:a1, g1:a2`). Maven Central repository contains approximately $9M$ indexed packages. However, the aforementioned 6 months time frame contains approximately $537K$ *versioned packages*. These *versioned packages* are released within the sampling frame. For example, in Figure 4.2 there are 6 *versioned packages* within the sampling frame (gray area) including two `junit:x`, one `g2:a1`, and three g3:a2 releases. Figure 4.3 shows the number of *versioned packages* that are released within the sampling frame on Maven Central.

Before we sample, we apply two filters to the sampling frame; which allows us to create a more representative set of *versioned packages*. The first filter we apply is to remove testing-related *versioned packages* that contain the keywords `assertj`, `junit`, `mock` and `test` from the dataset. We do this because our purpose is to analyze the regular library API usage while testing-related libraries have different usage patterns. Also, the *versioned packages* uploaded to Maven Central usually do not contain sufficient bytecode information to analyze the testing-related part of the code. In the example, artifact `junit:x` will be filtered after this step which leads to four remaining *versioned packages* derived from two unique artifacts i.e. `g2:a1, g3:a2`. After applying this filter on Maven data, approximately 380K *versioned packages* remain. These *versioned packages* are derived from 10.6K unique *artifacts*. To avoid a bias towards artifacts with a high release frequency, we randomly select only one version from each of the 10.6K unique artifacts that we could identify in the dataset. In the example, this step is specified with *S1*. I.e. from each remaining artifact (`g2:a1, g3:a2`) we randomly select one version. In case of `g2:a1` there is only one *versioned package* (`Vi1`) in sampling the frame, while `g3:a2` has three versions and we pick the second one (`Vj2`) randomly as shown in the Figure 4.2.

Popular *versioned packages* with many dependents influence the overall ecosystem more, thus we perform weighted random sampling based on the number of dependents (direct and transitive) that *versioned packages* have. In example, assume that `g3:a2:vj2` has 20 dependents and `g2:a1:vi1` has 10. Consequently, `g3:a2:vj2` is twice more likely than `g2:a1:vi1` to be selected in this step. Sampler uses Dependent resolver (see Section 4.2.2) to get the number of dependents that each *versioned package* has. Table 4.1 shows the number of dependents per *versioned package*. More than 7.5K artifacts do not have any dependents, which using weighted random sampling cannot be selected. To verify the correctness of this, we randomly picked 10 of these cases and manually

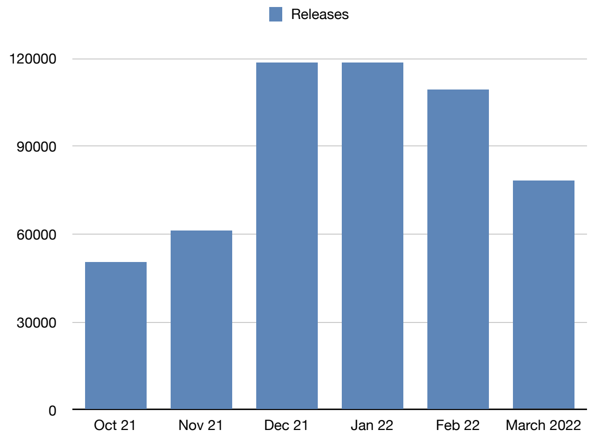


Figure 4.3: Number of releases in the dataset per month

checked their dependent number with two other sources: Libraries.io[2] and the usage tab of Maven. They all had no dependents. We observed that some of them are very new releases. Therefore, they did not have the chance yet to attract users. The rest are unpopular *versioned packages* due to different reasons such as being from a very unpopular package. We believe these cases happen because the majority of the *versioned packages* are barely used, especially in the early stages of their lives while a minority are highly used. Previous research [17, 34, 180] as well as our findings show very similar patterns in library usage within the ecosystem. To achieve generalizable results, we made sure to select a representative subset of libraries. The dependent distribution for *versioned packages* follows an inverse logarithmic distribution. We selected 384 *versioned packages* from the 3.1K *versioned packages* with 10.7K non-unique dependents (7,190 unique), which gives our results a confidence level of 95% and max the margin of error of 5%.

Our popularity analysis requires CG generation for all dependents of selected *versioned packages*. Thus we use the described 384 *versioned packages* in *Popularity Analyzer* component (Section 4.2.8). However, Static Analyzer (Section 4.2.5) and SemVer Analyzer (Section 4.2.6) use all versions of the selected artifacts that are available on Maven. In Figure 4.2 we show this by *S2* as the final step of our selection process. Using weighted random sampling we pick `g3:a2:vj2`. This *versioned package* is used in *Popularity Analyzer* component for the impact analysis. However, Static and SemVer Analyzers inspect all versions of the corresponding artifact `g3:a2` including `vj0, vj1, vj2, vj3`, and `vj4`.

### 4.2.4 Dependency resolver

*Dependency resolution* is resolving what dependencies a *versioned package* needs to compile, build, test or run. These dependencies are (directly or transitively) specified in the *pom.xml* file. For transitive dependency resolution, one needs to recursively get the dependencies of all dependencies and consider the Maven-specific resolution rules for solving conflicting definitions within the dependency set.

[2]https://libraries.io/

Table 4.1: Number of dependents

| **Dependents** | ***versioned package*** |
|---:|---:|
| 0 | 7533 |
| 1 | 2091 |
| 2-9 | 844 |
| 10-24 | 88 |
| 25-49 | 33 |
| 50 | 27 |

Dependency resolution in Maven is not deterministic because of version ranges [176]. New releases on Maven may change the outcome of dependency resolution for existing projects, even if the specified dependencies in *pom.xml* are stable. One such scenario arises when there are conflicting versions in transitive dependencies. Suppose we have two dependencies, $D_1$ and $D_2$, that rely on different versions of the library $L$, for instance, $D_1 \rightarrow L_{v1}$ and $D_2 \rightarrow L_{v2}$, and project $X$ depends directly on both $D_1$ and $D_2$. This leads to a dependency conflict because $X$ cannot include both $L_{v1}$ and $L_{v2}$ in its dependency set simultaneously. Different versions of the same package may have varying APIs and behaviors, so Maven permits only one version from each package to be present in the dependency set after resolution.

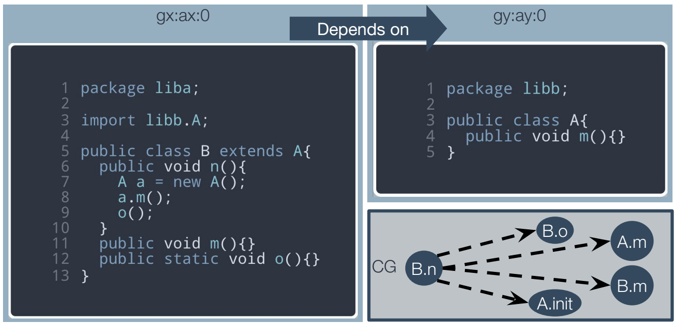
When a conflict occurs, Maven addresses it by conducting a breadth-first search and choosing the closest version of the conflicting dependency to the root. For instance, if $X$ defines $D_1$ before $D_2$, the closest version of $L$ to the root ($X$) is $L_{v1}$. As $D_2$ defines $L_{v2}$ in the dependency tree of $X$, $L_{v2}$ appears after $L_{v1}$. In this example, the dependency set may alter from the perspective of $D_2$ when $X$ relies on $D_1$, compared to when it does not. Maven has numerous similar cases of dependency resolution, making it excessively complicated. We do not implement this feature ourselves, but instead, we use a re-implementation of Maven's built-in dependency resolution from a Java library [178]. This tool enables us to include all dependencies, including transitive ones, when resolving dependencies. As a result, we handle transitive dependencies similarly to direct dependencies.

### 4.2.5 Static analyzer

The static analyzer component extends an existing framework OPAL [143] to perform two types of analyses. This analyzer both generates CGs for *versioned packages* and their dependencies and extracts the method definitions in versions of a given artifact. In the following, we explain each of these analyses.

**Generation** After receiving a dependent and its dependency set we perform *class hierarchy analysis (CHA)* [73] to ensure all possible method calls are contained within the CG. A class hierarchy analysis determines a program's class inheritance graph and the set of methods defined in each class. Using these two pieces of information we add all possible invocations of any method to the CG.

Figure 4.4 illustrates a minimal dependency set and its corresponding CG, which we will use as an example to elaborate on the steps we take in our analyses. As previously



```java
package liba;

import libb.A;

public class B extends A{
  public void n(){
    A a = new A();
    a.m();
    o();
  }
  public void m(){}
  public static void o(){}
}
```

```java
package libb;

public class A{
  public void m(){}
}
```

Figure 4.4: Example dependency set and its CG

explained, CHA analysis overapproximates and draws edges to all possible implementations of a target method. For example, in the running example, method n in class B calls method m using object a. However, since the CHA algorithm does not reason about the control flow of the program, the exact type of a is unknown. Therefore, B.n is connected to both A.m and B.m, as illustrated in Figure 4.4. This becomes more complicated when object a is not defined in the same scope, such as when it is passed as an argument to method n. It is worth noting that A could also be an interface, and m could be an abstract method. In such cases, the algorithm would perform similarly, except it would not draw an edge to A.m because an abstract method is not executable. Nonetheless, the algorithm considers A in the type hierarchy and draws edges to its subtypes.

It is important to note that the imprecision caused by this algorithm only occurs for virtual dispatch calls, which are calls that cannot be statically resolved. Despite this imprecision, we believe CHA is a suitable trade-off for our analysis, balancing soundness, scalability, and precision. More precise analyses, such as CFA CG generation, lack scalability. Dynamic CG generation is also not practical for our use case due to a lack of scalability and coverage.

In this part, we also generate unique *identifiers* for each method within the ecosystem. We call these identifiers *Global IDs* (GID). These *GID*s help us join the generated CGs in CG Joiner 4.2.7.

**Method definition extraction** For each target artifact, we first query Maven for all versions. Having all versions we extract all *public method definitions* that they have and assign them a *GID*. It is important to note that we only consider method definitions (methods with a body) because CGs only resolve edges to defined methods since declared methods (without a body) cannot be executed. In the next steps of the study, we limit the scope to methods with body i.e. method definitions to be able to connect semantic versioning violations to method popularity. Moreover, since public visibility is the most common access modifier for library usage we consider such methods as API endpoints that we analyze. Hence, we use the term *publicly defined method* to refer to such methods.

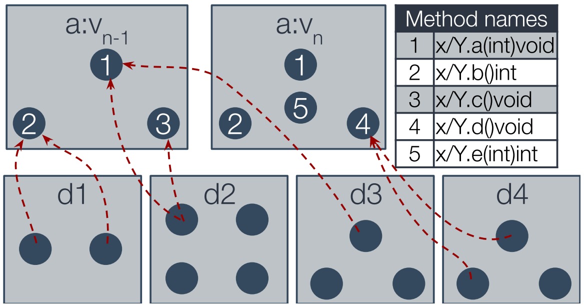


Figure 4.5: Two releases of an example artifact and their dependents.

## 4.2.6 SemVer analyzer

This component analyzes *semantic versioning* violations in a given *versioned package*. We categorize these violations into two categories. The first category is *breaking changes*, which are changes that break compatibility within a major version. The second category is *illegal API extensions*, which is considered an extension of the API in patch versions.

We define breaking changes as changes that alter or remove a method signature. In addition to the name, our definition of method signature also includes return type and all arguments. Such changes should be introduced in major version upgrades only, *Semantic Versioning* is violated when these changes occur in minor or patch releases. This means that, when a new minor or patch version is released, a public method signature has been altered or removed which breaks the compatibility of an API. This leads to issues in case this method is called by some dependents. Therefore, changes that remove method signatures are not allowed, unless they are part of a major version upgrade.

To detect such violations we analyze the evolution of method signatures in the release history within the sampling frame. We look at the set of method signatures publicly available in a given *versioned package* $v_n$ and compare it to the previous version's set of method signatures, the goal is to identify cases, in which signatures that existed before are removed in $v_n$ (See Equation 4.1).

$$BC(v_n) = \begin{cases} Sig(v_{n-1}) - Sig(v_n) & \exists\{(v_n, v_{n-1}) \in Major(v_n)\} \\ \emptyset & otherwise \end{cases} \tag{4.1}$$

where:

$BC(v_n)$ = Set of breaking change signatures introduced in $v_n$
$Major(v_n)$ = Set of patch and minor in the same major version as $v_n$
$Sig(v_n)$ = Set of public method signatures defined in $v_n$

Figure 4.5 shows two consecutive releases of artifact a. We use this figure as a running example to explain the next steps of our approach. For conciseness, we use a number to refer to each method. This figure also shows fully qualified method names next to their corresponding number. To calculate the breaking changes of *versioned package* $a : v_n$, assuming that $v_n$ and $v_{n-1}$ are within the same major version, we need to subtract the set of methods in $a : v_n$ from $a : v_{n-1}$. This means $\{1,2,3\} - \{1,2,4,5\}$ resulting in $\{3\}$ which can also be illustrated as $BC(a : v_n) = a : v_{n-1}/x/Y.c()void$.

The second category of violations is the extension of the API in a patch version. We use a similar approach as before with a small adaptation. We iterate over the different patch versions and detect if a new method signature is added. This means there has been an extension of the API within a patch version. We find illegal API extensions of *versioned package* $v_n$ by finding the difference between the previous patch version's set of method signatures and the set of method signatures in $v_n$, the goal is to identify cases, in which a signature did not exist before but was added to the $v_n$ (See Equation 4.2).

$$IEX(v_n) = \begin{cases} Sig(v_n) - Sig(v_{n-1}) - Up(v_n) & \exists\{(v_{n-1}, v_n) \in Minor(v_n)\} \\ \emptyset & otherwise \end{cases} \tag{4.2}$$

where:

| | |
|---|---|
| $IEX(v_n)$ | = Set of signatures illegally added to $v_n$ |
| $Up(v_n)$ | = Set of updated signatures in $v_n$ |
| $Minor(v_n)$ | = Set of patch releases within the same minor version as $v_n$ |
| $Sig(v_n)$ | = Set of public method signatures defined in $v_n$ |

Our approach shares similarities with a conventional diff calculation function in that it treats updated parts as a removal and an addition. Consequently, the updated methods belong to both the 'BC' and 'IEX' categories if we do not exclude them. While these updated methods qualify as 'BC' because they may impact users, they should not be considered as 'IEX' because they are related to previously existing methods and not independently added. To identify the methods that truly belong to the 'IEX' category, we subtract the updated methods ($Up(v_n)$) in Equation 4.2. To compute the set of potential updated methods, we use a heuristic approach that only considers sets of fully qualified signatures in releases. The heuristic approach, outlined in Algorithm 1, identifies five categories of changes in a release, including package name, class name, method name, parameters, and return types refactoring. This heuristic searches for pairs of removals and additions that contain only one renamed piece of fully qualified method names. For instance, if an added method has a similar package name, class name, method name, and parameters to a removed method and only differs in the return type, we consider it as one potential update. We do not consider other cases of updates in this heuristic such as when two or more elements of the signature are updated. Although this approach may not capture all types of updates that can occur in a release, it provides a reasonable approximation for our study.

$Up(v_n)$ is the only source of unsoundness in our calculation of 'IEX' and can be replaced with more accurate approaches if necessary. Notably, achieving accuracy in this context would require calculating the differences between the complete binary files of consecutive releases, which is resource-intensive and impractical in terms of scalability. Therefore, this approach falls outside the scope of our study.

Returning to the example in Figure 4.5, we illustrate the process of calculating illegal API extensions for *versioned package* $a : v_n$. First, we subtract the set of methods in $a : v_{n-1}$ from $v_n$. That is, $1,2,4,5 - 1,2,3$, which results in $4,5$. Alternatively, we can express this as $IEX(v_n) + Up(v_n) = x/Y.d()void, x/Y.e(int)int$. Next, using Algorithm 1, we compare

**Algorithm 1** Find Updated Methods

```
Require: $added$: list of method signatures added to $v_n$
Require: $removed$: list of method signatures removed from $v_n$
Ensure: $result$: list of updated signatures in $v_n$
 1: $result \leftarrow \{\}$
 2: for $a \in added$ do
 3:     for $r \in removed$ do
 4:         $r.sig \leftarrow [r.pkg, r.class, r.name, r.params, r.return]$
 5:         $a.sig \leftarrow [a.pkg, a.class, a.name, a.params, a.return]$
 6:         if r.sig differs from a.sig in only one element then
 7:             $result \leftarrow result \cup \{a\}$
 8:         end if
 9:     end for
10: end for
11: return $result$
```

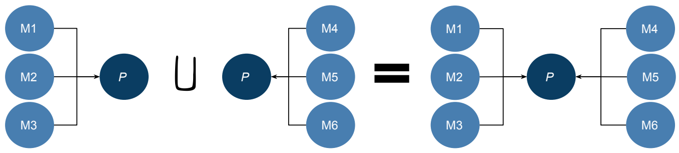


Figure 4.6: Example of CG joining

all pairs of additions and removals in $v_n$. The set of additions in $v_n$ is 4, 5 and the set of removals is 3. This means that in this algorithm, we compare $x/Y.c()void$ to $x/Y.d()void$ and $x/Y.e(int)int$. The only two cases with a single element difference (method names) are 3 and 4. Thus, we count 4 as an updated version of 3. This implies that $Up(v_n) = x/Y.d()void$ and $IEX(v_n) = x/Y.e(int)int$.

