TU Delft

Delft University of Technology

Fine-Grained Analysis of Software Supply Chains

Hejderup, J.I.

DOI
10.4233/uuid:0c46d4a2-148f-4661-a196-6be7bcc7b9db

Publication date
2024

Document Version
Final published version

Citation (APA)
Hejderup, J. I. (2024). *Fine-Grained Analysis of Software Supply Chains*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:0c46d4a2-148f-4661-a196-6be7bcc7b9db

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

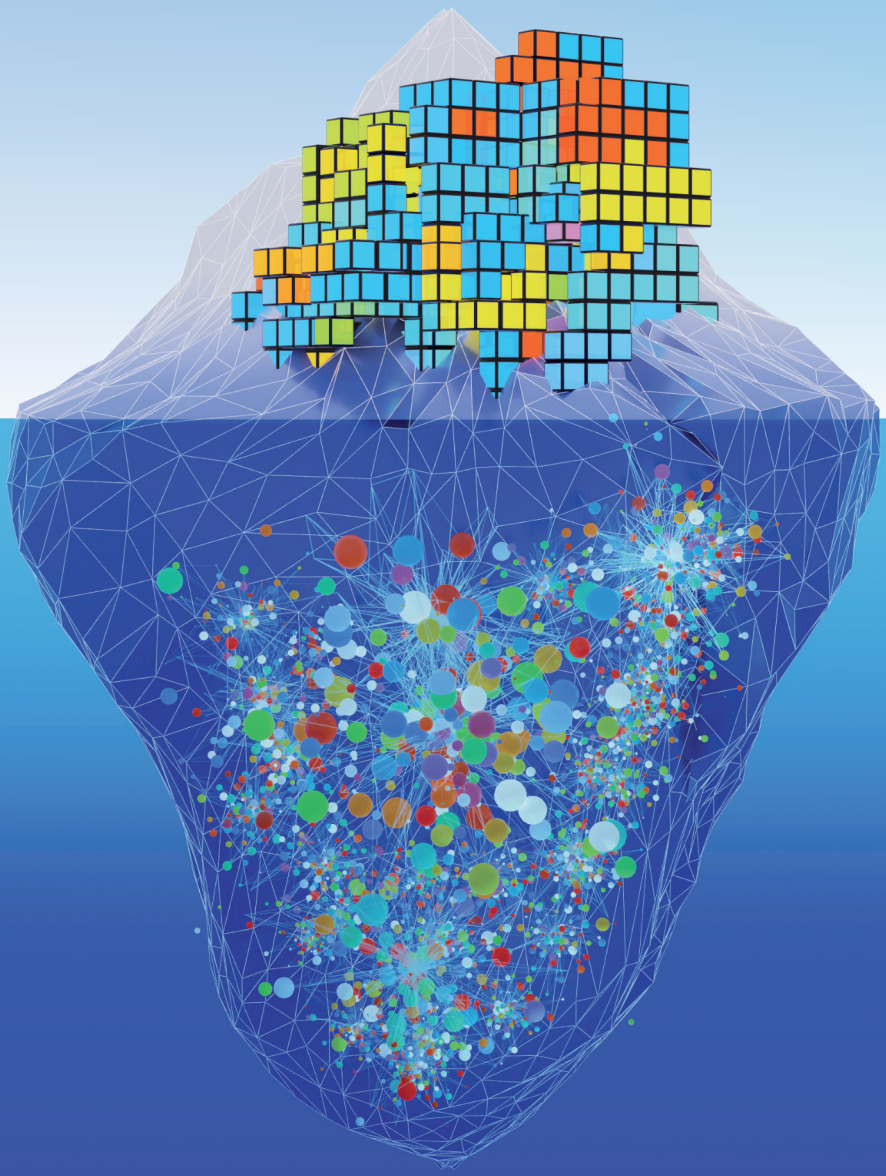
Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Fine-Grained Analysis of Software Supply Chains

Joseph Hejderup



Fine-Grained Analysis of Software Supply Chains

Fine-Grained Analysis of Software Supply Chains

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
Tuesday 21 May 2024 at 15.00 o'clock

by

Joseph Ibrahim HEJDERUP

Master of Science in Computer Science,
Delft University of Technology, the Netherlands
born in Tranemo, Sweden.

This dissertation has been approved by the promotors.

Composition of the doctoral committee:

Rector Magnificus	chairperson
Prof. dr. ir. A. van Deursen	Delft University of Technology, promotor
Dr. ir. G. Gousios	Delft University of Technology, promotor
Independent Members:	
Prof. dr. A. Massacci	Vrije Universiteit Amsterdam
Prof. dr. E. Meijer	Open University of the Netherlands
Prof. dr. A. Møller	Aarhus University, Denmark
Prof. dr. ir. G. Smaragdakis	Delft University of Technology
Dr. J. Bell	Northeastern University, United States of America
Prof. dr. A.E. Zaidman	Delft University of Technology, reserve member

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: Software Supply Chains, Static Analysis, Dependency Management

Printed by: ProefschriftMaken, `www.proefschriftmaken.nl`

Cover: Jacob Hejderup

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.

ISBN 978-94-6469-930-2

An electronic version of this dissertation is available at
`http://repository.tudelft.nl/`.

When you do things from your soul, you feel a river moving in you, a joy.

Rumi

Contents

Summary

Developers strategically reuse code to expedite project development and lower maintenance costs. With the advent of software supply chains, integrating open-source libraries into projects has transitioned from a cumbersome, manual task to an automated, streamlined process. However, this ease of integration has downsides; adding just one library can typically bring in multiple others, each following different coding standards and maintenance protocols. This layer of complexity significantly impedes the effective monitoring and evaluation of security vulnerabilities and breaking changes stemming from these external libraries in software projects. To mitigate these challenges, developers use various tools to oversee and react to software supply chain activities. However, these tools frequently result in a high rate of false positives and are often not insightful, primarily because they rely on metadata from build manifests.