### 4.2.7 CG joiner

Using the list of dependents, we need to determine how every dependent interacts with target *versioned package*. Within Static Analyser 4.2.5 we generated a CG for every dependent and its dependencies. Note that the dependencies of each dependent contain a target *versioned package*. Initially, we build one unique CG per dependent, but as every node has a unique identifier, we can join these individual CGs to get a joint CG for one target *versioned package*. Every node within our CGs has a unique identifier. While analyzing each *versioned package* for the first time we use a combination of Maven coordinates of the *versioned package* and fully qualified names of methods (including java package and return types) to uniquely identify each method within the ecosystem. Consequently, we can join CGs that are related to a target *versioned package* into a single CG. See Figure 4.6 for an example of CG joining. In this Figure, a node $P$ is common between two graphs. When we join these graphs the result shows what other nodes call this node from both graphs. See Equation 4.3 for the mathematical formula behind joining CGs.

$$G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{(a,b) \ : \in V_1, b \in V_2\}) \tag{4.3}$$

where:

$G_x$ = Graph $x$
$V_x$ = Vertices of graph $x$
$E_x$ = Edges of graph $x$

After joining the CGs of a target *versioned package* we have a representation of the interactions between *versioned package* and its dependents, in an individual CG. However, this joined CG contains many edges that are irrelevant to our study. Every possible edge that happens in the context of each dependent is present in this joined graph, such as internal calls of the dependents. In the next step, we filter all irrelevant edges. Suppose we are analyzing target *versioned package x*, we reduce this joint CG to the edges that have a source outside of $x$ and a target inside of $x$. Consider the running example in Figure 4.4. Suppose `gy:ay:0` is one of our target *versioned packages.* In the filtering step, we iterate over all four CG edges shown in this figure. For each of them, we check the two aforementioned conditions. All of these edges pass the first check which is whether or not their source method is defined outside of `gy:ay:0`. This is because `B.n` is defined in `gx:ax:0` and not in our target *versioned package* (`gy:ay:0`). In the second condition check, however, two of the edges are identified as filterable edges. Since `B.o` and `B.m` are defined within `gx:ax:0` we filter `B.n->B.o`, `B.n->B.m`. Similarly, we process any existing edge in the joined CG. This procedure also filters indirect calls to *versioned packages.* For example, a call from another method to `B.n` would be filtered since `B.n` is not defined within target *versioned package.* In this study, our focus is the intentional usage of library methods and indirect calls in the CG do not capture them. The remaining CG only includes the method-level interactions between the library and its dependents. This allows us to determine popularity scores and influence ratings based on all interactions within a given *versioned package.*

### 4.2.8 Popularity analyzer

This component calculates two types of popularity values. Firstly, it calculates the popularity of the target *versioned packages.* And secondly, it calculates the popularity of methods within them. We are inspired by Raemaekers et al. [37] to use the term popularity in this study. For the library popularity, we divide the number of dependents of *versioned package* $v$ by the number of all unique dependents (7,190) to calculate the popularity of $v$ (see Equation 4.4). The value reflects the popularity of a *versioned package* among dependents.

$$P(v) = \frac{|Dependents(v)|}{\left|\bigcup_{t \in T} Dependents(t)\right|} \tag{4.4}$$

where:

$P(v)$ = popularity of *versioned package v*
$Dependents(v)$ = dependent set of *versioned package v*
$T$ = set of target *versioned packages*

Consider the Figure 4.5. To calculate the popularity of $a : v_n$ we should divide the number of dependents that $a : v_n$ has by the number of all dependents. Assuming that our dataset only includes four dependent *versioned packages* ($d1, d2, d3, d4$) we should divide one by four. $d4$ is the only dependent that uses $a : v_n$ thus $P(v_n) = 1/4$. This value is $3/4$ for $P(a : v_{n-1})$.

Having the joined CGs we count the distinct dependents that call each method of the target *versioned package*. We then divide this by the number of all dependents that the target *versioned package* has to understand the relative popularity of a method among its dependents. We devise a simple metric called *distinct dependents usage*. Equation 4.5 shows this metric for a given method $M$.

$$DR(m) = \frac{1}{|Dependents(p)|} \sum_{d \in Dependents(p)} \begin{cases} 1 & \exists\{d, m\} \in CG_P \\ 0 & otherwise \end{cases} \tag{4.5}$$

where:

| | |
|---|---|
| $m \in p$ | = *versioned package* p defines method m |
| $DR(m)$ | = ratio of dependents that call the method $m$ |
| $Dependents(p)$ | = dependents set of the target *versioned package* $p$ |
| $\{d, m\}$ | = edge between a dependent $d$ and a method $m$ |
| $CG_P$ | = joined CG of $p$ |

Consider Figure 4.5 once again. The popularity value for $a : v_{n-1}/x/Y.a(int)void$ is $2/3$ since out of three dependents of $a : v_{n-1}$ ($d1, d2, d3$) two of them ($d2, d3$) call this method. Moreover, $DR(a : v_{n-1}/x/Y.b()int) = 1/3$ since only $d1$ calls this method. $DR(a : v_{n-1}/x/Y.c()void)$ equals $1/3$ because only $d2$ calls it. Finally, $DR(a : v_n/x/Y.d()void) = 1/1$ as $d4$ is the only dependent of $a : v_n$ and the only caller of this method.

## 4.3 RQ1: How often do semantic versioning violations occur?

The first category of violations of semantic versioning is through breaking changes, which are changes that break compatibility. We expect that the evolution of an API sometimes leads to incompatibility, due to developers removing or changing the existing set of method signatures of an artifact within a minor or patch release. The second category of violations of semantic versioning is the extension of an API through patch versions. Hence, we investigate the extent of semantic versioning violations in the first research question.

As a first step, we filter the *versioned packages* that do not comply with the semantic versioning structure. I.e we filter *versioned packages* with a qualifier such as pre-releases, snapshots, betas, etc. We do this because semantic versioning does not provide any rules for them. Selected artifacts have 5,763 releases with one type of such qualifiers. However, they have 13,876 releases that adhere to the default structure of the semantic versioning and we use them in our study.

We use *SemVer Analyzer* as described in Section 4.2.6 to retrieve the set of breaking change methods in all remaining target *versioned packages* and their newer versions. We compare the retrieved set of breaking change methods with the set of total methods defined

Table 4.2: Type of release per artifact for the selected artifacts

| | mean | median | std | sum |
|---|---|---|---|---|
| **major** | 1.4 | 1.0 | 2.1 | 509 |
| **minor** | 11.1 | 5.0 | 16.6 | 3894 |
| **patch** | 26.8 | 11.0 | 55.2 | 9473 |
| **total** | 39.3 | 23.0 | 60.1 | 13876 |

in respective artifacts and calculate the percentage of methods that break in each artifact. Using this data, we can show the extent of violations within the ecosystem at the library level. We then compare the number of methods that artifacts define and the number of breaking changes they introduce. This information helps us discover any method-level trends between the number of methods and the number of violations if they exist. We conduct similar analyses for illegal API extensions as well.

To understand the extent of semantic versioning violations we analyze the selected artifacts. Table 4.2 shows the number of releases per artifact. As this table shows, patch releases (9,473) happen more often than minor releases (3,894). This is expected because multiple patches are usually released between two minor releases. The same explanation is also valid when comparing the number of minor (3,894) and major (509) releases. Overall, we analyze all releases (13,876) of 384 selected artifacts. In the rest of this section, we investigate how often breaking changes occur and, if they occur, what portions of the artifacts are affected. Firstly, we found that among selected artifacts 244(63%) introduce at least one breaking change in their history. Secondly, as Figure 4.7 shows the percentage of methods involved in breaking changes differs among artifacts that have at least one breaking change. This figure shows the percentage of public methods, defined in an artifact, that are involved in breaking changes. In 22(9%) of artifacts, less than 1% of methods are involved in breaking changes. These percentages are calculated for only artifacts with breaking changes. On average, 19% of methods of the artifacts with breaking changes are affected by the breakage. In addition, we can notice that for higher ratios, only a small number of artifacts have such a high ratio. Only 22(9%) artifacts have more than 50% of their methods involved in breaking changes. Fifty-five percent of artifacts feature a ratio that is not bigger than 15%. However, several outliers exist that feature a noticeably higher ratio, indicating that these artifacts contain many breaking changes.

Now we realize different artifacts break to different extents. That is the number of methods involved in breaking changes highly differ between artifacts. We suspect that some libraries may break more methods because they have more methods in total. To this end, we investigate whether or not the total number of methods that these artifacts define is related to the number of breaking change methods. Figure 4.8 shows the total number of public methods that artifacts have against the total number of breaking changes that occur in them. The axes in this figure are logarithmically scaled because the number of methods and breaking change methods highly differ between artifacts. Therefore, it is not practical to show them in a linear plot. As can be seen in this figure there is a tendency for artifacts with more methods to involve more methods in breaking changes. We fit a linear regression model with a confidence interval of 95% to better show this trend. The blue line in the

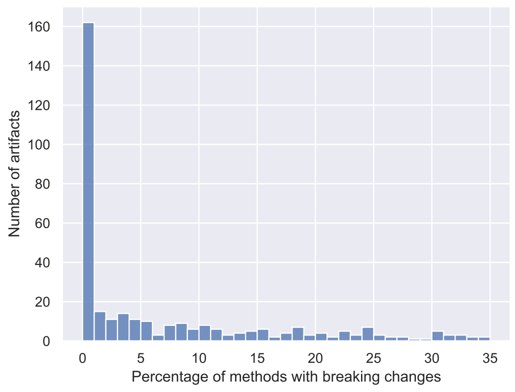


Figure 4.7: Ratio of methods involved in breaking changes

figure shows this regression model. We conclude that indeed a larger number of methods leads to a larger number of breaking changes. However, we observe that the number of breaking changes does not grow as rapidly as the number of methods. This is in line with the findings of Raemaekers et al. [37]. They found a correlation between the number of methods in a library and the number of breaking changes and showed that bigger libraries introduce more breaking changes. They also reported that 30% of all releases contain at least one breaking change. Unlike them, in this RQ we conduct artifact-level analysis. To further understand how our results compare to theirs we calculated the number of minor and patch releases that contain breaking changes. We found that 14% of minor and patch releases contain breaking changes. We did not include the major releases here because major releases are allowed to have backward incompatible changes according to semantic versioning. The difference can be explained by another finding of Raemaekers et al. [37] that says adherence to semantic versioning principles increases over time. For example, they reported a decrease of breaking changes in non-major releases from 28.4% in 2006 to 23.7% in 2011. This is a positive observation about the ecosystem and shows significant improvement in practices over time.

We showed that there exist some artifacts that introduce the first category of semantic versioning violation, breaking changes, and they do this to different extents. Now, we investigate the second type of semantic versioning violation, Illegal API extensions. Therefore, we inspect the number of methods that are added in patch releases. In Figure 4.9, the number of artifacts is plotted against the percentage of methods of artifacts that are added via illegal API extensions. One may notice the distribution is similar to breaking changes (Figure 4.7), however, the number of artifacts with API extensions is overall a bit lower than breaking changes. More specifically, 207(54%) of the artifacts introduce at least one illegal API extension in their history. Note that, the set of artifacts is not equal to the set of artifacts that feature breaking changes. Overall, 257(67%) of artifacts introduce at least one type of semantic versioning violation. It is worth mentioning that artifacts with illegal extensions on average add 14% to their set of existing methods through the illegal

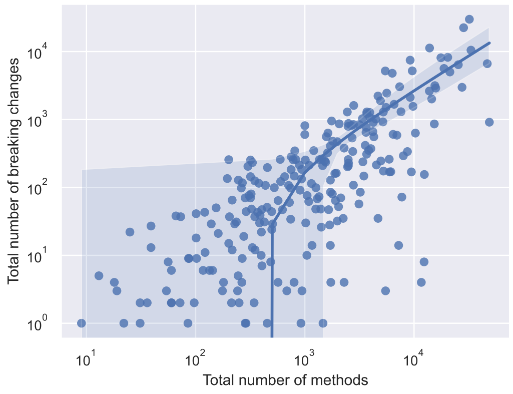


Figure 4.8: Increase of breaking changes in relation to the total number of methods

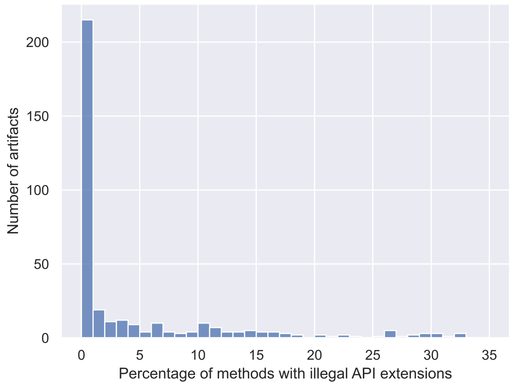


Figure 4.9: Ratio of methods involved in illegal API extensions

API extension. Thirty-eight (18%) of these artifacts illegally extend their API methods up to 1% and only 12(5%) of them add 50% or more methods via illegal extensions.

Finally, after elaborating on illegal extensions, we realize that similar to breaking changes the extent of the effect is different between libraries. Therefore we aim to understand whether or not this extent is related to the overall size of the artifacts. One could expect that bigger libraries may prevent adding yet more methods in general not to mention adding via illegal extensions. However, surprisingly Figure 4.10 shows that similar to breaking changes the number of illegal API extensions also grows along with the number of methods.

Sixty-three percent of the artifacts introduce breaking changes, and 67% of the artifacts feature at least one type of semantic versioning violation. Furthermore, the more public methods artifacts define the more likely it is that they introduce semantic versioning violations.

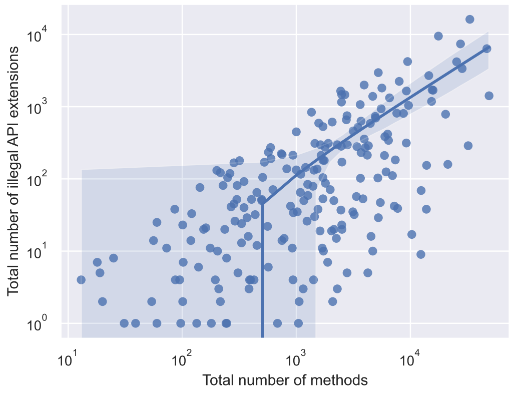


Figure 4.10: Increase in number of API extensions in relation to the total number of methods

## 4.4 RQ2: How is popularity distributed among library methods?

Now that we have answered the first research question, we have an insight into how often violations of semantic versioning occur in general. However, understanding the importance of the methods that break from the perspective of users within the ecosystem is also critical. Hence, as the next step, we investigate the popularity of the methods. Despite any relation that popularity and breaking changes may have, all attributes of popular methods propagate more within the ecosystem. Therefore, it becomes important to understand what portion of libraries are generally popular and widely used. Maintainers could for example pay extra attention to the important parts of their libraries and be more careful not to break them.

To answer this research question, we first investigate the popularity of target *versioned packages* at the library level. We use the output of *Sampler* component (see Section 4.2.3) to show the difference between the number of dependents that target *versioned packages* have. After this, we analyze the method-level popularity. We use the *Popularity Analyzer* as described in Section 4.2.8 to calculate a popularity score for every method within target *versioned packages*. We inspect these popularity scores in two groups. Firstly, the popularity scores for the majority of the public methods indicate zero usage by other libraries. Hence, we first look into the ratio of unused methods compared to the total number of public methods that *versioned packages* define. Secondly, we analyze the methods that are used by other libraries at least once and show how their popularity scores are distributed.

To fully comprehend the distribution of library-method popularity, one must recognize that not all libraries have the same number of users. Figure 4.11 shows the number of dependents that selected *versioned packages* have. As shown in this figure a limited number of *versioned packages* are used by many dependents but most of the *versioned packages* have less than 100 dependents. More specifically, only 12(3%) of the selected *versioned packages* have more than 100 dependents while 310(80%) of them have less than 20 dependents. The distribution follows the *Pareto principle*, which states that roughly 80% of consequences

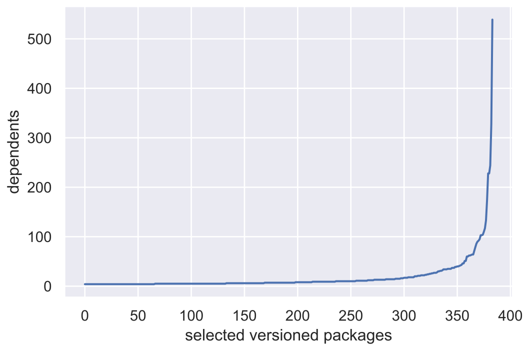


Figure 4.11: Distribution of Number of Dependents for selected *versioned packages*

come from 20% of the causes [189].

There also exist public methods within libraries that are not used by others. To this end, we first look into the extent of public methods that are not used by other libraries. Figure 4.12 shows that the majority of publicly defined methods are not used by any other library. On average *versioned packages* that we selected for this study define 1,688 public methods. However, the median of public methods for this *versioned packages* is 386 which indicates that some outlier projects skew the average. That is, we have a *versioned package* with 32k publicly defined methods as an outlier that affects the average. Additionally, there are some *versioned packages* that primarily concentrate on interfaces and define only a few public methods that can be called externally. In fact, we have 13 *versioned packages* with less than 10 publicly defined methods. Moreover, the second violin plot shows the distribution of publicly defined methods that are not called by any other library. On average 1.4k unused public methods are defined by selected *versioned packages*. However, similar to the number of methods this average is skewed by the outliers such as the project with the maximum number of unused methods with 31.6k unused methods. Therefore, the median of the unused methods is 386. There are also 14 *versioned packages* with less than 10 unused public methods. Finally, on average, the ratio of public methods that are not used by other libraries is 87% (median of 92%). The minimum ratio is 0.39 and the maximum ratio is 1. That is, 1) maximum coverage of public methods is 39% among all analyzed libraries, and 2) there exist libraries within selected packages that non of their publicly defined methods are used by others, which is because of their interface-based nature as our manual inspection revealed.

To look more closely at the used parts of these libraries we calculate the dependent usage ratio metric (see Section 4.2.8) for each method in the selected libraries that is used at least once by others. As seen in Figure 4.13, the dependent ratio follows a logarithmic distribution. The x-axis of this plot shows the public methods of libraries. Since libraries have a different number of methods we normalize this axis by using the Quintiles when sorted by the value of the y-axis. The y-axis is the ratio of dependents calling the method. Each dot in this plot shows the mean of all selected *versioned packages* in that particular x value. The orange line shows the trend that the mean of all selected *versioned packages*

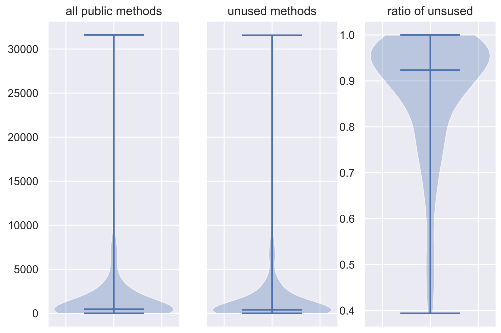


Figure 4.12: Unused public methods of selected *versioned packages*

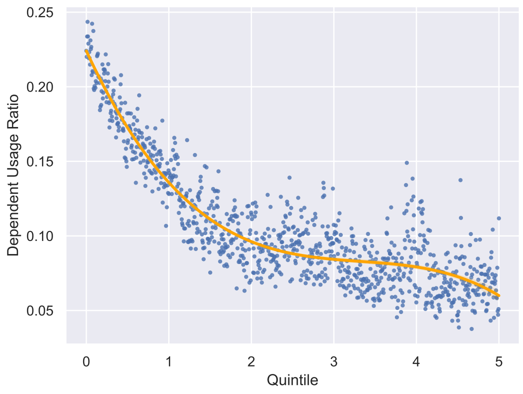


Figure 4.13: Popularity Distribution for Dependent Usage Percentage

follow. This line is calculated by a regression model that we fit on the aforementioned dots. Hence, this figure shows that on the current state of Maven on average there are only a limited number of methods in each library that are widely used.

The statement regarding the Pareto Principle also relates to the conclusions made by Harrand et al. [34], which notes that most clients depend on a small fraction of an API. From our research, we can come to a similar conclusion; popularity skews towards the most used methods, but comparatively, more methods exist with low popularity values. More specifically, the area under the orange curve in Figure 4.13 is 0.53. A coordinate with x=1.76 and y=0.1 is a point where the left area and right area under the curve are almost equal. This means that on average 35% of the methods in Maven libraries are receiving 50% of all calls from dependents. The remaining majority (65% of the public methods) receive the other half of the unique dependent calls. Note that these are only about the 13% of the methods that are at least called once by another library.

The majority of public methods that libraries define are not used by others. On average, 87% of publicly defined methods are never used by other libraries, and 35% of the remaining 13%, cover half of the dependent calls.

## 4.5 RQ3: Is there a relation between popularity and breaking changes?

In the next step, we want to investigate the extent of the problem from the users' point of view by connecting popularity information and breaking changes. Deletion of an unpopular method does not nearly have as many consequences as the deletion of a popular method. We aim to understand the relationship between popularity and breaking changes, this would reveal if the maintainers of the libraries are already aware of their users and try not to break them.

We investigate this research question in two levels similar to previous questions, i.e., library and method levels. For the library-level investigation, we first get the library popularity scores for the target *versioned packages* from *Popularity Analyzer* as described in Section 4.2.8. We want to inspect if there is any relation between these popularity scores and semantic versioning violations. So we also retrieve the violations of the target *versioned packages* from *SemVer Analyzer* as described in Section 4.2.6. We then calculate the ratio of the number of packages with violations to their corresponding popularity scores. By fitting a second-order regression model on the popularity scores and their corresponding violation ratio, we inspect whether or not a trend exists. Next, we use the *Popularity Analyzer* to obtain method-level popularity scores for target *versioned packages.* We investigate the difference between the population curve of these scores for methods that are involved in breaking changes and those that are not.

To explore the relationship between popularity and breaking changes we first analyze the popularity of the libraries. As described in Section 4.2.8 we measure what portion of the dependents are attracted to each target *versioned package.* Then we connect this information to the breaking changes to understand whether or not the popularity of libraries has any effects on making breaking changes. As shown in Figure 4.14 there exists a slight tendency for more popular *versioned packages* to introduce more breaking changes. This can be due to the higher number of change requests that popular libraries receive from their users. Note that the overall number of unique dependents is 7,190. Therefore, when we divide the number of dependents of each *versioned package* by such a big number, the maximum popularity value lies around 0.04.

To look closer at the popularity of breaking changes we inspect the relation between method popularity and breaking changes. Figure 4.15 shows the population of the popularity of methods that are not involved in breaking changes compared to the popularity of methods involved in breaking changes. The popularity is represented by the percentage of dependents that call the method. For example, when the x-axis is equal to one, the method is called by every dependent of the *versioned package.* The orange line in the figure shows the density of the population of the methods that are defined in our target *versioned packages* and are not identified as a breaking change. The blue curve, however, is the methods of the same *versioned packages* that are involved in the breaking changes. Methods with zero popularity are filtered from this figure since they make the rest of the

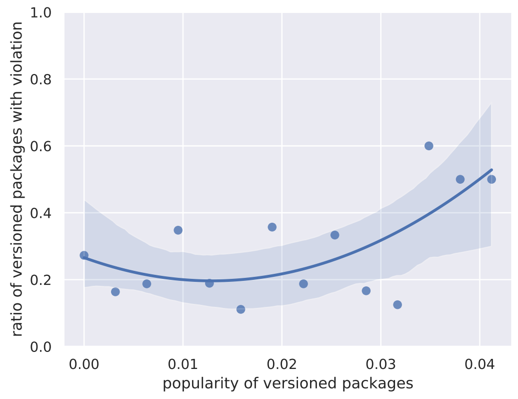


Figure 4.14: Semantic versioning violations and popularity of the *versioned packages*

figure invisible due to their high number. More specifically, 61k methods that were not used by any other library were filtered from the orange curve and 2.2k were eliminated from the blue curve.

In Figure 4.15, we can compare the difference between two populations by investigating the difference in the density at a certain point of the x-axis. If the curve of breaking changes has a higher density than the curve without breaking changes on a certain point $x$, this means that on average, a method with a breaking change more often has popularity $x$ than a method without a breaking change. However, in this figure, it can be noted that both lines follow a very similar distribution. Whether or not the method is involved in a breaking change does not seem to be related to its popularity. To prove this, we performed a *T-test* on both populations and observed that there is no significant difference between them. The results of this test show a *p-value=0.48* for a *two-sided* independent samples, showing that these populations do not have a significant difference.

One observation that we can also make from this figure is that the orange curve contains several spikes. Our manual inspection shows that the popularity values of methods can be concentrated on the same numbers. This means many methods are assigned with similar popularity values. Since *versioned packages* have a limited number of dependents and there are not many permutation options for method usage, these numbers can be similar. For example, assume a *versioned package v* has 10 dependents and 20 methods. If 4 of these methods are related to one functionality and are usually used together, when five of the dependents only use this functionality, the popularity value will be 0.5 for all four methods. The popularity of the remaining 16 methods can also be 0.5 if they are used by five of the dependents. Hence these spikes can be observed when creating the population curve for all data points. However, despite these spikes, the overall distributions are similar as validated by the *T-test*.

The popularity of a method does not play a role in whether or not it is involved in breaking changes, no significant difference exists between the popularity of the methods that break the semantic versioning and the ones that do not.

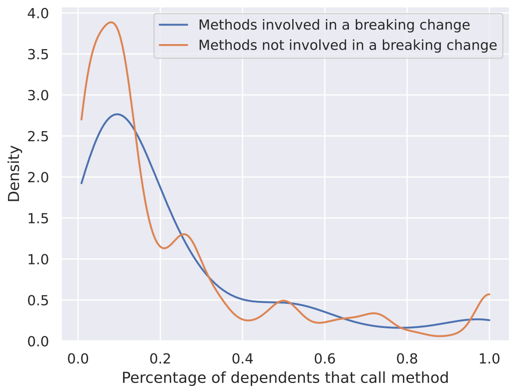


Figure 4.15: Distribution of percentage of dependents that call methods

## 4.6 Discussion

We found that a large number of *versioned packages* contain breaking changes. Users run into issues when they unsuspectingly attempt to upgrade their dependencies, as the API is no longer compatible with their artifact. This happens even though the versioning convention of the library promises compatibility. Furthermore, illegal API extensions do not pose a threat to the compatibility of users. Therefore, it might seem intuitive that maintainers can be less concerned with these illegal API extensions. However, the maintainers that extend their API illegally are not only employing bad practices but also increasing the chances of having more breaking changes and violations in the future. This is because having more methods in a library increases the chance of violations as we showed in Section 4.3. By doing this study, we found that maintainers on one hand need to pay more attention to semantic versioning and any type of library API expansion as a preventive measure to decrease the chance of semantic versioning violations. Researchers on the other hand can continue our work by investigating what type of tooling, design decisions, and development methodologies lead to more stable APIs.

We did not find a meaningful relationship between the popularity of a method and whether it is involved in a breaking change or not. However, there might be some factors at stake. For example, on one hand, more popular methods may get involved in more issues by the users. This can potentially help maintainers understand which methods are more popular. On the other hand, attracting more attention to the issues naturally increases the chance of changes in the methods and consequently more involvement in breaking changes. Similarly, in the case of unpopular methods, maintainers may realize that not as many people open issues related to a particular method. Hence they are not very popular. Maintainers may change such methods more easily since they assume they would not affect many people. We speculate that there are many similar factors affecting the behaviors of maintainers in case of breaking changes that are worth understanding for the software community. Hence, we encourage researchers to conduct more research to understand such aspects. Our results emphasize the need for such studies by showing that breaking changes are still a big problem for members of the ecosystems. However, in this study, we only focused on finding the breaking change usages in the ecosystem. It is also important to understand the collective cost that breaking changes impose on the ecosystem members in terms of time and money since the cost may differ from one dependent project to another. We encourage researchers to reuse some parts of our approach as a starting point to study these collective costs.

We speculate that library maintainers do not have sufficient data to accurately differentiate between popular and unpopular methods. They can use this information before introducing breaking changes and during their maintenance. An established popularity ranking of the usage of their own methods across the whole ecosystem would allow library maintainers to improve their workflow. Utilizing such a popularity ranking, one could prioritize a list of issues to work on based on their importance within the ecosystem; a change to a popular method will have more impact compared to an unpopular one. During this study, we realized that there are numerous opportunities for client-side and platform-based tools. IDEs can recommend maintainers to adjust their version appropriately whenever a change in method signatures is detected. Build tools can offer warnings when finding an inappropriate version upgrade. GitHub can warn users when they break a popular method in their *pull requests*. Maven Central can warn developers about incompatibilities, impose strict requirements for versioning of artifacts, or give a high score to projects without any semantic versioning violations. To create a safer and more trustworthy environment within the ecosystem researchers and engineers can focus on building such tools in the future as we indicated the need in this study.

Method popularity also introduces opportunities for the users of the libraries. Method popularity might be used as a recommender system that supports developers during the coding activity. Currently, library popularity in Maven helps users determine which library to use. However, users can also pay attention to the popularity of the methods that they need or compare the popularity of methods with similar functionalities. Another use case that is worth further research is integrating method popularity ranking within auto-complete engines to improve their recommendations.

In this study, we only focused on public methods as the main mean of library reuse. While it is possible for the developers to use package private and protected library methods, such methods may not as often be intended for API usage. Maintainers intentionally expose the public methods for the reuse of their library. Hence, we limit our scope to the intended and most common modifier for library reuse. We also limited the scope of violations to the defined methods since the CGs only add edges to such methods. We ran our popularity analysis experiment on all methods and we observed minimal difference in the overall results. We believe a more detailed investigation in this regard is out of this study's scope. Future research is needed to investigate this in detail. We realized that despite the existing studies there is room for further research. For example what parts of code are more likely to break or whether public methods break more often than protected methods. Answering these questions benefits the community to understand what parts of the code they should pay more attention to.

### 4.6.1 Threats to Validity

In this study, we limit our dependent resolution to a recent six months while multiple decades of evolution are available on Maven Central. We believe our dataset is already extensive and that a larger study would substantially increase the cost of execution, while there is no reasonable intuition to expect that the insights would differ for a bigger period. A more extended period surely increases the number of *versioned packages* and their dependents. However, we do not expect the overall usage pattern of *versioned packages* to change.

Generating a perfect static CG is an undecidable problem. Hence existing solutions use over-approximation and sacrifice precision for scalability. Our illegal API extension extraction also uses approximation. An AST-level solution is needed for better precision which was not practical for our scale. Although these design decisions may impact our precision, we opted to prioritize scalability, which enabled us to offer a more comprehensive perspective of the ecosystem. Nevertheless, our findings entirely align with previous research which shows the effects are minor.

Maven Central does not only contain Java artifacts; it also contains artifacts that are implemented in other JVM-based languages. In some cases, we had difficulty generating CGs for such *versioned packages*, and thus the artifacts were not included in the analysis. However, we found that no more than 1% of analyzed artifacts were Scala or Kotlin packages. Thus, we believe it is safe to only focus on Java projects in this study.

During this study, we developed several programs and used open-source libraries which may contain implementation errors. One can never ensure all implementations are bug-free. However, we tried to mitigate this by performing manual tests and code reviews as well as releasing our code and executable to the public.[3]

## 4.7 Summary

In this study, we showed that a large number of Maven libraries do not completely adhere to semantic versioning. We know this, as 63% of analyzed artifacts break compatibility within their major versions. Illegal API extensions also occur in 54% of artifacts. Therefore, these violations form a big problem for the trustworthiness of semantic versioning on Maven. Moreover, deletion or alteration of an unpopular method does not have the same impact as changing a popular method. We investigated whether a relationship between method popularity and involvement in breaking changes exists. We discovered that breaking changes occur in popular methods as frequently as in unpopular ones. By analyzing all interactions between software artifacts and their dependents we found that the majority (87%) of publicly defined methods are not used by others and 35% of the remaining are responsible for half of the dependent calls. Similarly, the number of dependents per library also follows a power law distribution.

[3]https://github.com/ashkboos/semver-vs-popularity

# 5 Maven Unzipped: Exploring the Impact of Library Packaging on the Ecosystem

Maven is a popular dependency management tool and ecosystem used by millions of developers. However, the overwhelming amount of available open-source software and the lack of proper ecosystem governance pose risks to the security and effectiveness of the ecosystem. This necessitates a comprehensive understanding of the ecosystem to guide future decision-making and promote effective practices. Despite numerous studies on aspects of Maven, such as vulnerabilities, breaking changes, and bloated dependencies, a knowledge gap concerning its overall state and health still exists. This gap impedes the adoption of effective practices, potentially impacting the productivity and efficiency of projects and the ecosystem as a whole. This paper explores the fundamental aspects of the Maven ecosystem. We investigate the packaging practices of Maven libraries with a focus on the content of the libraries, their impact on the ecosystem and each other, and their evolution over time. Our goal is to provide insights into the ecosystem's practices and trends. To achieve this, we create a scalable infrastructure and collect a comprehensive dataset of 480K unique packages by randomly selecting one version from each Maven project. We use this dataset to analyze the content of Maven releases and their packaging practices. We discover three concerning practices that deserve the community's attention: various data inconsistencies within Maven, improper use of Maven archives, and exponential dependency growth. We discuss practical recommendations to mitigate these issues, such as implementing stricter release checks and dependency minimization during deployments. To help promote more research, we open our dataset and tools for public use.[1]

[1]This chapter is based on the following paper: Mehdi Keshani, Gideon Bot, Priyam Rungta, Maliheh Izadi, Arie Van Deursen, and Sebastian Proksch: Maven Unzipped: Exploring the Impact of Library Packaging on the Ecosystem. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2024.

Maven is the most relevant software ecosystem for Java development and represents a vibrant, central hub for managing, storing, and sharing reusable open-source software libraries. It has also grown substantially over the last few years [190]. Recent reports have shown that 96% of proprietary software uses open-source [191]. This highlights the importance of ecosystems like Maven in the entire software industry with a revenue of $699B in 2024 [192].

Even politics identified the role that such ecosystems play in innovation and value generation in the economy and the risk of insecure supply chains: The US executive order #14028 was issued to "Improve the Nation's Cyber Security" [12], and the *EU Cyber Resilience Act* [193] aims in the same direction. As such, many studies investigate various aspects of software ecosystems, such as Maven. These studies include examining the effects of vulnerabilities [40–43, 108, 194], breaking changes [28, 32, 79], and bloated dependencies [29]. However, it seems that the unprincipled and prolific use of public repositories has navigated the industry into a challenging situation: the explosively growing amount of available open-source software makes it hard to keep a holistic view of the ecosystem and maintain its health.

Previous works studied dependency practices in software ecosystems like NPM [22–26] and PyPi [27], focusing on the developer's perspective. However, considering the entire ecosystem is also crucial. Some practices are harmless in isolated instances, but they compound and become problematic on an ecosystem scale. We postulate that the governance of ecosystems can be improved only if a comprehensive understanding of their contents and evolution is available. This knowledge can inform future decisions within the software development community and catalyze the development of effective software practices and tools. In this paper, we explore the Maven ecosystem to identify and quantify recurring patterns in the ecosystem that yield negative effects. We refer to these issues as *ecosystem bad practices*.

We aim to explore the software ecosystem as a whole. The term *ecosystem* originates from biology, where it refers to *"A biological system composed of all the organisms found in a particular physical environment, interacting with it and with each other"* [195]. To understand an ecosystem, we need to determine which organisms live within it and their interactions with their ecosystem and each other. In the Maven context, projects resemble organisms, and the repository serves as their environment. These organisms interact with the ecosystem by hosting their files in it and with each other through dependency relations. We explore these fundamental aspects of Maven by asking the following Research Questions (RQs):

**RQ$_1$** What type of files are being released in Maven?

**RQ$_2$** What are Maven packages' storage requirements?

**RQ$_3$** What factors contribute to larger libraries?

**RQ$_4$** How has the size of dependency sets evolved over time?

We focus solely on Maven Central, the largest repository of Maven, which, as of the time of writing, is at least five times larger than any other Maven repository [190]. While answering the RQs, we look for recurring patterns that are harmful to the ecosystem.

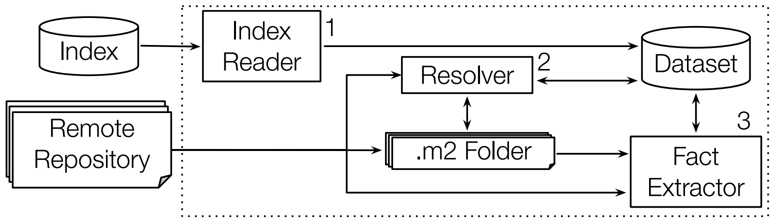


Figure 5.1: Overview of dataset creation

Sonatype [196], the maintainer of this repository, ensures a minimum quality for uploaded artifacts by enforcing mandatory requirements [197]. We aim to explore opportunities for enhancing such quality assurance measures. Throughout the paper, we will use the term Maven interchangeably to refer both to the Central repository and the build tool.

To answer our RQs, we establish a scalable infrastructure and collect a comprehensive dataset of Maven packages. From *all* available Maven projects (group and artifact), we select one random version to avoid bias towards very active projects. We chose not to pick the newest version of each project, to cover a wider period. The result is a dataset of 480K unique Maven releases, which we use to answer our RQs.

Our analysis results reveal three issues within Maven that need the community's attention: 1) the prevalence of missing and inconsistent data in the ecosystem, 2) improper use of archives, resulting in a rising trend in the average size of Maven libraries, and 3) exponential growth in the number of dependencies, which leads to a tenfold increase in the space needed for transitive dependency sets over the last decade. We coin these three issues *Erratic Data*, *Archive Misuse*, and *Transitive Growth*, respectively. These issues negatively impact the ecosystem's maintainability, and security. Addressing these challenges through enhanced tooling, stricter guidelines, and further research will be crucial to sustaining and improving the health of the Maven ecosystem. We discuss the implications of these issues on the ecosystem and provide actionable recommendations for different stakeholders to mitigate them. These recommendations include enforcing rigorous consistency checks during deployment, separating versioning and management of data and code, as well as monitoring and minimizing project dependencies. In summary, this paper presents the following main contributions:

1. An infrastructure to mine Maven packaging data.
2. A study of the structure, size, and evolution of packages.
3. Identification of three ecosystem bad practices.
4. Discussion on the implications of the identified bad practices, along with actionable insights and practical solutions.
5. Publicly available code, and dataset [198].

## 5.1 Dataset Creation

To answer the RQs, we create a dataset by analyzing public resources like the Maven repository. This section outlines our approach and the technologies used to implement our pipeline.