This thesis proposes call-based reachability analysis to enhance comprehension and precision by treating source code as first-class citizens in software supply chain analysis. Initially, we generate function call graphs for package releases, forming a call-based dependency network. We then compare such networks with manifest inferred networks to understand better their similarities and differences in approximating package relationships. The next phase of the thesis explores third-party library updates. Here, we examine the extent to which project test suites cover third-party functionality, followed by applying code-based reachability analysis to augment code areas lacking test coverage. Finally, we investigate the reuse of imported third-party library code in software projects, providing insights for developers and organizations to benchmark and question their reliance on imported third-party code over first-party code.

Our results demonstrate that reachability analysis based on build manifest inferred networks often tends to overestimate transitive package relationships compared to call-based networks, overlooking the specific usage patterns of third-party libraries in projects. Software projects typically utilize only a portion of the functionalities from a third-party library, a pattern that also extends to dependent third-party libraries. Tools relying on manifest-inferred data incorrectly infer that projects use all functionalities of imported third-party libraries. This results in false alarms when identifying software supply chain issues like security vulnerabilities. Similarly, tools that automate third-party library updates also assume that project test suites can detect regressions effectively. Project tests often fail to cover all functionalities from a third-party library, leading to approved updates where changes are unchecked. Finally, while most projects are aware of importing a large number of third-party libraries, we observe that the reuse of imported libraries remains relatively low, raising critical questions about how organizations should strategically balance third-party versus first-party code, especially in light of the risks inherent in software supply chains.

Samenvatting

Ontwikkelaars hergebruiken code om de ontwikkeling van projecten te versnellen en de onderhoudskosten te verlagen. Met de komst van software supply chains is het integreren van open-source bibliotheken in projecten overgegaan van een omslachtige, handmatige taak naar een geautomatiseerd, gestroomlijnd proces. Deze gemakkelijke integratie heeft echter nadelen; het toevoegen van slechts één bibliotheek kan typisch andere bibliotheken met zich meebrengen, elk met verschillende coderingsstandaarden en onderhoudsprotocollen. Deze extra laag van complexiteit belemmert de effectieve monitoring en evaluatie van veiligheidsrisico's en significante veranderingen die voortkomen uit deze externe bibliotheken in softwareprojecten. Om deze uitdagingen het hoofd te bieden, gebruiken ontwikkelaars verschillende hulpmiddelen om toezicht te houden en te reageren op activiteiten binnen software supply chains. Echter, deze hulpmiddelen geven vaak valse positieven en zijn niet inzichtelijk, voornamelijk omdat ze vertrouwen op metadata uit build-manifesten.

In dit proefschrift introduceren we een methode gebaseerd op call-based reachability analysis om het inzicht en de precisie te verbeteren door broncode te behandelen als een essentieel onderdeel in de analyse van software supply chains. We beginnen met het genereren van grafieken van functieaanroepen voor pakketreleases, waarmee we een netwerk van afhankelijkheden vormen gebaseerd op het aanroepen. Vervolgens vergelijken we deze netwerken met de netwerken die zijn afgeleid uit manifesten om hun overeenkomsten en verschillen in het benaderen van pakketrelaties beter te begrijpen. Daarna kijken we naar updates van third-party bibliotheken en onderzoeken we in hoeverre de testpakketten van projecten deze externe functionaliteiten dekken. We passen call-based reachability analysis toe om de dekking van delen van de code die niet goed getest zijn te verbeteren. Tot slot bestuderen we het hergebruik van geïmporteerde code uit third-party bibliotheken, waarmee we ontwikkelaars en organisaties inzichten bieden om te bepalen hoe afhankelijk ze willen zijn van hun eigen code versus externe code.

Onze resultaten tonen aan dat de huidige methoden voor het analyseren van afhankelijkheden vaak te veel nadruk leggen op indirecte relaties tussen pakketten, zonder adequaat te kijken naar hoe bibliotheken werkelijk gebruikt worden in projecten. Projecten gebruiken doorgaans slechts een deel van de functionaliteiten uit een third-party bibliotheek, maar tools die alleen naar de manifesten kijken gaan er ten onrechte vanuit dat alle functionaliteiten worden gebruikt. Dit kan leiden tot onnodige waarschuwingen bij het identificeren van problemen zoals veiligheidsrisico's. Deze tools gaan er ook vaak ten onrechte vanuit dat testpakketten van projecten in staat zijn regressies te detecteren. Tests falen vaak om alle functionaliteiten van een third-party bibliotheek te dekken, wat resulteert in de goedkeuring van updates waarvan de wijzigingen niet worden gecontroleerd. Hoewel de meeste projecten een groot aantal third-party bibliotheken importeren, blijkt uit ons onderzoek dat het werkelijke hergebruik van geïmporteerde code beperkt is. Dit roept vragen op over hoe organisaties een evenwicht kunnen vinden tussen het gebruik

van eigen code en third-party code, vooral in het licht van de risico's die gepaard gaan met software supply chains.

Acknowledgments

After spending a decade at TU Delft, progressing from a master's student to a scientific programmer and ultimately to a doctoral student, I want to take a moment to express heartfelt thanks to everyone who has contributed to my personal and professional growth. Whether your role was large or small, each of you has significantly impacted my journey at TU Delft, for which I am deeply grateful!