Table 5.1: Number of versions per groupId:artifactId

| Min | Avg | Median | Std | Max |
|---|---|---|---|---|
| 1 | 21.5 | 5 | 81 | 4631 |

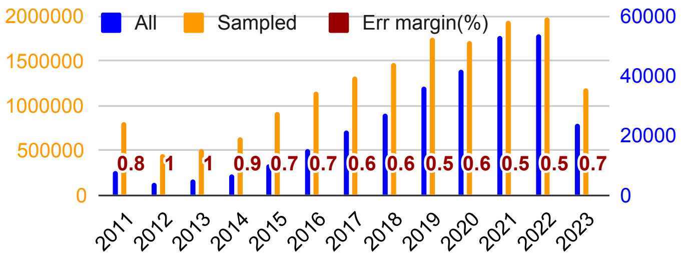


Figure 5.2: Maven vs our sample clustered by release year.

**Infrastructure** The data from each Maven release is independent, so we designed and implemented an extensible infrastructure that extracts data in parallel for each package, only limited by available computing resources. We utilized a powerful server with 256 cores that took 3.5 hours to populate the whole dataset and another 6 hours to resolve dependencies. The infrastructure can be extended through custom analyzers that add additional information to the dataset. We employ a reusable execution environment that eases reproducing our dataset from scratch or updating it periodically with new packages. We create a docker-compose environment that minimizes environment dependencies and executes all experiments with a single command. Ultimately, a Postgres database can be queried to generate insights about the ecosystem.

**Dataset** We use the methodology in Figure 5.1 to generate our dataset and answer RQs. Maven Central maintains a weekly-updated index file [88] cataloging all new releases, with metadata like the last modification date. We process indices from May 17, 2023 using the *Index Reader* as shown in Figure 5.1. As of this writing, Maven Central hosts 10.3M releases of 480K distinct packages. Table 5.1 presents summary statistics for the number of versions per package. Notice that the standard deviation is high, which biases the average, making the median a more appropriate metric. Each release is uniquely identified by a triplet in the format `groupId:artifactId:version`. Our goal is to obtain a representative sample of Maven, which is feasible to process. Analyzing all projects (instead of sampling projects) is most generalizable. Different versions of the same project may share similarities, but different projects represent independent data points and diversify the results. Therefore, we insert a randomly selected version from each package (`groupId:artifactId`) into our *dataset* and keep the identifiers as the key. This sampling strategy ensures our results are not skewed towards packages with higher release frequencies.

Figure 5.2 presents the distribution of releases on Maven Central and our sampled data, showing a significant increase in the number of libraries released within the ecosystem over time. It also shows that our sampled data mirrors the overall population of Maven releases over time. The margin of error for each year, varying from 0.5 to 1, is shown in this figure for a confidence level of 99%. This indicates that we can draw conclusions about the history of Maven Central. Importantly, selecting only the last or first versions of libraries does not yield such a distribution, thus preventing historical analyses. The total number of sampled packages is statistically representative of the entire Maven Central population with a confidence level of 99% and a margin of error of 0.18%, allowing us to draw conclusions about the entire ecosystem.

All libraries, their files, and directory structures are stored in the local `.m2` Maven repository. The *Resolver* component allows querying the dataset. On request, the *Resolver* inspects this folder to find a queried library. If it does not exist, the *Resolver* downloads it from the remote repository, resolves the dependencies, and records them in the *dataset*.

Our main components then start the data collection process. The *Fact Extractor* first reads the list of selected libraries from the *dataset*. For each artifact, it aggregates information like file details and sizes from either the local `.m2` folder or the remote repository, depending on where the required data is available.

Overall, our dataset contains 480K releases. Among these, 6,419 (1.3%) packages were unresolvable, due to corrupted POMs (Project Object Model) or non-available parent POMs. As the *Resolver* is unable to download these packages into the `.m2` folder, they are excluded from the results. Another 63,250 packages do not have an archive file, for example, parent POMs that define dependencies or configure settings for child artifacts. We exclude these packages as well.

## 5.2 What type of files are being released?

### 5.2.1 Packaging types

The most crucial element in libraries is their archive, which enables code reuse. In addition to their `groupId:artifactId:version`, each package has a *packaging type*. Most files are packaged in compressed `jar` files, which contain multiple `class` files, metadata, and resources. It serves as a specialized `zip` file for distributing Java libraries. Our study extends beyond `jar` files to explore types such as `ear`, `war`, and various generic types, each with its own unique use case [199].

**Collecting data** To derive packaging types, we utilize three sources: the Maven index, the package's pom.xml file, and the Maven repository itself. First, when we read the Maven index using *Index Reader* we identify the executable (archive) artifact and record its packaging type. Second, we utilize the package's pom.xml file previously downloaded by *Resolver*. A POM file is a configuration file utilized by Maven to manage projects. The POM file instructs Maven on how to build the project and specifies the dependencies. To facilitate the retrieval of information from the pom.xml, we leverage Maven's built-in model [99]. This model provides us with a convenient framework for accessing and extracting the required data. We perform this extraction in the *Fact Extractor* component. Then, we interact directly with the Maven repository, which acts as our third source of data. We send a request for each package to obtain the list of files linked to that particular package in the repository. By examining this list, which also includes the executable file, we determine packaging types. We extract the executable file's extension from a file name in the format of `artifactId-version.extension`. We then compare these three sources to determine their agreement level.

Table 5.2: Packaging types found in three sources

| POM | % | Index | % | Repo | % |
|---|---|---|---|---|---|
| jar | 75.0 | jar | 80.8 | jar | 80.0 |
| pom | 12.0 | pom | 11.3 | pom | 11.0 |
| bundle | 4.6 | aar | 2.3 | aar | 2.4 |
| aar | 2.4 | module | 1.7 | module | 1.7 |
| maven-plugin | 1.3 | zip | 1.3 | zip | 1.2 |
| war | 1.2 | war | 1.2 | war | 1.2 |
| Others <1.0 | 3.5 | Others <1.0 | 1.4 | Others <1.0 | 2.4 |

**Findings** In our analysis of POM files, we identified 181 distinct packaging types, with `jar` being the most prevalent, accounting for 75% of all types. Excluding the top 6 packaging types, the rest constitute only 3.5% of the total. Table 5.2 shows the distribution of the top 6 most frequent packaging types from our three sources. Regarding the packaging types from the Maven Index, we identified 110 distinct types, with `jar` being the dominant type, constituting approximately 80.8% of all types found in the Index. Moreover, only 1.4% of packages fall outside the top 6 most prevalent types. From the Maven Central Repository itself, we identified 137 distinct types, with `jar` again being the most prevalent, accounting for approximately 80% of the total distribution. Excluding the top 6 packaging types, only 2.4% of packages fall into alternative categories. All sources also agree on the second most common packaging type, pom, prevalent due to the abundance of parent projects on Maven that provide configurations for their child projects without having an executable archive.

Some packages in the Maven repository have multiple archive files. A manual inspection of 40 of such packages, revealed that alternative archives, despite differing extensions, typically contained identical content to the archive specified in the POM file or the Maven index. 97% of packages only have a single type of packaging. Around 3%, incorporate two types. Having three or more types is very rare (<0.001%).

Ideally, the packaging type in the POM should dictate the project's packaging and distribution, with all three sources in agreement. However, we found discrepancies. The packaging types in the POM and index differ for 9.2% of packages, showing 293 unique discrepancy pairs. The most common disparity (48.7%) occurs when the POM specifies a `bundle` but the index lists it as a `jar`. The `bundle` type, used in Open Service Gateway Initiative (OSGi) framework projects, is a `jar` file complying with OSGi specifications, containing additional metadata and Manifest files. Maven produces OSGi-compliant `jar` files during the build process, leading to a `jar` extension. Besides `bundle` to `jar`, Maven inherently converts certain packaging types into `jar` during the library publishing process, as its default packaging value is `jar` [200]. Additionally, 3.9% of packages show different packaging types in the Maven repository compared to the Index due to the Maven indexer capturing only one packaging type per package, though developers sometimes upload more than one type for a package. Furthermore, a 12.3% discrepancy exists between the packaging types in the POM and the repository. In a manual inspection of 30 such packages, we found that the packaging type in the index might not be the library's primary archive. For example, we observed a case where the main `jar`, containing typical binary files, exists in the repository, but the index lists an archive that includes the `jar` with dependencies along with other resources, all zipped into a single file [201].

Table 5.3: Checksums combinations and their percentage

| | Checksum Combinations | % |
|---|---|---|
| **1** | MD5, SHA-1 | 98.5 |
| **2** | MD5, SHA-256, SHA-512, SHA-1 | 1.4 |
| **3** | None | <0.1 |
| **4** | Various subsets of (2) combined | < 0.1 |

## 5.2.2 Checksum files

The generation of *checksums* (or *hashes*) plays a pivotal role in the Maven ecosystem by ensuring the integrity of artifacts and associated metadata. These *checksums* validate the correctness of downloaded artifacts by comparing the computed checksum with the expected value. For example, a file `a-1.jar.md5` contains the checksum for the corresponding file `a-1.jar`. The `.md5` extension indicates the MD5 algorithm was used to generate the checksum. Maven supports multiple checksums per uploaded file, and we undertake a comprehensive analysis of these checksums next.

**Collecting data** We retrieve checksum information from the list of files uploaded to the Maven repository using the request method discussed in Section 5.2.1. We rely exclusively on this source for its reliability and comprehensiveness, as the `.m2` folder and index file alone might not have all files available remotely. We store the `checksum` component in the dataset for artifact names conforming to the specified pattern of `filename.checksum`.

**Findings** Table 5.3 shows the checksums found in our dataset, with an overwhelming majority, 99.9%, of packages employing both MD5 and SHA-1 checksums, aligning with Maven's default configuration. In 98.4% of the packages, MD5 and SHA-1 are used exclusively. Another 1.4% of packages additionally employ the more robust SHA-256 and SHA-512 checksums, attributable to maintainers facilitating a transition to stronger algorithms while ensuring compatibility with systems reliant on weaker ones.

Alarmingly, a small subset of packages (~0.07%), does not incorporate any form of checksum, and another small subset of packages (<0.07%) is lacking some checksums. This has likely been caused by failed artifact uploads and would have been easy to detect in the release process. However, the non-availability now makes it impossible for users to validate their downloads, which can have a major impact, depending on the number of package users. Upon manual inspection of 50 such cases, we found that 17 of them have other dependent libraries on Maven Central. This number represents only the direct usages reported by Maven. However, it is important to note that these usages also propagate transitively, affecting even more packages. Moreover, previous studies indicate that the majority of packages either have no users or very few users [79]. Therefore, it is expected to find many packages without any usage.

To gain a deeper understanding, we analyze the temporal evolution of two prevalent checksum combinations. Figure 5.3 shows the annual ratio of releases employing these popular checksum combinations, revealing a trend of gradual adoption of stronger checksum algorithms starting in 2014, peaking in 2021, and then quickly declining from 2022 onwards. Naturally, there is an inverse relationship between the usage of the two

Table 5.4: Classifiers and their percentage

| **Classifier** | **%** |
|---|---|
| sources | 82.3 |
| javadoc | 76.8 |
| tests | 5.9 |
| test-sources | 2.3 |
| source-release | 0.6 |
| jar-with-dependencies | 0.6 |

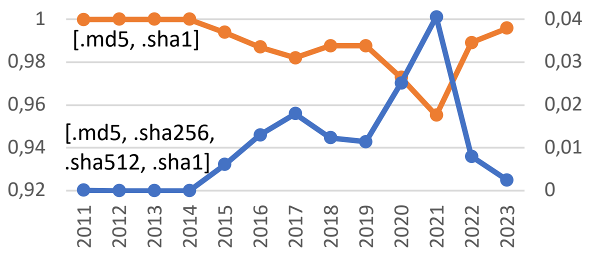


Figure 5.3: Checksum ratio over time. Left axis for MD5,SHA-1

combinations: as the adoption of stronger algorithms increases, that of weaker algorithms decreases, and vice versa. In 2021, the adoption of strong algorithms peaked at 0.04, but over the last two years, the trend has reversed to its earlier levels. This decline correlates with a discussion on Maven's mailing list [202] at the end of 2021. The conversation, starting with a call for more secure algorithms, revealed that the use of checksums in Maven confuses some users. It was clarified that there was no need for better algorithms in Maven, as they serve only as integrity checks for downloads, without offering security benefits. Ultimately, the overhead of such algorithms is not justified in Maven's context. Basic algorithms suffice for integrity validation, which has been explicitly mentioned in the Maven checksum documentation [203] since. This observation emphasizes the value of discussions, interventions, and updated documentation for an ecosystem.

### 5.2.3 Additional files

In addition to archive files containing binary content of libraries, Maven also hosts documents and source codes. A classifier acts as an optional attribute distinguishing artifacts sharing the same group, artifact ID, and version. For example, the presence of `a-1-sources.jar` indicates that the `jar` file contains source code files, such as Java source files. In our investigation, we explore the common usage of classifiers within Maven Central as our next aspect.

**Collecting data** To gather all classifier artifacts associated with a specific package, we utilized the same approach as for checksum retrieval. The pattern for these artifacts is in the format `artifactId-version-classifier.extension`. Among artifact names adhering to this pattern, we store the `classifier` component in the dataset as a supplementary file with the package.

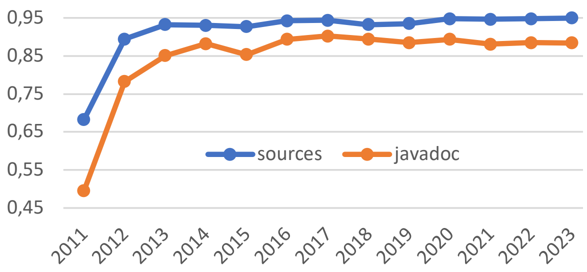


Figure 5.4: Ratio of packages with sources/javadoc over years

**Findings** We identified a total of 2,673 distinct classifiers in the Maven Central repository, underscoring the repository's diversity of classifiers. Notably, a significant proportion of them, 46.3% and 43.2%, are `sources` and `javadoc`, respectively. Table 5.4 displays the top 6 classifiers in Maven Central, along with the percentage of packages that employ them. The `sources` classifier emerges as the most prevalent, found in 82.3% of the packages. The second most common classifier, `javadoc`, is included in 76.8% of the packages. These numbers indicate an emphasis on code transparency and documentation within the ecosystem. This represents a notable improvement compared to Raemaekers et al.'s findings in 2013 [28], where only 68.4% of jar libraries had source files and 53.1% had Javadoc files. The `tests` and `test-sources` classifiers are found in a relatively smaller proportion of packages, 8.2% when combined. Furthermore, 0.6% of the packages include their dependencies in their releases using the `jar-with-dependencies` classifier.

The wide availability of `sources` and `javadoc` classifiers is unsurprising, as they are commonly consumed by modern development environments to enhance usability and transparency for developers. The numbers reported in this section only include projects with archives though, which means that ~23.2% of releases do not feature `javadoc` and ~17.7% are lacking sources. To understand the impact of this situation on the ecosystem, we examine the evolution of the `javadoc` and `sources` classifiers over time.

Figure 5.4 demonstrates the evolution of the usage of the `sources` and `javadoc` classifiers in Maven. We exclude the packages with pom packaging from this figure since the Maven documentation mentions that the `sources` and `javadoc` classifiers are not mandatory for such releases [197]. In 2011, the proportion of packages with the `sources` classifier was relatively low, at ~68%, but it significantly increased to 89.4% by 2012. We speculate that this sudden increase might be due to the introduction of requirements for `javadoc` and `sources`. However, unfortunately, the history of these requirements is not transparent, which prevents us from confirming this. From 2012 onwards, we observe a gradual increase, peaking at ~95% in 2023. The `javadoc` classifier follows a similar trajectory, albeit with lower overall percentages. It started at 49.5% in 2011 and experienced a substantial increase to 78.3% in 2012. In the subsequent years, it witnessed a steady climb, reaching a peak at ~90.3% in 2017. Following this, there were mild fluctuations, with percentages generally hovering ~88.0%. While we observe progress over time, the growth rate appears to have slowed down, suggesting a plateauing trend, especially in the past few years. It is important to note that there is still considerable room for improvement, particularly in the inclusion of the `javadoc`.

### 5.2.4 Inside executables

The content of the executable itself is the most crucial component of libraries, as it forms the foundation upon which clients build their projects. Consequently, as our fourth aspect, we aim to identify the specific types or categories of files that comprise the executables distributed within Maven libraries.

**Collecting data** To retrieve all files within the executable artifact, we begin by decompressing the executable artifact. This step is performed for all packages except POM packages, which do not contain any executable files and, therefore, no binary content to investigate. We parse the names of all entries within the extracted artifact, collecting and storing their extensions in the dataset. Since Java 9, Java *Modules* help developers to encapsulate and organize related Java packages into discrete, manageable entities. By defining required packages and specifying exported ones in a *module-info.java file*, developers can use this feature. This is particularly relevant as this Java feature directly introduces an additional file to the content of executables. Java modules are identified by unique module descriptors. Each descriptor resides in a single file, named *module-info.class*, for every module. To determine if an archive uses Java modules, one can simply scan all its entries. The presence of any files named *module-info.class* confirms the use of Java modules.

**Findings** Our experiments uncovered more than 14K distinct file types, with `class` files representing ~22.1% of the total. About 65% of all files in the ecosystem do not belong to the 6 most commonly used file types, and 99.9% of the packages contain file types outside these top 6 categories. This emphasizes the ecosystem's broad diversity in file type usage. Table 5.5 shows the top file extensions within the archives and the percentage of packages incorporating these file types. `class` files constitute ~84.8M of the files in the entire ecosystem and are included in 71.2% of Maven libraries. Java projects typically contain numerous `class` files, each encapsulating different functionalities and components such as classes and inner classes. The artifact with the most `class` files has ~208K of them.

The second and third most frequent file extensions are `sjsir` and `js`, mainly used in Scala and JavaScript projects, respectively. These files are found in only around 3.2% and 6.8% of the packages. Interestingly, `png` files rank high, despite being included in just 3.6% (17K) of the releases. However, these libraries contain a significant number of `png` files, averaging approximately 70.4 per package, explaining their high ranking. Common extensions also include `mf` and `xml`. A manifest file (`mf`) contains archive metadata, present in about 78.6% of libraries. Over 1M `xml` files are used in approximately 61.2% of packages, primarily due to Maven projects' pom.xml files. Lastly, around 1.69% of analyzed packages utilize Java modules, but they are not included in Table 5.5 since they are rare and not among the top files.

These findings highlight the extensive diversity within Maven. This diversity highlights the necessity for developers of tools that analyze Maven artifacts to ensure broad compatibility and support across a wide array of technologies and languages.

Table 5.5: Number of occurrences of archived files and percentage of packages that include them.

| Packaged file | Count | %packages |
|---|---:|---:|
| `class` | 84.8M | 71.2 |
| `sjsir` | 2.8M | 3.2 |
| `js` | 1.8M | 6.8 |
| `png` | 1.2M | 3.6 |
| `xml` | 1M | 61.2 |
| `mf` | 380K | 78.6 |

25% of Maven libraries are not in the `jar` format. Almost all releases utilize MD5 and SHA-1 checksums. While the majority of Maven releases include `sources` and `javadoc`, many projects do not release these crucial artifacts. Lastly, the primary archive contents are `class` files and their corresponding files in Scala (`sjsr`) and JavaScript (`js`).

## 5.3 How much storage do packages need?

Library size is important. Bigger libraries often have more features, saving users from unnecessary duplication of effort. However, they also bring challenges. Bigger libraries are more likely to introduce more bugs [30], lead to breaking changes [79], and consume more disk space, memory, and download bandwidth. This can be an issue for users with limited resources. Such libraries can also expose more vulnerabilities and often have more dependencies, thus contributing to dependency bloat in projects. They are harder to manage, slower to load, and more complex. Yet, we do not have a thorough understanding of library sizes in software ecosystems, specifically in Maven Central. This section aims to address this gap.

**Collecting data** We use the *Fact Extractor* to extract details for each archive file, collecting information such as file count, archive file byte size, and various file extensions with their statistical measures within the archives. This provides a comprehensive understanding of library sizes.

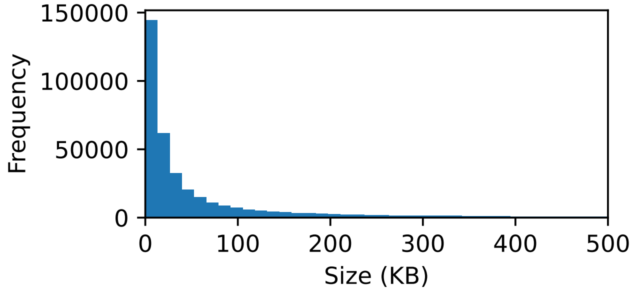


Figure 5.5: Distribution of size of artifacts in Maven Central. The x-axis has been cut off at 500KB for readability.

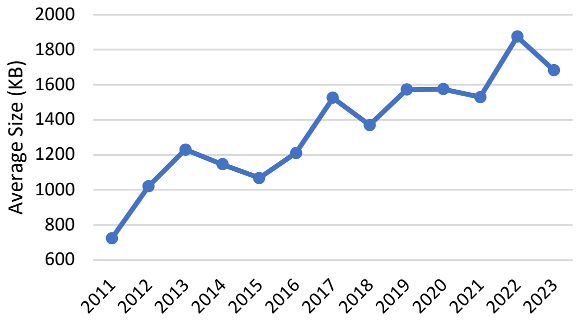


Figure 5.6: Average size of an artifact per year

**Findings** Figure 5.5 shows that the size distribution of Maven libraries has a long tail. Most libraries are small and the bigger the size, the fewer libraries fall into that category. We determined that the average size of a Maven library is ~1,447 KB and the median size is 25.9 KB. The large difference between the average and median suggests a significant variance in Maven library sizes. The majority of the packages are significantly smaller than the average size, though: 73% of libraries fall within the 0-100 KB range, and 86% use less than 400 KB of space. To improve readability, Figure 5.5 is truncated at 500 KB, which removes the biggest ~49K packages from the plot (~12%). Some packages occupy an enormous amount of space, the largest identified package has ~987 MB [204], which dominates the average values.

We observed that a minority of packages are substantially large and pull the average towards themselves. Now, we aim to understand how the average size changes over time. Figure 5.6 shows that the average size of a library has increased from ~723 KB in 2011 by more than 232% to 1,683 KB in 2023. Despite some observable fluctuations, the diagram reveals a clear overall trend: library sizes on Maven have been substantially increasing over time.

The size distribution of Maven artifacts is long-tailed. 73% of libraries are smaller than 100 KB. An average library size is 1.4 MB, while the median is 25.9 KB, indicating a significant variation. Over the last decade, the typical library doubled in size.

## 5.4 What factors contribute to larger libraries?

After understanding the space requirements of libraries, our goal is to explore the potential factors that contribute to larger libraries. Identifying these factors can help us pinpoint any specific patterns or practices that lead to unnecessary bloat in the package sizes.

**Collecting data** We examine correlations between library sizes and other characteristics, such as the number of files. We also manually inspect a variety of cases, such as libraries with a small number of files yet a large total archive size.

**Findings** Figure 5.7 displays the relationship between the size of the libraries and the number of files they contain. Two distinct categories of libraries emerge from this figure, forming visible trend lines in this figure at 0 degrees and at ~60 degrees. The former

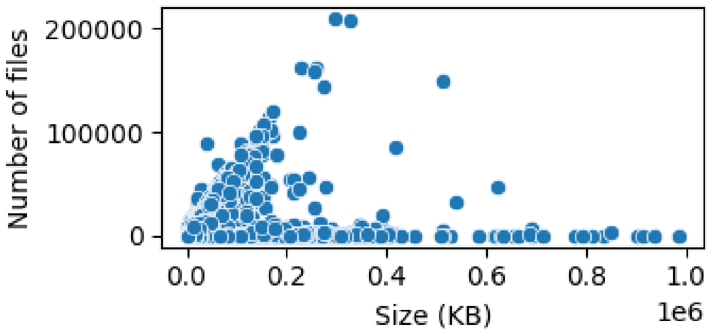


Figure 5.7: Number of files included in a `jar` and archive size

exhibits a slight increase in size as more files are added, which is the expected behavior. The latter, however, have few files but their sizes span from very small to very large as they contain some large files that contribute substantially to the overall library size.

We manually examine the top 10 largest packages that contain fewer than 200 files. We documented these libraries in our replication package. Most of these were data analysis and machine learning-related projects containing significantly large files (primarily data files), such as `.data`, `.csv`, `.db`, `.nt`, and `.rnn` (recurrent neural network). Some of these packages, which focus on natural language analysis, house many large `.ol` files detailing specific language rules (e.g., hyphenation rules for the Bulgarian language). We also observed one package containing three enormous `.so` files. Shared object (SO) files are dynamically linked libraries that encapsulate compiled code and data in Unix-like operating systems. By establishing links to these shared object files, programs can access functions, data, and other resources contained within the files without incorporating them directly into the archive.

The cases we have discussed so far are rather diverse. A single type of file can significantly increase the size of a particular library. However, such instances are relatively rare, as depicted in Figure 5.7. Each case involves different files with distinct functions, but they all share a common feature: they involve one or a few extremely large files. These types of libraries can affect both their direct and indirect users. However, another pattern collectively impacts many more libraries within the Maven ecosystem. This pattern arises when certain types of files, though not large individually, accumulate and contribute to the increased size of libraries. When many libraries incorporate these types of files, they collectively bloat libraries and the ecosystem as a whole, despite their small individual file size. A widespread use then affects many developers.

Table 5.6 displays the top 10 file extensions with the largest accumulated size on Maven Central. Each file extension may occur multiple times within a release, so we also include the average space occupied by each file extension in libraries that contain them. This analysis highlights the specific file types that significantly contribute to the overall space requirements of Maven. Interestingly, the most prevalent file type is `jar`, accumulating to 308 GB across our dataset. It's important to note that these figures only account for files packaged inside the archives. These `jar` files are likely repackaged dependencies that developers use to modify their dependencies' code or for similar purposes. Libraries that include these inner `jar` files are, on average, enlarged by 10.8 MB. Our dataset contains

Table 5.6: File extensions by total size on the ecosystem

| Extension | Total (MB) | AVG/package (KB) | #Packages |
|---|---|---|---|
| jar | 308019 | 10881 | 29005 |
| class | 94075 | 369 | 336672 |
| so | 18697 | 5002 | 4012 |
| war | 10516 | 23256 | 459 |
| zip | 9778 | 4888 | 2069 |
| none | 9775 | 192 | 54346 |
| gz | 7634 | 6467 | 1219 |
| js | 5235 | 263 | 31959 |
| png | 5220 | 413 | 17238 |
| dll | 4515 | 2211 | 2417 |

29,005 releases (~7%) that include internal `jars`.

The table also lists other types of internal archives, such as `war`, `zip`, and `gz`, which cause similar problems to varying degrees. The second-largest file type is the `class` file. As this file extension corresponds to Java bytecode, it is unsurprising that it occupies 94 GB of space in our dataset. Of all the releases in our dataset that included an archive, 336,672 (82%) contained `class` files, with an average size of 369 KB per library. Comparing these figures with the repackaged files, the significant impact of the latter on the ecosystem becomes clear. On average, inner `jars` occupy 29.5 times more space per library than `class` files. In other words, inner `jars` occupy 3.27 times more space than the primary content of the ecosystem, `class` files. This problem is even more pronounced for inner `war` files. They occupy 23.2 MB of space per library, which is 63 times more than `class` files. However, due to the smaller number of libraries containing inner `war` files (only 459 libraries), the cumulative effect is not as significant as with `jar` files.

We identified two key issues that can cause libraries to become excessively large. First, large files, such as data or models, can contribute to the size of these libraries. Second, when combined, nested archives can also lead to an increase in library size.

## 5.5 How has the size of dependency sets evolved?

There is another critical aspect to consider in the space requirements of libraries. When developers incorporate a dependency into their project, they include not just the library itself but also its direct and transitive dependencies. This means the total space taken up by a program consists of its own space plus the space of its direct and transitive dependencies. Consequently, we examine the number of dependencies Maven libraries have.

**Collecting data** For each release, we resolve the dependency set and store both the number of its direct and transitive dependencies. The dependency resolution process is conducted using Shrinkwrap [178], an open-source implementation of Maven's default resolution. We do not limit dependency resolution results to the selected versions; Shrinkwrap may output any release, not just the sampled versions. Since we only need to compare the number of dependencies, the exact versions are less relevant to us. Additionally, we used the same resolver for all releases, so any inaccuracies inherited from the open-source

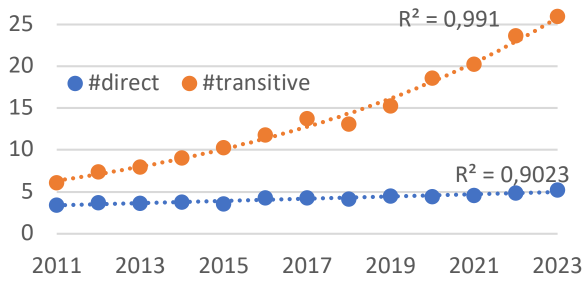


Figure 5.8: Average number dependencies per year

resolver affect all data points equally and should not impact our conclusions. However, for transparency, we store all resolution results, allowing other researchers to compare different resolutions with ours in the future.

**Findings** Figure 5.8 illustrates the average number of direct and transitive dependencies of Maven libraries. There is a notable increase in the number of dependencies that Maven libraries introduce over time. This rise is more subtle for direct dependencies compared to transitive ones. This figure also shows a linear trendline with $R^2$=0.9 for the growth in the average number of direct dependencies. In 2011, Maven libraries had on average 3.4 direct dependencies, which increased to 5.2 by 2023. This 53% increase in direct dependencies might not seem substantial initially. Nonetheless, it is essential to understand that this increase has a compounding effect on the ecosystem. An increase is passed down to all users of libraries and their subsequent users. Hence, an increase in the number of direct dependencies by one popular library can potentially affect a significant portion of the ecosystem, not just its immediate users.

Figure 5.8 also displays the average number of transitive dependencies. The increase in transitive dependencies is more pronounced than the direct ones, as it shows exponential growth. The exponential trendline on transitive dependencies fits with $R^2$=0.99. This means the degree of interdependency between libraries is increasing at an accelerating rate in Maven, which results in a substantial increase in the complexity of their dependency management process. The figure rose from 6 in 2011 to 25.9 in 2023, a 4.3X growth. A possible explanation for this increase in the number of transitive dependencies could be the aforementioned compounding propagation. With a 53% rise in the number of direct dependencies, stronger growth in the number of transitive dependencies can be anticipated.

The combination of a distinct upward trend in the library file sizes and the counts of direct and transitive dependencies results in an exponential growth in the space requirements of Maven applications. With a 2.3X increase in Maven library sizes and a 4.3X increase in the number of dependencies, the average Maven application in 2023 requires 10X more space than it did 12 years ago.

The Maven ecosystem has been witnessing substantial growth in the number of its libraries and their sizes. Furthermore, there is an exponential increase in the number of transitive dependencies, leading to a corresponding rise in the overall size of applications.

## 5.6 Discussion

We have presented several non-obvious and concerning practices that will compound and hinder the effective use and evolution of the Maven ecosystem if they go unaddressed. In this section, we delve into their implications and propose specific strategies to address them. We believe that this reflection can assist stakeholders, such as the governors of the ecosystem (ecosystem maintainers), in improving the overall health and robustness of the ecosystem, and represents one of the main contributions of this paper.

### 5.6.1 Erratic Data

**Inconsistent packaging** As shown in Section 5.2.1, there are inconsistencies in Maven packaging across various sources. These inconsistencies can compromise reliability and disrupt downstream tasks that depend on Maven libraries. Relying solely on the index or POM files may lead to errors. Manual inspection for accurate data per package is time-consuming and not scalable. Therefore, stricter automated checks and consistent communication about release processes are required from ecosystem maintainers and library developers to mitigate this issue. Reviewing configuration files and scripts, in addition to implementing an automated change-tracking system could help detect and rectify such discrepancies.

**Default packaging** Downstream tasks in Maven often assume that the majority of packages use the default jar packaging. However, our findings (as shown in Section 5.2.1) indicate that approximately 25% of libraries do not use the jar format. As such, relying solely on default packaging results in a significant number of overlooked libraries. We suggest that developers of downstream tasks consult the additional sources we identified in this study for a more comprehensive perspective.

**Design documentation** As discussed in Section 5.2.2, there is a possibility that users may mistakenly attribute a different purpose to checksums in Maven and consider using more powerful algorithms for improved security. Our findings indicate that this misunderstanding can be widespread throughout the ecosystem. Although current ecosystems offer comprehensive and well-maintained documentation to assist developers, we postulate that extensive and up-to-date resources are required for every ecosystem to explain the fundamental design decisions. Such documentation would also facilitate effective contributions from outsiders and researchers.

**Rigorous checks and monitoring** Some released packages violate Sonatype's extensive list of requirements for publishing artifacts in Maven Central [197], as discussed in Sections 5.2.2 and 5.2.3. For example, some packages do not include any checksums. We recommend that ecosystem maintainers enforce more rigorous checks for package releases and closely monitor the repository's content to prevent such cases in the future. Simple inconsistencies are easier to prevent in a more rigorous release process than through the execution of an external infrastructure like ours. More advanced checks might be needed though, for which this paper caters with the required infrastructure.

**Missing required files** Source code and documentation are explicitly required for every library on Maven Central, yet, they are missing for a significant percentage of libraries, as illustrated in Section 5.2.3. This poses difficulties for developers who rely on these resources, for example, they cannot get support from their development environments when working with these libraries. Libraries lacking these files disrupt this convenience and block downstream analyses dependent on these files, such as code quality attribute assessments. While library developers should prioritize addressing this issue, ecosystem maintainers should also improve their checks for these requirements. It is worth noting that even libraries that do have these files may contain meaningless content, as even the Maven documentation suggests uploading placeholder files with simple READMEs if the actual documentation cannot be provided, just to pass the checks. The fact that many libraries still lack documentation and sources despite this advice is concerning.

**Future research** Further research can provide insights into the proportion of libraries with placeholder files, reasons for the absence of proper documentation, or the use of placeholder files, as well as strategies and tools to encourage more libraries to provide these resources. The rise of Large Language Models (LLMs) holds the potential for improvements, as they simplify the documentation tasks. We encourage researchers to investigate the effects of LLMs on documentation uploaded to Maven. Exploring solutions that leverage LLMs to assist developers in preparing these files, and potentially generating them for libraries that lack them, would also be valuable.

### 5.6.2 Archive Misuse

**Large library problem** As discussed in Sections 5.3 and 5.5, the size of libraries and dependency sets has been increasing at an accelerating rate. While storage and computing capacities continue to grow, an exponential increase in the size of programs is not sustainable. Computers may continue to scale at a similar pace but the increasing size of programs significantly adds to their complexity and reduces maintainability. It is widely accepted that very large libraries slow down builds and deployments, making applications more challenging to distribute and scale. Furthermore, larger libraries tend to contain more code, making them more prone to bugs, security vulnerabilities, and breaking changes [30, 79]. The presence of multiple versions of packages, with newer versions often incorporating additional features, can contribute to the libraries' gradual growth. It is generally advantageous if developers pay attention to it and mitigate the impact through regular refactoring and improvements. *Large Files* and *Inner Archives* are two additional causes for the increase in size requirements, which can become severe problems if left unaddressed.

**Large files** As emphasized in Section 5.4, projects that incorporate significantly large files, such as data files, contribute substantially to overall size growth. Many of these projects are in the domains of machine learning and big data, a trend that aligns with the recent expansion in these fields. Beyond size-related issues, these files also present challenges in version control. Data often changes at a different pace compared to code, and merging them can compromise the ability to manage the code and data separately. Furthermore, data and model files are likely to increase in size at a faster rate than code. Therefore, we repeat the advice of modern ML-Ops practices and recommend that library maintainers ensure data and model files are versioned separately from the code [205]. Developers may also consider using specialized platforms like Hugging Face, TensorHub, or Kaggle, which are specifically designed for hosting machine-learning models and datasets.

**Inner archives** Inner archives contribute to larger libraries, as discussed in Section 5.4. When these files appear, they usually increase the size of the libraries substantially. Many of these inner files are libraries developers embed to modify functionalities or avoid complex dependency trees ("shading"). This practice results in overlooked vulnerability patches and improvements in subsequent versions of the nested library. These inner libraries also block downstream analyses, such as an automated vulnerability detector, which does not recognize the nested library. When libraries package another vulnerable library, they not only block the detection of vulnerability but effectively prevent a meaningful dependency management that could otherwise be used to introduce very specific updates to libraries in the dependency tree. Embedded libraries might cause code duplication when the original libraries are included as regular dependencies. Given that the ecosystem has millions of users and a highly complex network of transitive relationships, these files can accumulate rapidly and worsen the overall effect. To mitigate the accidental introduction of nested libraries by developers who may not thoroughly examine the content of their dependencies when adopting them, we recommend the automatic detection of such nesting upon deployment. It should then be mandatory for these libraries to openly document the content of the inner archives and justify their inclusion. This approach discourages the maintainers of such libraries from engaging in this practice while also allowing users to make informed decisions about whether they wish to adopt such libraries, given the potential risks.