Georgios: if there is one thing I vividly remember from my PhD journey, it has been your saying: "No one promised a PhD was going to be easy." With the thesis in my hand, it was a bumpy journey (to say the least), but equally, a journey embracing uncertainty. There have been many moments where I questioned myself about the direction and scope of my research work; you have always been very supportive, giving me the space and time I need and not budging on doing work that can impact practice (even though it meant missing deadlines :p). I am genuinely thankful for this and the other million things you have taught me about technical work, career, and life. More than all this "serious" stuff, the occasional dinners, BBQs, and movie nights at your house with the early SAL group may seem insignificant, but they helped you cool off and enjoy life. You have the most enormous thanks in my PhD journey!

Moritz: To single out the individual most responsible for shaping my academic skills, that honor undoubtedly belongs to you! You taught me a) how to write some decent academic English and b) the essentials of reproducibility & quality assurance 101. I am grateful you were my mentor and helped me quickly grow as a researcher. From hacking Präzi using LLVM to the numerous trips and the interesting personalities we met during those (including the Danish "love glow" man at Elba Island), I have had among the most fun moments with you in my PhD. *Danke Schön Moritz!*

Arie: Whether discussing a thesis internship at UBC or reaching out to a potential collaborator or someone who can provide feedback, from talking about it to executing it, actions are always taken on the spot. I am very thankful that you are practical and, more importantly, quick to utilize your network to explore new avenues whenever I ask. At the same time, I am also very thankful for the opportunity and the ability to continue pursuing the research topic I once explored during my MSc thesis throughout my PhD at SERG.

Kostas, Ilias: I am incredibly grateful to have supervised your projects related to software supply chains. *Kostas*: Your dedication to learning the internals of the Rust compiler and building a call graph generator was instrumental in enabling most of my research work; I cannot thank you enough! *Ilias*: Although you are likely the last to work on one of my crazy ideas, I am equally thankful that you took on this project for your honors program and single-handedly pushed through a very advanced project!

SAL (Amir, Ayushi, Elvan, Enrique, Maria, Mehdi, Wouter): Thank you for all the engaging discussions, lively debates, hearty laughs, delightful food truck lunches, and back-to-back coffee breaks during and outside office hours! A special shoutout to *Amir* for being

a steadfast presence in the office during the COVID years of my PhD; the limited social interaction that was permitted made those challenging times significantly more endurable!

SERG: Being a member of SERG for many years and witnessing its rapid growth in size, it isn't easy to name everyone here, but I would like to thank both former and current members of the group! I feel fortunate to have been surrounded by and hanging out with people from diverse backgrounds, nationalities, and faiths, which has expanded my horizons and enriched my thinking. *Minaksie* and *Kim*, thanks for taking care of every organizational bit of my PhD journey!

Endor Labs: I am deeply grateful to *Dimitri* and *Varun* for giving me the opportunity to continue my work on software supply chain analysis in a dynamic Silicon Valley startup right after my PhD contract ended. Thanks to all the *ewoks*—from engineering to marketing and sales—for broadening my understanding beyond my academic knowledge!

Pind Gang (Aditya, Daniel, Debarshi, Preet, Prashanth, Sangeeth, Soran, Sowmya, Vani): From our casual chats, epic food crawls, festival celebrations to spontaneous weekend trips across countries, you've been more than friends—you've been the family I found in the Netherlands during my years abroad. There have been times when I've been the elusive one, but I hope to be more available and share even more memories moving forward. :)

Mamma, Pappa, Jonas, Jacob: Det har varit en tid fylld av många och väldigt tuffa prövningar under mina år som doktorand. Utan ert stöd, djupa förståelse och kärlek hade ingenting av det jag nu har uppnått varit möjligt. Nu har ni inte bara en son eller bror som är ingenjör, utan också en doktor – om än inte i den traditionella meningen!

Last but not least, I am indebted to my best friend and wife, *Samprajani*; you have been with me in the ups and downs. I am not sure I would have survived this journey without your love and support.

Joseph
Delft, April 2024

1

Introduction

1

Software reuse has become a central theme in software engineering due to its potential to drive efficiencies in time and cost in software development. The practice involves leveraging existing components, such as libraries, frameworks, and design patterns, to reduce the need to build software from scratch. The emergence of software supply chains, which are processes and tools for managing and integrating software components from various sources, has streamlined and scaled the reuse of open-source libraries (also known as packages) across development projects. In turn, this has given birth to vibrant communities that encourage low entry barriers for publication, fostering the development of reusable libraries that others can effortlessly integrate and build upon. As of June 2023,[1] JavaScript's npm registry serves over 3.2 million packages to its users, being the most prominent community, followed by Java's Maven Central, with over 500,000 reusable libraries.

Libraries from software supply chains do not exist in isolation; they often rely on other libraries to function. An intriguing, often overlooked characteristic is that directly importing a few libraries in a development project can unintentionally result in pulling in hundreds of additional libraries. Network analyses of repositories hosting reusable libraries shed light on this characteristic, revealing that a small set of libraries are implicitly dependent on a large number of other libraries, hinting at the scale-free nature of these systems [1, 2]. Removing the `left-pad` package from NPM, a seemingly minor library, was a stark example of this intricate interdependency [3]. Its removal rendered popular packages, powering enterprise websites and applications like `React` and `Babel`, unusable, echoing its hidden importance in software supply chains. This incident highlights one of the critical challenges in managing open-source libraries: the risk of unexpected breaking changes, performance bugs, and security vulnerabilities in indirectly imported libraries, often referred to as transitive dependencies.

The decentralized nature of software supply chains and the lack of uniform development and testing standards make coordinating and managing library incidents a formidable task. Developers have direct control only over the libraries they import, leaving potential issues in indirectly imported libraries largely outside their immediate control. Therefore, the need for continuous monitoring and understanding of software supply chain changes becomes critical for developers and organizations, specifically the ability to prioritize based on the context of reused libraries in their projects. Researchers [1, 4–7] have addressed this by developing techniques and conducting empirical studies on software supply chains utilizing metadata descriptors in open-source projects and libraries. Metadata descriptors, often manifesting as build-instructions, contain information necessary for building projects and detailing imported libraries and their version constraints. By emulating the resolution process of package managers, researchers [1, 8] can construct both graphs of individual libraries and networks of repositories that host them, providing enhanced visibility and enabling wide-reaching impact and reachability analysis. Many state-of-the-art tools also base on this technique; two notable tools are `Dependabot`, an open-source tool, and `Snyk`, a commercial tool.