**Future research** We identified various problems and inconsistencies in the ecosystem and discussed their combined effects. Yet, the impact of these issues from the users' perspective deserves exploration. Researchers could conduct surveys or ecosystem-wide impact analyses [79] to measure how these issues affect users. Future research could also explore the domain of the projects and examine how these issues evolve within the packages, such as whether they are resolved in subsequent versions or worsen over time. Moreover, these issues might correlate with other problems, like concealing vulnerabilities in inner archives. We encourage researchers to investigate the relationship between the problems identified in this study and related issues, such as examining the ratio of vulnerable inner archives and their propagation or assessing the quality of libraries engaging in such practices. They could also explore whether lower quality results in larger libraries and vice versa. The outcomes of these studies could inform the development of better tooling, like refactoring tools to break down or declutter large libraries. Researchers can utilize our infrastructure for follow-up studies and extend their investigations to other software ecosystems, such as npm and PyPI. This allows for the creation of comparable datasets to explore the aspects studied here across different ecosystems and to conduct comparisons.

### 5.6.3 Transitive Growth

**Exponential transitive growth** Our analysis in Section 5.5 demonstrates that the number of direct and transitive dependencies has been consistently increasing over time, with transitive dependencies expanding exponentially. This significant increase poses major risks for library users and the entire ecosystem. It contributes to larger project sizes, leading to decreased maintainability and increased complexity. Additionally, a larger number of dependencies amplifies potential vulnerability exposure. Developers must dedicate additional time to managing these dependencies [206] to avoid problems like "dependency hell" [207, 208]. Consequently, a critical challenge emerges concerning dependency resolution and usage within ecosystems.

**Dependency minimization** We believe this issue may stem from developers' lack of awareness about the extent of transitive effects. We recommend developers consistently be mindful of their dependencies, particularly transitive ones. When adding new dependencies, developers usually consider the new library's quality aspects. However, the quality and number of transitive dependencies a new addition brings can be equally important. We advise frequent reviews of dependency sets to decrease the number by eliminating unused ones or replacing those introducing more transitivity. Automated approaches like DEPCLEAN [29] can help reduce dependency bloat. The authors showed that 75.1% of dependencies in Maven are bloated, so debloating can significantly alleviate the issue. We encourage ecosystem maintainers to integrate research-based solutions into their tools, like Maven packaging, or at least into library deployment. Notably, the same authors indicated that transitive dependencies and the complexity of managing them in multi-module projects are the main bloat causes. This aligns with our findings. They also demonstrated that developers are eager to remove bloated dependencies. Thus, we deduce that the present situation stems from escalating complexity and unawareness, which is likely to worsen if current trends continue.

**Future research** Optimized dependency resolutions try to resolve dependencies based on specific goals, such as minimizing dependencies or vulnerabilities. Previous research has shown promising results for npm [209]. It is an open question whether the Maven ecosystem would also benefit from such approaches. Thus, we encourage researchers to explore optimized resolvers and their potential effects on Maven.

### 5.6.4 Threats to validity

In this section, we discuss the limitations of this study.

**Internal Validity** The internal validity of this study may be affected by several factors. We conducted multiple manual inspections, introducing the possibility of human errors. To mitigate this, we documented these cases. Additionally, one author performed manual investigations while another randomly verified cases, finding no mistakes. We also used the open-source tool ShrinkWrap for resolving dependencies. Although ShrinkWrap is widely used and actively maintained there is a potential for inheriting flaws from this tool. We addressed this by designing a flexible infrastructure that allows the incorporation of alternative dependency resolvers for comparison. Moreover, hidden bugs in our implementation could impact results. To mitigate this, we conducted numerous manual tests, developed an extensive test suite, and made our data and code publicly available for review.

**Construct Validity** The analysis of file types for executables was limited to artifacts classified as primary executables in the Maven Index file. Therefore, the findings pertain specifically to these executables and exclude other archive types in the repository for multi-archive packages. We acknowledge the potential issues with the Maven Index, as mentioned in Section 5.2.1. To mitigate this, we thoroughly reported on the prevalence of multi-archive packages within the ecosystem.

**External Validity** We acknowledge that there are 6,419 packages for which the POM file could not be resolved due to the parent being hosted on a different repository. While other repositories besides Maven Central can be investigated similarly, we only focused on Maven Central because it is the biggest and the main repository. We also used a subset of releases from Maven, which could bias our results. To mitigate this, we ensured that our data was statistically representative of both the entirety of the ecosystem and its history.

## 5.7 Related Work

Numerous studies analyzed various aspects of software ecosystems. In this section, we discuss the studies most relevant to this research and classify them into subsequent topics.

**Software updates** Kula et al. [36] found 59.63% of existing project dependencies and 81.16% of new dependencies used the latest versions. The same authors [107] observed a growth-peak-decay trend in 93.87% of popular Maven packages. Bavota et al. [47] studied the Apache Java ecosystem's [210] evolution, including dependency graphs, project size, and developer activity. Düsing et al. [108] found half of Maven vulnerabilities were patched before disclosure. Jafari et al. [211] discovered that the number of dependencies, age, and release status of packages influence their update strategies.

**Maven repository** Raemaekers et al. [28] created the Maven Dependency Dataset, revealing 68.4% of libraries had sources and 53.1% had Javadoc. Keshani et al. [79] linked method popularity to breaking changes, finding similar rates of breaking changes across popular and unpopular methods. Karakoidas et al. [106] generated a dataset from metrics related to object-oriented design and program size. Tufano et al. [116] found uncompilable snapshots in 96% of Apache projects due to dependency issues. He et al. [212] studied similar compilation issues. Soto-Valero et al. [29] found 75.1% of dependencies bloated and introduced DEPCLEAN to resolve it. They also [17] found that 30% of libraries had multiple actively used versions. Mitropoulos et al. [30] linked artifact size to bugs, finding larger artifacts are more likely to contain bugs.

**Software ecosystems** Abdalkareem et al. [109] explored trivial packages, finding they constitute 16.8% of npm packages studied. Chowdhury et al. [213] investigated the universality of these packages and found that removing even a single trivial package could impact up to 29% of the ecosystem, highlighting their importance despite their small size. Cogo et al. [110] studied same-day releases in the npm ecosystem, discovering that 96% of popular packages had at least one. 39% of these releases were error-prone, requiring additional same-day fixes. Cogo et al. [214] also investigated dependency downgrades in the npm ecosystem, finding that 49% of downgrades replace a range of acceptable versions with a specific old version, suggesting a more conservative approach to updates after a downgrade. Kula et al. [111] introduced the Software Universe Graph (SUG) for analyzing software ecosystems like Maven. The SUG allows insights into library popularity and adoption. Decan et al. [180, 215] and Kikas et al. [216] compared multiple ecosystems and elaborated on their differences. Hejderup et al. [217] proposed a method-level dependency network for analyzing crates.io. Claes et al. [218] conducted a historical analysis of package incompatibilities in the Debian ecosystem and studied how package conflicts evolve. Nguyen et al. [219] conducted a twelve-year longitudinal study on the evolution of the Debian software collection. They investigated the life cycle of packages from inception to end and analyzed attributes such as package age, bugs, maintainers, and popularity. Herraiz et al. [220] investigated the relationship between installation counts and perceived quality in the Debian ecosystem. Their study revealed that widely deployed programs tend to have more reported defects, regardless of their actual defect count.

**Library duplication** Benelallam et al.[16] developed an artifact-level model of the Maven Central Repository, showing that 12.5% of artifacts are duplicated or have corrupted pom.xml files, and unique libraries average 10 versions. Kanda et al.[112] studied inner `jar` file occurrences and duplications within the Maven Central Repository, revealing around 0.8% of `jar` files contained inner `jars`. Tools like DepTrim[221] prune symbols without errors. Repackaging can involve merging multiple `jars`, not easily detected by file extensions, often in the Android ecosystem, and identified using techniques like CodeMatch[222]. Y. Shao et al. [223] proposed a technique to detect these repackaged libraries, relying on external resources like GUI layout files, which remain unaltered by repackaging.

Overall, existing research provides insights into various facets of libraries and software ecosystems. However, none of them offers a comprehensive view of packaging practices, their trends, and impacts at the ecosystem level. This gap hinders the community, especially ecosystem maintainers, from making effective decisions for the future of the ecosystem. In this study, we complement the existing knowledge by offering a holistic view of the Maven ecosystem and its packaging practices that did not exist before.

## 5.8 Concluding Remarks

Our comprehensive examination of the Maven ecosystem and its Java libraries, including 479,915 releases, uncovered important insights into its evolution and practices. We pinpointed three critical bad practices within the ecosystem that deserve attention. The first, *Erratic Data* is characterized by the presence of missing and inconsistent data in the ecosystem. The second issue, *Archive Misuse*, highlighted the concerning usage of archives which contributes to an increasing trend in the average size of Maven libraries. The third bad practice, *Transitive Growth*, refers to the exponential rise in the number of dependencies, which has led to a tenfold increase in the space requirements for transitive dependency sets over the past 12 years. Our study's findings bear significant implications for various stakeholders in the ecosystem, including maintainers, library developers, and researchers. In response to these issues, we provided practical recommendations such as stricter checks at deployment stages and the minimization of dependency sets.

# 6 Conclusion

In this chapter, we revisit our research questions, provide answers to them, and discuss the implications and future directions of our research. We also explain the limitations of our studies and end this section with concluding remarks.

## 6.1 Research Questions Revisited

In this section, we answer the research questions that were laid out in Chapter 1.

### 6.1.1 RQ1: To what extent can published libraries be automatically reproduced from code?

This RQ explores the potential for automating the process of reproducing artifacts, which is critical for securing software supply chains. Security breaches, such as the SolarWinds attack [68], exploit the issue of non-reproducible binaries. The state of reproducibility varies across different ecosystems; in some, like Go and Debian, it is quite advanced [224], while in others, such as Maven, it is less developed. The initiative Reproducible Central, which curates a list of reproducible Maven libraries, represents a manual and limited solution to date.

In Chapter 2, we introduced AROMA, a tool designed to automate the process of linking Maven releases to their source code and reconstructing the original release environment. AROMA streamlines the identification of reproducible artifacts by extracting build information through heuristics. We demonstrated that automatic reproducibility is achievable for a substantial portion of the ecosystem. AROMA enables reproduction for 23.4% of packages, and full reproducibility was achieved for 8% of these.

We discussed several key insights and actionable recommendations regarding reproducibility. There seems to be a lack of awareness among developers regarding the `project.build.outputTimestamp` property that can be set in Maven projects, which significantly improves automated reproducibility. We suggested Maven tools to highlight the absence of this property, or even set it automatically [225] to encourage its use and enhance reproducibility as a quality attribute for packages. We highlighted the need for standardization in release tagging styles to facilitate automated processing and improve ecosystem transparency. We highlighted the potential to significantly expand the reproducible dataset despite only a small fraction of all Maven packages being fully or partially reproducible. We elaborated on the areas that could be improved in future work such as heuristics to extract `release profiles`.

Overall, our findings highlight the feasibility and importance of automating reproducibility within software ecosystems. Our study can enhance the security of Maven by achieving high accuracy in information retrieval and identifying flaws in existing manually curated lists. Furthermore, it contributes actionable insights and resources to the reproducibility community. It demonstrates that even in ecosystems like Maven, where reproducibility rates are currently lower compared to other ecosystems like Debian, automation can greatly accelerate the process towards achieving overall reproducibility. Nevertheless, the algorithms and techniques used in this study may need to be adjusted for application in other ecosystems.

### 6.1.2 RQ2: How can we improve the scalability of call graph generation for library analysis?

The scalability of method-level analysis of Maven library interactions, as discussed in Chapter 3, can be significantly enhanced through the adoption of a summarization-based approach for call graph generation. This approach was specifically tailored for Java programs and used caching of Partial CGs (PCGs) for dependencies that remain unchanged between builds. It employs a stitching algorithm named Frankenstein for merging these partial results into a CG for the entire program.

The proposed approach showed substantial improvements in performance over existing frameworks, with speed enhancements of up to 38% and an average memory requirement of just 388 megabytes despite processing millions of edges. This illustrates a substantial improvement in making static analyses practical for use in build systems with limited resources, without sacrificing the quality of the analyses. The generated CGs maintain a near-identical set of edges compared to baseline frameworks, achieving an F1 score of up to 0.98, which indicates a high level of precision and recall in the analysis results.

Furthermore, we highlighted the practicality of this approach by evaluating it in a real-world application, particularly in environments similar to GitHub's continuous integration environment. We also justified the adoption of Class-Hierarchy Analysis (CHA) for CG generation, despite its basic nature, due to its efficiency and the applicability of the generated CGs in practical use cases. We emphasized the importance of balancing soundness, precision, speed, and memory consumption in program analysis to make static analyses more relevant and feasible in real-world scenarios, especially those involving large projects with numerous dependencies. We highlighted the potential of summarization techniques to improve CG generation for practical use cases. We recommended that existing static analysis frameworks adopt these techniques for enhanced practicality. We emphasized the importance of configurability, modularity, and modularizing intermediate data, which would allow analysis modules to pause and resume tasks, enabling the incorporation of static analysis into daily use cases, such as continuous integration tools. We also discussed the trade-offs between accuracy, memory usage, and speed.

Although we implemented this approach only for Java, it can be adapted for other languages as well. In fact, within the FASTEN project, a similar approach was implemented for C++ and Python. Moreover, a company named EndorLabs [59] is currently extending this approach to other languages such as Go and Rust. Therefore, we believe that this approach can be generalized to other languages and help improve library analysis for the supply chain of other languages. However, it is important to note that adapting this approach to other languages can be challenging and requires extensive knowledge of the target language.

### 6.1.3 RQ3: How do libraries impact their users at the method level?

In Chapter 4, we discussed a detailed analysis of how libraries impact their users at the method level by examining the frequency and nature of semantic versioning violations across 13,876 versions of 384 artifacts. We studied the impact of these libraries on 7,190 dependent projects. We found that 67% of the artifacts introduced at least one type of semantic versioning violation, such as breaking changes or illegal API extensions, throughout their version history. Our impact analysis highlighted strong centralization in method usage: 87% of publicly accessible methods were never utilized by dependents, and half of the unique method calls from dependents were concentrated on only 35% of the used methods. This suggests that most public methods in a library are rarely, if ever, used by others. Additionally, our study found no correlation between method popularity and stability. Popular methods are just as likely to undergo breaking changes as less popular ones. This suggests a potential lack of awareness among library maintainers regarding the popularity of their methods and the impact of breaking changes on the ecosystem. The findings highlight the need for better tools and practices to inform library maintainers about the usage of their methods to enhance library upgrade strategies and minimize disruption caused by breaking changes.

We found that maintainers who extend their API in a way that is incompatible with semantic versioning not only engage in poor practices but also increase the likelihood of future breaking changes and violations. We found no direct correlation between method popularity and breaking changes. This highlights the importance of future research on the factors that influence maintainers' decisions. We proposed that library maintainers could benefit from data on method popularity when making decisions regarding breaking changes and prioritizing maintenance tasks. This could, for example, include the development of tools and features in IDEs, build tools, and platforms like GitHub and Maven.

The findings of this study were focused solely on Maven. We expect that other ecosystems may also exhibit similar trends. However, future research should extend our analyses to other ecosystems and compare the results. Nonetheless, the methodology enabling our analyses has also been implemented for C++ and Python within the FASTEN project. Therefore, we believe it is feasible to conduct similar studies for other languages.

### 6.1.4 RQ4: How do the packaging practices of libraries impact the package repositories?

As discussed in Chapter 5, our analysis identifies three specific issues arising from the existing packaging practices within the Maven ecosystem: the presence of missing and inconsistent data, improper utilization of archives, and an exponential increase in the number of transitive dependencies.

These issues could be mitigated largely by better governance of the ecosystem, as suggested in Chapter 5. However, understanding the concrete reasons behind developers' frequent engagement in such practices requires further research into their processes and mindsets when developing and deploying their libraries. Nevertheless, it is clear that if library maintainers were more mindful of their impact on users, many of these issues could be avoided. For example, a developer deciding whether to include a dependency for a simple task or to implement the task themselves would benefit from understanding the transitive impact of adding this dependency on the ecosystem. Additionally, raising awareness about the security aspects of developing and deploying libraries could significantly mitigate the mentioned issues. For instance, informing developers that the library they are considering calls a vulnerable method from another library might encourage them to seek a more secure option. Similarly, if developers were made aware that an alternative library not only avoids using any vulnerable methods but also is reproducible, they might be encouraged to use that library. Another scenario involves a developer attempting to add a library's jar file to another archive as an inner jar. Being notified about the library's vulnerabilities could discourage them from this practice, as they would understand that it exposes dependents to risk and effectively conceals the vulnerability from them. In this thesis, we proposed solutions aimed at increasing developers' awareness of their users and the security implications of their programs and dependencies. We believe these measures would help mitigate the existing challenges within the ecosystem.

We identified pressing issues in Maven. To mitigate them, we recommended various solutions such as stricter checks and ecosystem design documentation. We suggested future research to explore the impact of Large Language Models (LLMs) on documentation quality in Maven. We pointed out the problems with large libraries, large files, and inner archives and discussed their negative impacts on maintainability, performance, and security. We discussed strategies to address these issues, including adhering to modern ML-Ops practices for separate versioning of data/model files and code. We explained that the exponential increase in the number of transitive dependencies is a major risk, contributing to larger project sizes and increased complexity. We advised developers to be mindful of their dependencies, especially transitive ones, and recommended using automated tools like DEPCLEAN to reduce dependency bloat. We also encouraged ecosystem maintainers to integrate debloating practices into development and deployment tooling. We proposed future research directions such as exploring the evolution of these issues over time and examining the potential benefits of optimized dependency resolutions for the Maven ecosystem.

Although this study focused exclusively on the Maven ecosystem, we believe the findings and recommendations could be also beneficial for other ecosystems. We acknowledge that the challenges faced by each ecosystem may be unique; however, the similarities among different ecosystems suggest that some of these challenges could be common across them. For instance, any build tool that enables transitive dependencies might cause a similar exponential growth to what we observed in Maven. Ecosystems like NPM, famous for the use of small and numerous libraries, could encounter even more critical issues. Nevertheless, future research should explore the challenges we identified in Maven within other ecosystems to find their prevalence and impact.

## 6.2 Discussion

In this section, we elaborate on the impact of this thesis, new insights, ideas, and future directions that emerged from this thesis. We also explain the limitations of the thesis.

**Reflection on the impact of the thesis** In this thesis, we started by envisioning more effective dependency management for Maven projects. Our goal was to create better tooling and techniques for dependency analysis. This thesis was also part of the FASTEN project, which aimed at enhancing the accuracy of dependency analysis using CGs. Specifically, our goal was to address challenges regarding the impact and security of libraries. The tools and techniques developed in this thesis formed the core of FASTEN. Frankenstein was directly used within FASTEN to analyze Java libraries in various scenarios and use cases. Other work packages of FASTEN were also inspired by Frankenstein and implemented similar techniques for Python and C++. One of the FASTEN industrial partners, Software Improvement Group (SIG) [60], continues to use our tooling and technologies in their products to provide services for their clients. Later, a US-based startup in the field, EndorLabs [59], also adopted Frankenstein and expanded its application to more languages, such as Go and Rust. Given these accomplishments, we believe we have largely achieved what we envisioned at the start of this thesis. However, we also identified many areas that could be further improved and new research directions that emerged from our work. In the next section, we will discuss these directions and the existing challenges in detail.

**Limitations** This section provides an overview of the challenges faced in this thesis and the measures taken to ensure the reliability and generalizability of the findings.

Internal validity concerns in this thesis were addressed through a combination of manual inspections, extensive testing, and public availability of code and data across all chapters. The reliance on manual inspections and heuristics, despite the risk of human error and the potential for overlooking corner cases, was mitigated by involving multiple authors in these processes and by benchmarking against manually curated baselines or existing frameworks. The dependency on external tools like ShrinkWrap, WALA, and OPAL introduced a risk of inheriting their limitations. This was addressed by developing flexible infrastructures to allow for the integration of alternative tools and by making our implementations and test suites publicly available for validation by the community.

The external validity of this thesis is primarily influenced by its focus on specific tools, languages, and repositories, which may limit the generalizability of its findings to other contexts. The studies mainly utilized Maven and Java projects, leveraging static analysis frameworks like WALA and OPAL. This focus was chosen due to the prevalence and importance of Java and Maven in the software development ecosystem, as well as the active maintenance of the selected analysis tools. However, this approach excludes projects hosted on other repositories or developed in other JVM-based languages such as Scala or Kotlin.

To mitigate these external validity concerns, the thesis acknowledges its scope limitations and suggests areas for future research. Replicating the study in different ecosystems, examining projects developed in other JVM-based languages, and exploring the impact of different version control systems could provide broader insights into software dependency management and its challenges.

**Analyses precision** The motivation behind transitioning from package-level to method-level analysis lies in the increased precision of analyses. Analyzing dependencies at the package level is highly inaccurate. Moving to method-level analysis is a significant step towards achieving more accurate analyses, as outlined in this thesis. However, CGs are not entirely accurate; they tend to overapproximate to remain sound, resulting in many false positive edges in a CG. Consequently, conducting analyses on top of CGs may also cause inaccuracies. The analyses may mistakenly inform developers that they are using a function in their dependencies when they are not. This happens due to the reported chain of calls originating from overapproximated edges in the CG. Our experiments frequently observed such cases, highlighting a limitation of current state-of-the-art analyses. Recently, there has been significant progress towards developing more practical CGs that prioritize precision without solely focusing on soundness. These techniques [194, 226, 227] prune the CGs to enhance their practicality for downstream analyses such as the ones we addressed in this thesis. These studies leverage Machine Learning (ML) to prune CG edges. However, these approaches are costly post-processing of the resulting CG when applied to the entire CG. We observed cases where overapproximation is concentrated in certain areas of CGs. These areas could potentially be found through simple heuristics and direct ML techniques to focus their effort only on these parts for better scalability. Such cases are recurring patterns of false positives, particularly related to native Java methods that lead to numerous overapproximations in the CG. For instance, when `Iterator.iterate` is overridden and used extensively within a program, it leads to the generation of many edges in a CG. Yet, `Iterator.iterate` typically executes basic operations and may not be relevant in dependency analysis scenarios, where the focus is on calls to dependencies. This research direction deserves further exploration in future research, where configuring the analysis to relax certain requirements in specific scenarios could produce more practical results for certain use cases.

**Compatibility** While studying the compatibility of packages in software ecosystems, we observed a fundamental challenge. Libraries use the Semantic Versioning convention to signal to their users about the changes made in each version. By doing so, maintainers announce, in their opinion, whether the current version is compatible with the previous ones or not. However, we realized that, firstly, it is very subjective and error-prone to ask a person to determine if the changes they introduced in the past few weeks or months are compatible. Secondly, compatibility is a feature of clients and not the libraries. Changes that may break one client may be totally safe for another. Thirdly, semantic changes may also break clients, and changes do not have to only affect the APIs to break the clients; it is not possible to detect such cases automatically. Therefore, future research can consider addressing these challenges by creating a network of users similar to what we did in Chapter 4 and examine compatibility by running the users' tests or even only by trying to compile them. This will enable us to treat compatibility as a more dynamic concept that can be largely automated and made more accurate.

**Standardization and distribution** Central repositories have become popular over the past decade, enabling clients to download their dependencies from a central location. These repositories conduct minimal quality checks on released items to ensure there are no fundamental flaws in the packages. Unfortunately, Chapter 5 showed that these checks are not always thorough, and many artifacts uploaded to the ecosystems contain flaws. Additionally, we have observed that many developers employ release practices that are harmful to the ecosystem. There is also a noticeable trend where developers host their artifacts in their own repositories instead of in central ones. Thus, a future research direction is to develop standards and protocols for library releases, along with rigorous automatic checks for libraries that adhere to these standards regardless of their repository location. This approach encourages a more distributed method of publishing libraries, eliminates the necessity for everyone to publish to a central location, and enables developers to release their binaries closer to their code. Moreover, by subscribing to the provided standards and protocols, developers will ensure a high level of quality that is transparent to the users.

**Support beyond writing code** In this thesis, we observed many instances of a lack of quality and awareness regarding important aspects of software, such as security, across different development phases. Many developers rely on libraries found on the internet with minimal investigation into their quality. The free nature of open-source software makes it challenging to incentivize quality-focused efforts. Additionally, we observed the number of transitive dependencies increasing exponentially, which means developers' actions can have a significant impact on the ecosystem transitively. For instance, a library maintainer might add a dependency, believing it to be harmless. However, they cannot foresee the effect of this addition on their users, as dependency relationships are dynamic and can change from one moment to another. The addition of a specific library might suddenly make a user call a vulnerable method due to a change in their class hierarchy. Even if developers were incentivized to deliver higher quality, the growing complexity makes it impractical for them to do so manually and thoroughly.

To mitigate these issues, a deeper understanding of not only our programs but also our processes and dependencies is necessary. Furthermore, the development of tools and techniques for better quality and the automation of processes that support this are beneficial. Currently, automation efforts are mainly focused on implementation tasks that enable the generation of more code in less time. A notable example is using Large Language Models (LLMs) for code generation. While LLMs automate many aspects of coding, they may hinder the deep understanding of complex systems if used carelessly or they may generate low-quality code. Recent studies [228] have shown that the licensing, privacy, and security implications of LLM-generated code are often overlooked. Given the discussed points, we expect LLMs to worsen the situation and contribute to the lack of quality and awareness, and security problems.

We believe the research community should prioritize automation in other aspects of development, such as maintenance, improvement of quality and security, and intuitive methods that help developers understand complex systems more deeply, rather than solely focusing on reducing implementation time and effort. This thesis represents the first step towards this goal, but it is insufficient, as automation in these areas significantly lags behind the automation of code generation. Automated tools that improve quality and security, such as those proposed in this thesis, as well as those that ease maintenance and enhance understandability, should be explored more extensively. For example, researchers could consider using graph neural networks to assess or predict the addition, removal, or replacement of library links for optimized security and quality.

## 6.3 Concluding remarks

This thesis explores and introduces novel approaches to address security challenges within the Maven ecosystem, an essential part of the software supply chain. The insights, methods, and tools presented provide valuable resources for developers, maintainers, and researchers. Our contributions include the development of tools for library reproducibility and scalable, call-graph-based dependency analysis techniques. We have introduced a set of methods to enhance dependency management and made them publicly available. Through empirical research and data mining, we examined the current state of Maven and identified issues in packaging practices that pose risks to ecosystem users. Additionally, we discussed various directions for future research that have emerged from this work. The results of this thesis were published in premier venues of our field, applied in practical software by our partners (SIG [60]), and continued by affiliated companies (Endorlabs [59]), which not only shows the impact of this work but also highlights the importance of future work in this research direction. We hope that the availability of our tools and datasets will enable others to continue where our work in this thesis has ended.

# Bibliography

## References

[1] Software supply chain management: An introduction. https://www.sonatype.com/resources/software-supply-chain-management-an-introduction. Accessed: 2024-02-01.

[2] What is software supply chain security? https://www.redhat.com/en/topics/security/what-is-software-supply-chain-security. Accessed: 2024-02-01.

[3] E. Levy. Poisoning the software supply chain. *IEEE Security & Privacy*, 2003.

[4] Robert J Ellison, John B Goodenough, Charles B Weinstock, and Carol Woody. Evaluating and mitigating software supply chain security risks. *Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TN-016*, 2010.

[5] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 2022.

[6] Log4shell. https://en.wikipedia.org/wiki/Log4Shell. Accessed: 2024-03-20.

[7] Nusrat Zahan, Elizabeth Lin, Mahzabin Tamanna, William Enck, and Laurie Williams. Software bills of materials are required. are we there yet? *IEEE Security & Privacy*, 2023.

[8] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, 2020.

[9] Log4j is why you need a software bill of materials (sbom). https://www.reversinglabs.com/blog/log4j-is-why-you-need-an-sbom. Accessed: 2024-03-20.

[10] One year after log4shell, firms still struggle to hunt down log4j. https://www.contrastsecurity.com/security-influencers/one-year-after-log4shell-firms-still-struggle-to-hunt-down-log4j. Accessed: 2024-03-20.

[11] Sbom. https://security.cms.gov/learn/software-bill-materials-sbom. Accessed: 2024-03-20.

[12] Executive order 14028. https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity. Accessed: 2023-07-31.

[13] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[14] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. What are weak links in the npm supply chain? *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022.

[15] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the Maven ecosystem. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 201–211. IEEE, 2023.

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

[17] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in Maven Central. *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019.

[18] César Soto-Valero, Thomas Durieux, and Benoit Baudry. A longitudinal analysis of bloated java dependencies. *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[19] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering*, 2020.

[20] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[21] Maven stats. https://mvnrepository.com/repos. Accessed: 2024-03-24.

[22] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. Dependency smells in javascript projects. *IEEE Transactions on Software Engineering*, 2021.

[23] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[24] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2021.

[25] Md Mahir Asef Kabir, Ying Wang, Danfeng Yao, and Na Meng. How do developers follow security-relevant best practices when using npm packages? *2022 IEEE Secure Development Conference (SecDev)*, 2022.

[26] Stan Zajdel, Diego Elias Costa, and Hafedh Mili. Open source software: an approach to controlling usage and risk in application ecosystems. *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume A*, 2022.

[27] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. Towards better dependency management: A first look at dependency smells in python projects. *IEEE Transactions on Software Engineering*, 2022.

[28] Steven Raemaekers, Arie Van Deursen, and Joost Visser. The Maven repository dataset of metrics, changes, and dependencies. *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[29] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering*, 2021.

[30] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The bug catalog of the Maven ecosystem. *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014.

[31] Xinlei Ma and Yan Liu. An empirical study of Maven archetype. *SEKE*, 2020.

[32] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. Breaking bad? semantic versioning and impact of breaking changes in Maven Central: An external and differentiated replication study. *Empirical Software Engineering*, 2022.

[33] Steven Raemaekers, Arie van Deursen, and Joost Visser. The Maven repository dataset of metrics, changes, and dependencies. *10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[34] Nicolas Harrand, Amine Benelallam, César Soto-Valero, François Bettega, Olivier Barais, and Benoit 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*, 2022.

[35] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. The nature of build changes: An empirical study of Maven-based build systems. *Empirical Software Engineering*, 2021.

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

[37] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software*, 2017.

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

[39] Tobias Lauinger, Abdelberi Chaabane, and Christo Wilson. Thou shalt not depend on me: A look at javascript libraries in the wild. *Queue*, 16(1):62–82, 2018.

[40] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. *Proceedings of the 15th international conference on mining software repositories*, 2018.

[41] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 2023.

[42] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

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

[44] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 1984.

[45] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the API usage in java. *Information and software technology*, 2016.

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

[47] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering*, 2015.

[48] André Hora, Anne Etien, Nicolas Anquetil, Stéphane Ducasse, and Marco Tulio Valente. Apievolutionminer: Keeping API evolution under control. *Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014.

[49] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Classification of changes in API evolution. *IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, 2019.

[50] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. A systematic review of API evolution literature. *ACM Comput. Surv.*, 2021.

[51] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. *28th IEEE International Conference on Software Maintenance (ICSM)*, 2012.

[52] Xz. https://www.puppet.com/blog/xz-backdoor. Accessed: 2024-04-06.

[53] Solarwinds compromise. https://attack.mitre.org/campaigns/C0024/. Accessed: 2024-05-08.

[54] Log4shell vulnerability. https://www.ibm.com/topics/log4shell. Accessed: 2024-05-08.

[55] left-pad incident. https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm. Accessed: 2024-03-26.

[56] Equifax incident. https://en.wikipedia.org/wiki/2017_Equifax_data_breach. Accessed: 2024-03-26.

[57] Fasten website. https://www.fasten-project.eu/.

[58] Dependabot. https://github.com/dependabot.

[59] Endorlabs. https://www.endorlabs.com/.

[60] Sig. https://www.softwareimprovementgroup.com/.

[61] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering*, 28(8):721–734, 2002.

[62] Paul Ralph, Nauman bin Ali, Sebastian Baltes, Domenico Bianculli, Jessica Diaz, Yvonne Dittrich, Neil Ernst, Michael Felderer, Robert Feldt, Antonio Filieri, et al. Empirical standards for software engineering research. *arXiv preprint arXiv:2010.03525*, 2020.

[63] Empirical standards. https://github.com/acmsigsoft/EmpiricalStandards/tree/master/docs/standards. Accessed: 2024-05-08.

[64] Roel Wieringa. Design science methodology: principles and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 493–494, 2010.

[65] Ahmed E Hassan. The road ahead for mining software repositories. In *2008 frontiers of software maintenance*, pages 48–57. IEEE, 2008.

[66] Tudelft research. https://research.tudelft.nl/. Accessed: 2024-04-06.

[67] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of Maven artifacts. *FSE*, 2024.

[68] Solarwinds attack. https://www.cisecurity.org/solarwinds. Accessed: 2023-09-28.

[69] David A Wheeler. Countering trusting trust through diverse double-compiling. *21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[70] Reproducible central repo. https://github.com/jvm-repo-rebuild/reproducible-central. Accessed: 2023-06-20.

[71] Maven pom. https://maven.apache.org/pom.html. Accessed: 2023-06-05.

[72] Mehdi Keshani. Scalable call graph constructor for Maven. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 99–101. IEEE, 2021.

[73] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *European Conference on Object-Oriented Programming*, 1995.

[74] Continuous integration. https://en.wikipedia.org/wiki/Continuous_integration. Accessed: 2024-03-26.

[75] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207, 2017.

[76] Github actions. https://github.com/features/actions. Accessed: 2024-03-26.

[77] Travis ci. https://www.travis-ci.com/. Accessed: 2024-03-26.

[78] Semantic versioning. https://semver.org/. Accessed: 2024-03-26.

[79] Mehdi Keshani, Simcha Vos, and Sebastian Proksch. On the relation of method popularity to breaking changes in the Maven ecosystem. *Journal of Systems and Software*, 203:111738, 2023.

[80] Mehdi Keshani, Gideon Bot, Priyam Rungta, Maliheh Izadi, Arie Van Deursen, and Sebastian Proksch. Maven unzipped: Packaging impacts the ecosystem. *Under review*, 2024.

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

[82] Reproducible builds. https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=74682318. Accessed: 2023-06-14.

[83] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. Automated localization for unreproducible builds. *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[84] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. Root cause localization for unreproducible builds via causality analysis over system call tracing. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[85] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. Automated patching for unreproducible builds. *Proceedings of the 44th International Conference on Software Engineering*, 2022.

[86] Debainrepstats. https://reproducible-builds.org/citests/. Accessed: 2023-09-19.

[87] Replication package. https://doi.org/10.5281/zenodo.8380775. Accessed: 2023-09-27.

[88] Maven index. https://repo.maven.apache.org/maven2/.index/. Accessed: 2022-10-28.

[89] Maven version plugin. https://www.mojohaus.org/versions/versions-maven-plugin/index.html. Accessed: 2023-09-27.

[90] Maven scm requirenment. https://central.sonatype.org/publish/requirements/#scm-information. Accessed: 2023-09-28.

[91] Jiakun Liu, Xin Xia, David Lo, Haoxiang Zhang, Ying Zou, Ahmed E. Hassan, and Shanping Li. Broken external links on stack overflow. *IEEE Transactions on Software Engineering*, 2022.

[92] Vcs ranking. https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems. Accessed: 2023-09-28.

[93] Git tagging. https://git-scm.com/book/en/v2/Git-Basics-Tagging. Accessed: 2023-09-28.

[94] apache repository. https://infra.apache.org/blog/relocation-of-apache-git-repositories. Accessed: 2023-08-22.

[95] Maven requirentments. https://central.sonatype.org/publish/requirements/#developer-information. Accessed: 2023-08-23.

[96] Replacing build-jdk with build-jdk-spec jira. https://issues.apache.org/jira/browse/MSHARED-797. Accessed: 2023-09-26.

[97] Replacing build-jdk with build-jdk-spec github. https://github.com/apache/maven-archiver/pull/2/files. Accessed: 2023-09-25.

[98] Maven plugin. https://maven.apache.org/plugins/maven-compiler-plugin/. Accessed: 2023-06-13.

[99] Maven model. https://maven.apache.org/ref/3.0.4/maven-model/apidocs/org/apache/maven/model/Model.html. Accessed: 2023-06-12.

[100] Plugin management. https://maven.apache.org/pom.html#Plugin_Management. Accessed: 2023-06-13.

[101] Rc default Maven. https://github.com/jvm-repo-rebuild/reproducible-central/blob/844298749c5f78b2a914f9180b949d9e1fc2bc56/doc/BUILDSPEC.md#format. Accessed: 2023-09-28.

[102] Maven package: com.crawljax.crawljax-cli. https://mvnrepository.com/artifact/org.infinispan/infinispan-commons-jdk21. Accessed: 2023-08-01.

[103] Git newline. https://www.git-scm.com/book/en/v2/Customizing-Git-Git-Configuration. Accessed: 2023-09-28.

[104] diffoscope. https://diffoscope.org/. Accessed: 2023-09-25.

[105] Reproducible build Maven plugin. https://zlika.github.io/reproducible-build-maven-plugin/. Accessed: 2024-02-11.

[106] Vassilios Karakoidas, Dimitrios Mitropoulos, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. Generating the blueprints of the java ecosystem. *IEEE International Working Conference on Mining Software Repositories*, 2015.