Despite their usefulness, techniques based on metadata descriptors provide a surface-level understanding of software supply chains, signifying how libraries declare a dependence on each other without giving insight into how the source code incorporates them.

[1] `http://www.modulecounts.com`

Projects do not reuse libraries uniformly, but individual goals and needs dictate the degree of library functionality they reuse. This variance in library usage introduces the likelihood of false positives in analyses when treating the use of libraries uniformly across projects. A simple example of this would be a project declaring dependence on a library and including an unused import statement from that library in the source code. A metadata-based technique would erroneously flag this library as being used in the project.

In this thesis, we aim to advance beyond current metadata-based techniques toward code-based ones. We develop techniques and empirically analyze software supply chains at the granularity of functions and the invocations between them, providing an approximation for reasoning about the reuse of library functionality. Traditionally, program analysis computes approximations of functions and their invocations at the project scope. We seek to build representations of function calls at the scope of package repositories. Given their scale and temporal properties, we initially focus on devising a technique that generates call graphs of individual library releases, which we then stitch into a network given a timestamp *t*. Using this technique, we conduct an empirical study comparing call-based dependency networks with traditional metadata-based networks to understand their similarities, differences, and cost trade-offs.

After analyzing package repositories of software supply chains, we shift our attention to the consumers of these supply chains. Initially, we assess how effectively library updating tools, such as `Dependabot`, can assist projects in avoiding semantically breaking changes. As `Dependabot` relies on the test suites of projects, we construct a complementary change impact analysis to uncover coverage gaps of reused library functionality in projects. Lastly, we delve into understanding how projects effectively reuse third-party libraries to comprehend better the trade-offs between using third-party code in place of first-party code.

1.1 Background and Context

In this section, we delve into the underlying fundamentals of software supply chain modeling techniques and the application of static analysis within this context.

1.1.1 Understanding Software Supply Chains

Converting information between different well-known formats, accessing external storage, manipulating information such as numbers, locations, and dates, or integrating with popular online services are essential operations developers must address in software projects. Unlike the standard library of programming languages, these essential operations evolve in response to technological progress (e.g., the shift from XML to JSON) or to cater to specific user communities (e.g., interfaces to Twitter API or Amazon AWS SDK).

In the face of these evolving demands, modern programming languages such as Java, JavaScript, C#, and Rust host public distribution channels. These dynamic repositories empower developers and organizations to contribute and continually maintain these essential operations as reusable libraries (known as packages). Social coding platforms such as GitHub and Gitlab are critical infrastructures for these distribution channels. These platforms provide a medium for external code contributions to be accepted, scrutinized, and seamlessly integrated into reusable libraries. Maintainers also adopt DevOps practices,

1

```
[package]
name = "my_rust_project"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = "^1.0"
serde_json = ">=1.0, <2.0"
tokio = { version = "^1.0", features = ["full"] }
reqwest = "0.10.10"
```

Figure 1.1: Build Manifest of a Rust Project

including continuous integration, the automatic building and testing of new code changes, and continuous delivery to ensure frequent and regular publishing of new library releases to the distribution channels.

These stages, from contribution to distribution, collectively form a system we recognize as the *software supply chain*. This chain ensures the efficient flow of enhancements and updates to existing libraries and the introduction of new, in-demand functionalities through new libraries to developers and organizations that make use of them.

In practice, developers engage with the software supply chain through the manifest descriptors of build systems. Figure 1.1 exemplifies a simple build manifest for a Rust project. Here, the package section (`[package]`) establishes the name and version of the project, serving as its identifier for publishing to a distribution channel. Conversely, the dependencies section (`[dependenices]`) enumerates the libraries a project reuses in the source code, accompanied by compatibility constraints (e.g., >=1.0, <2.0). When a build system processes the dependency section of a manifest, it resolves the compatibility constraints and subsequently makes the libraries available in a developer's workspace. The compatibility constraints further enable projects to take advantage of new enhancements and security fixes by automatically updating them to the latest compatible version. However, unlike DevOps environments, where code changes undergo validation, library updates in build systems are applied immediately without similar checks, such as running integration tests. This unverified immediate application of updates introduces potential risks, as breaking changes could be inadvertently incorporated into projects.

As shown in Figure 1.1, our example project utilizes four reusable libraries: `serde`, `serde_json`, `tokio`, and `reqwest`. These libraries, in turn, also build upon other pre-existing libraries from the distribution, forming what we refer to as dependencies. While the project explicitly relies on these four libraries, it also implicitly incorporates an additional 133 libraries necessary for its successful compilation. These 133 libraries, like the four declared libraries, also originate from open-source maintainers with diverse development and test conventions and varying release frequencies. Moreover, the tendency to import a large number of libraries is common in projects, as distribution channels often encourage the publication of specialized and modular libraries. This strategy, designed to foster simplicity and manageability, helps developers create complex systems by integrating diverse, task-specific libraries as cohesive building blocks. Illustrating the scale of this practice, JavaScript's npm registry, as the most prominent community, serves over 3.2 million pack-

```
my_rust_project v0.1.0
|-- reqwest v0.10.10
|   |-- base64 v0.13.1
|   |-- bytes v0.5.6
|   |-- futures-util v0.3.28
|   |-- http v0.2.9
|   |-- http-body v0.3.1
|   |-- hyper v0.13.10
|   |   |-- bytes v0.5.6
|   |   |-- futures-channel v0.3.28
|   |   |-- futures-core v0.3.28
|   |   |-- tokio v0.2.25
|   |   |-- tower-service v0.3.2
|   |   |-- want v0.3.0
|   |-- hyper-tls v0.4.3
|-- tokio v1.28.2
|   |-- bytes v1.4.0
|   |-- libc v0.2.144
|   |-- mio v0.8.6
|   |-- signal-hook-registry v1.4.1
|   |-- socket2 v0.4.9
|   |-- tokio-macros v2.1.0 (proc-macro)
|-- serde v1.0.163
|-- serde_json v1.0.96
```