[107] Raula Kula, Daniel German, Takashi Ishio, Ali Ouni, and Katsuro Inoue. An exploratory study on library aging by monitoring client usage in a software ecosystem. *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[108] Johannes Düsing and Ben Hermann. Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. *Digital Threats: Research and Practice*, 2022.

[109] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017.

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

[111] Raula Gaikovina Kula, Coen De Roover, Daniel M. German, Takashi Ishio, and Katsuro Inoue. Modeling library popularity within a software ecosystem. *Tech. Rep.*, 2017.

[112] Tetsuya Kanda, Daniel Morales German, Takashi Ishio, and Katsuro Inoue. Measuring copying of java archives. *Electronic Communications of the EASST*, 2014.

[113] Chen Zhang, Bihuan Chen, Junhao Hu, Xin Peng, and Wenyun Zhao. Buildsonic: Detecting and repairing performance-related configuration smells for continuous integration builds. *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[114] Ahmed Tamrawi, Hoan Anh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. Build code analysis with symbolic evaluation. *2012 34th International Conference on Software Engineering (ICSE)*, 2012.

[115] Mehdi Keshani, Georgios Gousios, and Sebastian Proksch. Frankenstein: fast and lightweight call graph generation for software builds. *Empir. Softw. Eng.*, 29(1):1, 2024.

[116] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 2017.

[117] Paul Gazzillo. Kmax: Finding all configurations of kbuild makefiles statically. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[118] Thodoris Sotiropoulos, Stefanos Chaliasos, Dimitris Mitropoulos, and Diomidis Spinellis. A model for detecting faults in build specifications. *Proceedings of the ACM on Programming Languages*, 2020.

[119] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. *Proceedings of the 40th international conference on software engineering*, 2018.

[120] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. History-driven build failure fixing: how far are we? *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[121] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. Understanding build issue resolution in practice: symptoms and fix patterns. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[122] Andrea Höller, Nermin Kajtazovic, Tobias Rauter, Kay Römer, and Christian Kreiner. Evaluation of diverse compiling for software-fault detection. *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.

[123] Yong Shi, Mingzhi Wen, Filipe R Cogo, Boyuan Chen, and Zhen Ming Jiang. An experience report on producing verifiable builds for large-scale commercial systems. *IEEE Transactions on Software Engineering*, 2021.

[124] Hongjun He, Jicheng Cao, Lesheng Du, Hao Li, Shilong Wang, and Shengyu Cheng. Constbin: A tool for automatic fixing of unreproducible builds. *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2020.

[125] Omar S Navarro Leija, Kelly Shiptoski, Ryan G Scott, Baojun Wang, Nicholas Renner, Ryan R Newton, and Joseph Devietti. Reproducible containers. *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[126] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. It's like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. *In 44th IEEE Symposium on Security and Privacy*, 2023.

[127] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. On business adoption and use of reproducible builds for open and closed source software. *Software Quality Journal*, 2022.

[128] Xavier de Carné de Carnavalet and Mohammad Mannan. Challenges and implications of verifiable builds for security-critical open-source software. *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[129] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in GitHub. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *In the proceedings of the 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE, Bergamo, Italy*, pages 805–816. ACM, 2015.

[130] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, 2018.

[131] Paolo Boldi and Georgios Gousios. Fine-Grained Network Analysis for Modern Software Ecosystems. *ACM TRANSACTIONS ON INTERNET TECHNOLOGY*, 21(1):1:1–1:14, 2021.

[132] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In Anders Møller and Manu Sridharan, editors, *In the proceedings of the 35th European Conference on Object-Oriented Programming, ECOOP, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 2:1–2:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[133] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[134] John Toman and Dan Grossman. Taming the Static Analysis Beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages, SNAPL, Asilomar, CA, USA*, volume 71 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[135] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. Modular call graph construction for security scanning of Node.js applications. In Cristian Cadar and Xiangyu Zhang, editors, *In the Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Virtual Event, Denmark*, pages 29–41. ACM, 2021.

[136] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a Balance: Pruning False-Positives from Static Call Graphs. In *In the proceedings of the 44th IEEE/ACM International Conference on Software Engineering, ICSE, Pittsburgh, PA, USA*, pages 2043–2055. ACM, 2022.

[137] h2o project. https://mvnrepository.com/artifact/ai.h2o/sparkling-water-package_2.11/3.26.8-2.4. Accessed: 2022-10-21.

[138] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-Large-Scale Software Repository and Source-Code Mining. *ACM Transactions on Software Engineering and Methodology*, 25(1):7:1–7:34, 2015.

[139] Raula Kula, Daniel German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 2018.

[140] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In Sukyoung Ryu, editor, *In the proceedings of the 16th Asian Symposium on Programming Languages and Systems, APLAS, Wellington, New Zealand*, volume 11275 of *Lecture Notes in Computer Science*, pages 69–88. Springer, 2018.

[141] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Y. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *In the proceedings of the ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA*, pages 762–774. ACM, 2014.

[142] Michael Eichberg, Florian Kübler, Dominik Helm, Michael Reif, Guido Salvaneschi, and Mira Mezini. Lattice based modularization of static analyses. In Julian Dolby, William G. J. Halfond, and Ashish Mishra, editors, *In the companion proceedings for the ISSTA/ECOOP Workshops, Amsterdam, Netherlands*, pages 113–118. ACM, 2018.

[143] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[144] T. j. watson libraries for analysis. http://wala.sf.net/. Accessed: 2022-01-15.

[145] Replication package for “frankenstein: fast and lightweight call graph generation for software builds”. https://github.com/ashkboos/LightWeightCGs/tree/mainrepPackage.

[146] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In Mary Beth Rosson and Doug Lea, editors, *In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications OOPSLA, Minneapolis, Minnesota, USA*, pages 281–293. ACM, 2000.

[147] Olin Shivers. Control-Flow Analysis in Scheme. In Richard L. Wexelblat, editor, *In the proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI, Atlanta, Georgia, USA*, pages 164–174. ACM, 1988.

[148] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In Mary Beth Rosson and Doug Lea, editors, *In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, Minneapolis, Minnesota, USA*, pages 264–280. ACM, 2000.

[149] Amitabh Srivastava. Unreachable Procedures in Object-Oriented Programming. *LOPLAS*, 1(4):355–364, 1992.

[150] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[151] David F. Bacon and Peter F. Sweeney. Fast Static Analysis of C++ Virtual Function Calls. In Lougie Anderson and James Coplien, editors, *In the proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, San Jose, California, USA*, pages 324–341. ACM, 1996.

[152] William Landi. Undecidability of Static Analysis. *LOPLAS*, 1(4):323–337, 1992.

[153] G. Ramalingam. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, 1994.

[154] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 22(1):162–186, 2000.

[155] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In Gregg Rothermel and Doo-Hwan Bae, editors, *In the proceedings of the 42nd International Conference on Software Engineering, ICSE, Seoul, South Korea*, pages 1049–1060. ACM, 2020.

[156] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In Dongmei Zhang and Anders Møller, editors, *In the proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA, Beijing, China*, pages 251–261. ACM, 2019.

[157] Lam, Patrick and Bodden, Eric and Lhoták, Ondrej and Hendren, Laurie. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop CETUS*, volume 15, 2011.

[158] The doop project. http://doop.program-analysis.org/. Accessed: 2022-01-15.

[159] Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In John Field and Gregor Snelting, editors, *In the proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE, Snowbird, Utah, USA*, pages 97–103. ACM, 2001.

[160] Chord: A program analysis platform for java. https://www.seas.upenn.edu/~mhnaik/chord/user_guide/index.html. Accessed: 2022-01-15.

[161] Karim Ali and Ondrej Lhoták. Application-Only Call Graph Construction. In James Noble, editor, *In the proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP, Beijing, China*, volume 7313 of *Lecture Notes in Computer Science*, pages 688–712. Springer, 2012.

[162] Karim Ali and Ondrej Lhoták. Averroes: Whole-Program Analysis without the Whole Program. In Giuseppe Castagna, editor, *In the proceedings of the 27th European Conference on Object-Oriented Programming, ECOOP, Montpellier, France*, volume 7920 of *Lecture Notes in Computer Science*, pages 378–400. Springer, 2013.

[163] Karim Ali. *The Separate Compilation Assumption.* PhD thesis, University of Waterloo, Ontario, Canada, 2014.

[164] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for Java libraries. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *In the proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE, Seattle, WA, USA*, pages 474–486. ACM, 2016.

[165] Amie L. Souter and Lori L. Pollock. Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance. In *In the proceedings of the International Conference on Software Maintenance, ICSM, Florence, Italy*, pages 682–691. IEEE Computer Society, 2001.

[166] Carol V. Alexandru, Sebastiano Panichella, Sebastian Proksch, and Harald C. Gall. Redundancy-free analysis of multi-revision software artifacts. *Empirical Software Engineering*, 24(1):332–380, 2019.

[167] Steven Arzt and Eric Bodden. StubDroid: automatic inference of precise data-flow summaries for the android framework. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *In the proceedings of the 38th International Conference on Software Engineering, ICSE, Austin, TX, USA*, pages 725–735. ACM, 2016.

[168] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA*, pages 49–61. ACM Press, 1995.

[169] Sharir, Micha and Pnueli, Amir and others. Two approaches to interprocedural data flow analysis. In *New York University. Courant Institute of Mathematical Sciences*, 1978.

[170] John Whaley and Martin C. Rinard. Compositional Pointer and Escape Analysis for Java Programs. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, Denver, Colorado, USA*, pages 187–206. ACM, 1999.

[171] Denis Gopan and Thomas W. Reps. Low-Level Library Analysis and Summarization. In Werner Damm and Holger Hermanns, editors, *In the proceedings of the 19th International Conference on Computer Aided Verification, CAV, Germany*, volume 4590 of *Lecture Notes in Computer Science*, pages 68–81. Springer, 2007.

[172] Atanas Rountev, Scott Kagan, and Thomas J. Marlowe. Interprocedural Dataflow Analysis in the Presence of Large Libraries. In Alan Mycroft and Andreas Zeller, editors, *In the proceedings of the 15th International Conference on Compiler Construction, CC, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Vienna, Austria*, volume 3923 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 2006.

[173] Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In Laurie J. Hendren, editor, *In the proceedings of the 17th International Conference on Compiler Construction, CC, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS, Budapest, Hungary*, volume 4959 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2008.

[174] Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA*, pages 567–577. ACM, 2011.

[175] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In Eelco Visser and Yannis Smaragdakis, editors, *In the proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, part of SPLASH, Amsterdam, The Netherland*, pages 359–377. ACM, 2016.

[176] Maven version ranges. https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, 2022. Accessed: 2022-10-21.

[177] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In Bjørn N. Freeman-Benson and Craig Chambers, editors, *In the proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA, Vancouver, British Columbia, Canada*, pages 183–200. ACM, 1998.

[178] Shrinkwrap. https://github.com/shrinkwrap/resolver. Accessed: 2023-06-26.

[179] Jol. https://openjdk.org/projects/code-tools/jol/. Accessed: 2023-05-06.

[180] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 2019.

[181] Raula Gaikovina Kula, Coen De Roover, Daniel M. Germán, Takashi Ishio, and Katsuro Inoue. Modeling library popularity within a software ecosystem. *Tech. Rep.*, 2017.

[182] Hussein Alrubaye and Mohamed Wiem Mkaouer. Automating the detection of third-party java library migration at the function level. *CASCON*, 2018.

[183] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009.

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

[185] Caroline Lima and Andre Hora. What are the characteristics of popular APIs? a large-scale study on java, android, and 165 libraries. *Software Quality Journal*, 2020.

[186] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of API-level refactorings during software evolution. *Proceedings of the 33rd international conference on software engineering*, 2011.

[187] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr, Anh Tuan Nguyen, Miryung Kim, and Tien N Nguyen. A graph-based approach to API usage adaptation. *ACM Sigplan Notices*, 2010.

[188] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 2018.

[189] Rosie Dunford, Quanrong Su, and Ekraj Tamang. The Pareto principle. *The Plymouth Student Scientist*, 2014.

[190] Maven repos. https://mvnrepository.com/repos. Accessed: 2023-11-15.

[191] Gartner Inc. Technology insight for software composition analysis. *Gartner Inc.*, 2023.

[192] Software market revenue. https://www.statista.com/outlook/tmo/software/worldwide. Accessed: 2023-07-26.

[193] Cyber resilience act. https://digital-strategy.ec.europa.eu/en/policies/cyber-resilience-act. Accessed: 2023-07-31.

[194] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. On the effectiveness of machine learning based call graph pruning: An empirical study. *arXiv preprint arXiv:2402.07294*, 2024.

[195] Oxford ecosystem definition. https://www.oed.com/dictionary/ecosystem_n?tab=meaning_and_use#5970695. Accessed: 2023-11-15.

[196] Sonatype. https://www.sonatype.com/. Accessed: 2023-11-15.

[197] Maven requirements. https://central.sonatype.org/publish/requirements/. Accessed: 2023-07-19.

[198] Replication package. https://zenodo.org/doi/10.5281/zenodo.10143429. Accessed: 2023-08-1.

[199] Packaging types. https://www.baeldung.com/maven-packaging-types. Accessed: 2023-11-17.

[200] Maven indexer. https://github.com/apache/maven-indexer/blob/master/indexer-core/src/main/java/org/apache/maven/index/artifact/DefaultArtifactPackagingMapper.java. Accessed: 2023-07-03.

[201] Multi-archive example. https://repo.maven.apache.org/maven2/de/mediathekview/MServer/3.1.60/. Accessed: 2023-07-03.

[202] Checksum discussion. https://www.mail-archive.com/dev@maven.apache.org/msg125281.html. Accessed: 2023-07-19.

[203] Checksum documentation. https://maven.apache.org/resolver/about-checksums.html. Accessed: 2023-07-06.

[204] Maven package: com.crawljax.crawljax-cli. https://repo1.maven.org/maven2/com/crawljax/crawljax-cli/5.1.1/. Accessed: 2023-08-01.

[205] Grace A. Lewis, Ipek Ozkaya, and Xiwei Xu. Software architecture challenges for ml systems. In *International Conference on Software Maintenance and Evolution*, pages 634–638, 2021.

[206] Russ Cox. Surviving software dependencies. *Commun. ACM*, 2019.

[207] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? *9th IEEE Working Conference on Mining Software Repositories (MSR)*, 2012.

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

[209] Donald Pinckney, Federico Cassano, Arjun Guha, Jonathan Bell, Massimiliano Culpo, and Todd Gamblin. Flexible and optimal dependency management via max-smt. *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023.

[210] Apache software foundation. https://www.apache.org/. Accessed: 2023-06-01.

[211] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. Dependency update strategies and package characteristics. *ACM Transactions on Software Engineering and Methodology*, 2023.

[212] Jincheng He, Sitao Min, Kelechi Ogudu, Michael Shoga, Alex Polak, Iordanis Fostiropoulos, Barry Boehm, and Pooyan Behnamghader. The characteristics and impact of uncompilable code changes on software quality evolution. *Proceedings - 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS 2020*, 2020.

[213] Md Atique Reza Chowdhury, Rabe Abdalkareem, Emad Shihab, and Bram Adams. On the untriviality of trivial packages: An empirical study of npm javascript packages. *IEEE Transactions on Software Engineering*, 2021.

[214] Filipe Roseiro Cogo, Gustavo A Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2019.

[215] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. *IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, 2017.

[216] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. *IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[217] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, 2022.

[218] Maelick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. A historical analysis of debian package incompatibilities. *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015.

[219] Raymond Nguyen and Ric Holt. Life and death of software packages: an evolutionary study of debian. *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, 2012.

[220] Israel Herraiz, Emad Shihab, Thanh HD Nguyen, and Ahmed E Hassan. Impact of installation counts on perceived quality: A case study on debian. *2011 18th Working Conference on Reverse Engineering*, 2011.

[221] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. Automatic specialization of third-party java dependencies. *arXiv preprint arXiv:2302.08370*, 2023.

[222] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: obfuscation won't conceal your repackaged app. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[223] Yuru Shao, Xiapu Luo, Chenxiong Qian, Pengfei Zhu, and Lei Zhang. Towards a scalable resource-driven approach for detecting repackaged android applications. *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[224] Reproducible builds projects. https://reproducible-builds.org/who/projects/. Accessed: 2024-03-25.

[225] Guide reproducible builds. https://maven.apache.org/guides/mini/guide-reproducible-builds.html. Accessed: 2024-04-07.

[226] Thanh Le-Cong, Hong Jin Kang, Truong Giang Nguyen, Stefanus Agus Haryono, David Lo, Xuan-Bach D Le, and Quyet Thang Huynh. Autopruner: transformer-based call graph pruning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 520–532, 2022.

[227] Akshay Utture, Shuyang Liu, Christian Gram Kalhauge, and Jens Palsberg. Striking a balance: pruning false-positives from static call graphs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2043–2055, 2022.

[228] A Al-Kaswan and M Izadi. The (ab) use of open source code to train large language models. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, 2023.

# Mehdi Keshani

**Education**

| | |
|---|---|
| 05/2019 – 01/2024 | PhD, Software Engineering Research Group, Delft University of Technology, The Netherlands, Topic: Enhancing the Security of Software Supply Chains: Methods and Practices. Supervisor: Dr. Sebastian Proksch Promoter: Prof. Dr. Arie van Deursen |
| 09/2016 – 12/2018 | MSc, Software Engineering, Sharif University of Technology, Iran, Topic: Cross-project code clones in GitHub. |
| 09/2011 – 05/2016 | BSc, Software Engineering, University of Isfahan, Iran |

**Services**

| | |
|---|---|
| 2019 – 2022 | Member of leading team, FASTEN Project, TU Delft |
| 2019-2023 | Co-reviewer, ICSE, FSE, and ASE conferences |
| 2024 | PC member, NLBSE |
| 2020 | Organizer, A member of ICSE virtualization team |

**Supervision**

| | |
|---|---|
| 2019 - 2023 | Supervisor of twelve undergraduate theses |
| 2019 - 2022 | Co-supervisor of five undergraduate research assistants |
| 2017 | Supervisor of one undergraduate thesis |

**Teaching**

| | |
|---|---|
| 2022 | Teaching Assistant, Release Engineering for Machine Learning by Dr. S. Proksch and Dr. L. Cruz, Graduate course, TU Delft |
| 2018 | Teaching Assistant, Database Design by Dr. A. Heydarnoori, Undergraduate course, Sharif |
| 2017 | Teacher, Operating System Workshop, Undergraduate course, Sharif |
| 2017 | Teaching Assistant, System Analysis and Design by Dr. A. Heydarnoori, Undergraduate course, Sharif |

**Tech. Experience**

| | |
|---|---|
| Coding | JAVA, Python |
| Data analysis | SQL, Redis, RocksDB, Kafka, Python, R |
| AI/Data Science | Python, R |
| System | Linux, Docker, K8 |