Figure 1.2: Truncated dependency tree of a Rust project

ages to its users as of June 2023. Java's Maven Central follows next, offering over 500,000 reusable libraries.[2]

Figure 1.2 visualizes a truncated version of the project's dependency tree. Examining this tree, we can observe both explicit and implicit relationships of packages the project imports. For example, the project has a transitive dependency on `bytes v0.5.6` through `hyper v0.13.10` and `reqwest v0.10.10`. Interestingly, the `bytes` package makes multiple appearances, one under `tokio v1.28.2`. This recurrence is a strategy to circumvent conflicts when no single major version of `bytes` fulfills the requirements of all packages. This strategy empowers developers and organizations to scale their reuse of libraries, enabling them to leverage any available library without worrying about compatibility conflicts.

1.1.2 Network Analysis of Software Supply Chains

Contrary to the exploration and analysis of individual software systems, the backbone of analyzing software supply chains is rooted in network analysis. Network analysis is a statistical and graph theory method to investigate social structures using networks and graph theory [9]. It involves characterizing and modeling a network to identify patterns, connections, and critical nodes or influential points. This method provides the tools to quantitatively measure and visualize the complexity of the relationships and interactions among software packages.

Software supply chains represent an intricate, continuously evolving socio-technical

[2] `http://www.modulecounts.com`

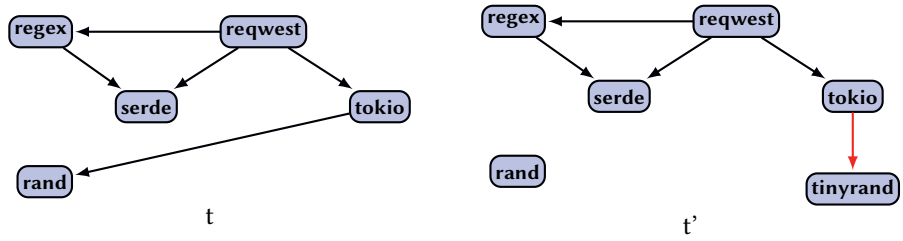


Figure 1.3: Network of a simplified software supply chain at timestamp *t* and *t'*

network, encapsulating relationships between libraries and among the diverse community of open-source developers. Network analysis is instrumental in capturing these relationships and effectively tracking changes propagating upstream (i.e., dependencies of a project) and downstream (i.e., dependents of a project) within a network, providing a holistic perspective of the supply chain and individual libraries. Examples of upstream applications are detecting known vulnerabilities, license violations, and breaking changes. On the other hand, downstream applications have received more significant attention among the research community, with analyses focusing on trend analysis, risk assessment, and stability. Examples of these applications include identifying highly popular or dormant libraries with extensive dependencies within the network (i.e., finding the central nodes of a software supply chain).

Given a distribution with five published packages — `reqwest`, `regex`, `serde`, `tokio`, and `rand` — the typical method to infer a network, also known as a package dependency network (PDN), involves parsing all dependencies and determining their constraints based on the build manifests from the distribution. The left-hand side of Figure 1.3 presents the inferred network of this distribution. We observe that the network forms a single component, with `reqwest` depending on each of the other packages, either directly or indirectly. All packages except `serde` and `rand` have dependencies, while all except `reqwest` have dependents.

Consequently, in the event of a vulnerability or severe bug in any package, `reqwest` would be affected due to its reachability to all packages. For instance, if `rand` were to have a vulnerability, it would impact `tokio` and `reqwest`, affecting 50% of all packages. However, if `reqwest` had a vulnerability or a severe issue, it would not affect any other package in the distribution. This inference represents an essential aspect of impact analysis or reachability analysis, where we assess the number of packages impacted by a change in a distant package either in a downstream or upstream direction. It is also a measure of network stability; if a package with multiple dependents, such as `tokio`, becomes unstable, it would become a single point of failure in the supply chain.

At the timestamp marked as *t'* in the right-hand side of Figure 1.3, we find that `tokio` releases a new version, introducing a new package into the distribution: `tinyrand`. Once we re-infer the network, we suddenly have two distinct components. This change comes as `tokio` replaces `rand` with `tinyrand`, thus isolating `rand` in the network. The network structure can evolve rapidly due to some packages' high frequency of releases. Thus, continuous monitoring of these changes, understanding their implications, and evaluating

their impact on network stability is crucial to identifying emerging trends and measuring the stability of the software supply chain.

1.2 The Need for Source Code-Focused Representations

Given the increasing dependence on third-party libraries in software projects, developers and organizations find it essential to have tooling that can monitor, optimize, and manage their reuse of libraries from the software supply chains. Consider, for example, the simple Rust project with its 135 imported libraries in Section 1.1.1. Monitoring and keeping these libraries up-to-date manually is a tedious and costly task in software development.

While the build system can assist projects by automatically updating to the latest compatible version of explicitly and implicitly reused libraries, the process is rarely seamless, as highlighted in Figure 1.4. Despite guidelines like the semantic versioning scheme [10] that assist library maintainers in orderly versioning code changes, breaking changes between supposedly compatible versions is prevalent [11]. In response, developers often resort to version pinning (i.e., disabling updates) or deploy tools such as `Dependabot` or `Renovate`. These tools emulate a DevOps experience for library updates, using continuous integration as a prerequisite for approving a library update. While these mitigation techniques offer stability, they also have associated problems. Avoiding updates through version pinning for an extended period incurs technical debt, making migrations to newer versions challenging, mainly if a security vulnerability arises. In addition, continuous integration testing may inadvertently approve updates, as developers may not necessarily write tests for the libraries they use.

Software supply chains are becoming a popular target for malicious activities, including tactics such as taking over maintenance of popular packages to hide malicious content like a Bitcoin wallet stealer inside the source code [12], typosquatting where malicious packages have similar names to popular packages (e.g., `urllib` instead of `urllib3`) [13], and attack vectors like poor input sanitization in packages. Despite using reachability analysis in security monitoring tools such as OWASP Dependency Checker, GitHub Security, and Snyk, many alerts, as shown in Figure 1.5, are false alarms.

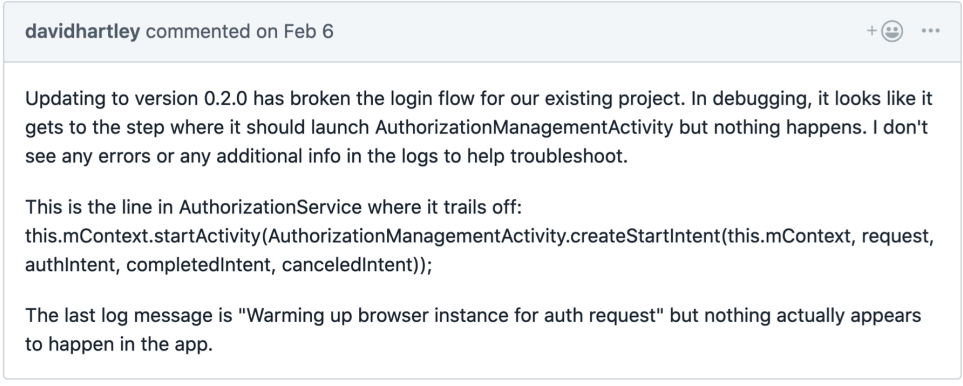


Figure 1.4: Update failure in `okta/okta-sdk-appauth-android`, #81

1

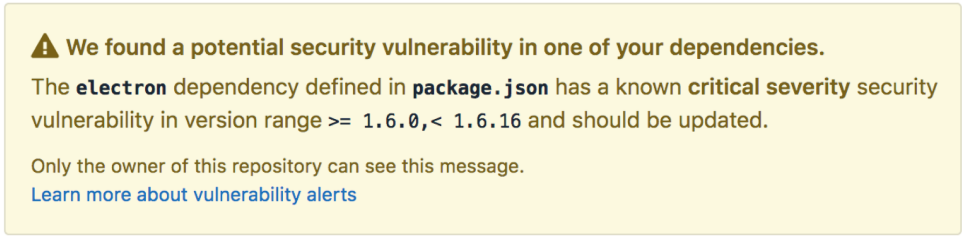


Figure 1.5: False Alarm of Security Vulnerability

Without knowing how packages reuse libraries, developers have a limited understanding of how vulnerabilities or changes in imported libraries affect their code. Increasingly, package repository workgroups such as the Rust Ecosystem WG [14] are also calling for a more comprehensive network analysis of package repositories to support code-centric analysis for more effective identification of critical yet unstable packages [15, 16]. One such example is the `Libz Blitz` [17] initiative, where community members contribute to poorly maintained yet critical packages in Crates.io to stabilize highly reused code in the distribution.

1.3 Research Goals and Questions

As software projects increasingly leverage software supply chains to maximize opportunities for code reuse and reduce development costs, the need for enhanced comprehension, evaluation, and management of these chains, ensuring minimized disruptions and stability in the supply chain, also grows. Existing methodologies and tools, which lean heavily on data derived from build manifests, often result in developers having to sift through false alarms. This issue further complicates the process of pinpointing supply chain-related problems in their use of libraries and source code, introducing costly maintenance burdens. To meet these challenges, we introduce research questions focused on exploring the efficacy and application of program analysis techniques within software supply chains:

RQ 1 How feasible is code-based reachability analysis in practice?

Here, we evaluate the practicality of developing a code-based model for conducting reachability analysis within package repositories. Our primary objectives include assessing the feasibility of generating a code-based representation for each release in a package repository and its associated computation time.

RQ 2 How does code-based reachability analysis complement tests in third-party library updates?

Building on **RQ1**, this research question explores how reachability analysis complements project test suites during third-party library updates, an essential software supply chain task. We will assess project test suites' effectiveness in identifying regression-like changes in these libraries. The focus is on how reachability analysis might address coverage gaps, particularly in detecting changed third-party functionality that project test suites might miss.

RQ 3 How do software projects reuse imported third-party libraries?

In contrast to the previous research questions, **RQ3** uses code-based reachability to quantify code reuse. The aim is to analyze third-party code adoption and utilization within open-source projects empirically. The objective is to provide insights that assist developers and organizations in balancing third-party library code use with in-house development, considering the maintenance and security implications of relying on software supply chains.

This thesis applies code-based reachability analysis to address software supply chain problems, focusing on enhancing the safety of third-party library updates and understanding code reuse of third-party libraries.

1.4 Research Methodology

The principal methodological approach of this thesis is Design Science Research (DSR), a discipline within information sciences that focuses on creating and refining artifacts aimed at enhancing their performance and efficacy [18, 19]. In **RQ1** and **RQ2**, we address two technical problems by developing artifacts for 1) detecting software supply chain issues (e.g., security vulnerabilities) in third-party libraries and 2) safely updating third-party libraries in projects. We establish three efficacy requirements to evaluate our solution:

1. **Scalability**: The analysis must handle the vast scope of package distributions, ranging from tens of thousands to over a million, with timely execution.
2. **Coverage**: The approach should include all necessary packages to ensure comprehensive analysis and prevent incomplete assessments.
3. **Accuracy**: It is critical to accurately infer relationships between packages to identify potential vulnerabilities and secure updates effectively.

These requirements guide our iterative process and evaluation, particularly emphasized in Chapters 2 and 4, where DSR principles are actively applied to develop and refine the respective artifacts. We also generate a large corpus of code-based representations of package releases as a result of developing and evaluating methods for code-based reachability analysis. This data facilitates the study of code reuse patterns and trends in third-party libraries within software projects. Therefore, in Chapter 3 and 5, we use Case Study Research (CSR) as our supplementary methodological framework. We utilize empirical

Table 1.1: Overview of Research Methodology per Chapter

Artefact/Study	Chapter	DSR	CSR
Präzi for call-based dependency networks	2	✔	
Comparative network analysis study of Crates.io	3		✔
Uppdatera for static update checking	4	✔	
Code Reuse patterns in Java	5		✔

1

Table 1.2: Overview of Replication Packages per Chapter

Package	Chapter	Zenodo DOI
Präzi v1.0 tool & dataset	2	10.5281/zenodo.8152060 [22]
Präzi v2.0 tool & dataset	2,3	10.5281/zenodo.4478981 [23]
Uppdatera tool, infrastructure & dataset	4	10.5281/zenodo.4479015 [24]
Code Reuse Notebooks & Dataset	5	10.5281/zenodo.7874912 [25]

methods [20, 21] to gain insights into patterns and trends of third-party library code reuse, focusing on function call and dispatch types and on import and reuse patterns of lines of code.

The synthesis of DSR and CSR in this thesis not only addresses the specific technical challenges posed in each research question but also contributes to a broader understanding of software supply chain dynamics. Table 1.1 provides a comprehensive overview of our research methodology, organized by chapter.

All tools, datasets, and infrastructure used in the research are publicly available, except for specific industry-based tools where confidentiality is necessary. Research implementations are accessible on GitHub, and we offer replication packages for each tool and study on Zenodo. Table 1.2 shows an overview of replication packages for relevant chapters.

1.5 Research Outline

This thesis is a compilation of various independent articles. These articles have been modified, and in some instances split, to form a unified thesis. However, we have preserved their fundamental structure to facilitate straightforward correspondence between the chapters and their papers. Table 1.3 connects each research question to relevant chapters. In the remainder of this section, we detail these chapters:

Chapter 2 introduces a novel approach called Präzi, which integrates build manifests with call graphs to create a call-based dependency network applied to the Crates.io repository. Through large-scale compilation, Präzi produces a call-based dependency network (CDN) that encapsulates 90% of all compilable packages, making it a viable option for conducting large-scale empirical studies of package distributions. A comparative evaluation

Table 1.3: Overview of Research Questions and corresponding Chapters

RQ	Research Question	Chapter
1	How feasible is code-based reachability analysis in practice?	2, 3, 4
2	How does code-based reachability analysis complement tests in third-party library updates?	4
3	How do software projects reuse imported third-party libraries?	5

shows the CDN to be 3.3 times more precise than a package-based dependency network (PDN) in reachability analysis. The accuracy of the call-graph generator in resolving language features, however, can affect the overall accuracy of the network in specific use cases. A CDN further enables new fine-grained applications, such as tracking code bloat and monitoring deprecation, which is not possible with metadata-based networks.

Chapter 3 explores the structure and evolution of Crates.io by examining the similarities and differences between metadata, compile-validated metadata, and call-based networks in a large-scale empirical study. While the three networks have similar efficacy for direct dependency analyses, significant disparities emerge in transitive dependency analyses. Here, the advantage of call-based networks becomes clear: they offer a more detailed, code-centric analysis that captures the actual usage context of package dependencies.

Chapter 4 examines the reliability of test suites in automated dependency update services, such as `Dependabot`, using 521 well-tested Java projects as a case study. The study finds that tests cover only 58% of direct and 20% of transitive dependency calls and detect only 47% of direct and 35% of indirect faults on average in over 1.1 million artificial updates with simple faults. A proposed change impact analysis tool enhances the detection rate to 74% for direct and 64% for transitive dependency faults, nearly doubling the detection efficacy of traditional test suites. In a study of 22 real-world dependency updates, we find static analysis complements test suites due to a lack of test coverage. The research underscores the need for better test practices around third-party library reuse, emphasizing the risks of relying solely on project tests for compatibility checks in automated dependency updates. The chapter concludes by advocating for adopting hybrid techniques that combine static and dynamic analyses to address gaps in test coverage and enhance the reliability of dependency updates.

Chapter 5 studies the usage of third-party libraries in 176 Java projects from the Census-II dataset across 3,182 releases. It reveals that despite third-party libraries accounting for 87% of the projects' code, only 12-38% of this external code is reused. Most third-party code remains untouched in projects, indicating potentially high operational and maintainability risks. Notably, this trend of low reuse stays relatively stable over time, despite a modest increase in third-party code usage overall. These findings emphasize the need for more detailed evaluations of dependency usage in software development to mitigate unnecessary risks and provide a baseline to evaluate how projects reuse code from third-party libraries.

1.6 Origin of Chapters

All chapters of this thesis, except for Chapter 5 currently under submission, have been published in peer-reviewed journals and conferences. **Chapter 2** includes an additional extensive manual evaluation that has not been peer-reviewed. Each chapter in this thesis is self-contained, comprising its background, related work, and implications sections.

- **Chapter 2** was published in the paper *Präzi: from package-based to call-based dependency networks* by Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios at the Empirical Software Engineering (EMSE) journal in 2022. This chapter also contains content from the paper *Software Ecosystem Call Graph for Dependency Management* by Joseph Hejderup, Arie van Deursen, and Georgios

Gousios at the International Conference on Software Engineering (ICSE) 2018 The New Ideas and Emerging Results (NIER) track.

- **Chapter 3** was published in the paper *Präzi: from package-based to call-based dependency networks* by Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios at the Empirical Software Engineering (EMSE) journal in 2022.
- **Chapter 4** was published in the paper *Can we trust tests to automate dependency updates? A case study of Java Projects* by Joseph Hejderup and Georgios Gousios at The Journal of Systems & Software (JSS) in 2022.
- **Chapter 5** is under submission in the paper *Evaluating the Impact of Third-Party Library Reuse in Java Projects: An Empirical Study* by Joseph Hejderup, Anand Sawant, Henrik Plate and Georgios Gousios at The Journal of Systems & Software (JSS) in 2023.

2

Präzi: From Package-based to Precise Call-based Dependency Network Analyses

This chapter presents Präzi, an approach that integrates manifest data with package call graphs, creating a nuanced, function-level dependency network. Unlike previous studies, Präzi analyzes actual usage of dependencies in source code, offering a more accurate representation. We implemented Präzi on Crates.io, a repository requiring large-scale compilation, yielding a call-based dependency network that encapsulates 90% of all compilable packages (70% of indexed packages). Our manual examination of discrepancies between metadata and call-based dependency networks revealed potential improvements in 133 cases out of 381 package relationships. Furthermore, our reachability analysis focusing on security and deprecated functions demonstrated that a call-based representation is 3.3 times more precise than a PDN-based representation. These findings underscore the feasibility of implementing a CDN for a package repository and the benefits of function-based representation while highlighting the need to carefully consider how a call-graph generator treats language features.

Modern programming languages like Java, JavaScript, and Rust promote software reuse by maintaining extensive, rapidly evolving repositories of highly interdependent packages or reusable libraries. Typically, researchers parse build manifest data to infer a package dependency network, which helps answer pivotal questions like "How many packages depend on packages with known security issues?" or "What are the most used packages?". Nevertheless, these studies often overlook a critical aspect: the necessity of examining the actual usage of these dependencies in the source code beyond manifest-inferred relationships.

In this chapter, we introduce Präzi, a method that combines build manifest data with call graphs of packages, enabling us to infer a more fine-grained dependency network at the function level. We chose to implement Präzi for Crates.io for two main reasons. Unlike repositories like Maven Central, which host analyzable binaries, Crates.io requires large-scale compilation of all its releases to produce the binaries necessary for call graph

generation. Furthermore, the approximations provided by program analysis techniques can erroneously add or exclude edges between packages.

Our primary goal is to evaluate the feasibility of inferring call-based dependency networks and the time to construct one, thereby assessing their practicality as a possible alternative or supplement to metadata-based networks. The inability to compile and build a call graph for the majority of Crates.io releases could lead to an incomplete representation, rendering it impractical for large-scale empirical studies or analyses. Creating a ground truth for a real-world package repository is unfeasible; hence, we devise an evaluation strategy that establishes a partial ground truth by manually examining discrepancies between metadata-based and call-based dependency networks. These discrepancies shed light on the accuracy of either network in correctly inferring relationships between packages.

Our work indicates that our Call-based Dependency Network (CDN) encompasses 90% of all compilable packages, nearly encapsulating the entire Crates.io repository. We manually established a ground truth for 381 package relationships by constructing a corresponding metadata-based network for comparison. In 133 of these cases, we found that a CDN could provide improvements. We also determined that conventional call graph generators may not fully support language-specific features, such as uninstantiated generics or conditionally-compiled functions, necessary for sound inferences or specialized use cases of CDNs. Finally, we conducted a comparative evaluation of reachability analyses focused on security scanning for known vulnerabilities. This analysis revealed that, when it comes to security vulnerabilities, the precision of reachability is 3.3 times higher when using a CDN-based representation compared to a PDN-based representation.

Our findings suggest that implementing a CDN for a package repository is feasible, with our CDN nearly fully representing Crates.io. However, it is crucial to consider how accurately a call-graph generator handles language features, as this could impact the approximation of package relationships. A function-based representation brings several benefits, notably improving the precision of reachability analysis for existing use cases like security and offering new insights into language-based mechanisms such as deprecated methods.

2.1 Background

2.1.1 Related Work

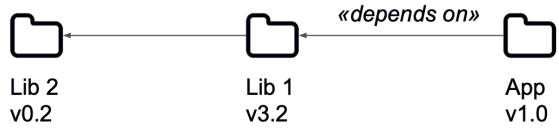
Analyzing package repositories from a network perspective has become an important research area in light of numerous incidents such as the removal of the `left-pad` package in npm and recent moves to emulate such problems on package dependency networks [1, 26–28]. The aftermath of the `left-pad` incident [3] in 2016 raised questions on how the removal of a single 11 LOC package downloaded over 575,000 times could break the build for large groups of seemingly unrelated packages in npm. To understand how certain packages exhibit such a large degree of influence in package repositories, Kikas *et. al.,* [1]'s network analysis of three package repositories—npm, Crates.io, and RubyGems—uncovered that package repositories have scale-free network properties [29]. As a result of a large number of end-user applications depending on a popular set of packages (such as the `babel` compiler), these popular yet distinct packages become hubs in package dependency

networks. Packages that act as hubs are not isolated packages; they also depend on small and common utility packages such as `left-pad` that appear as transitive dependencies for end-users. By reversing the direction of package dependency networks, Kikas *et. al.,* [1] identify that utility packages are highly central in package dependency networks with the power to affect more than 30% of all packages in the studied repositories.

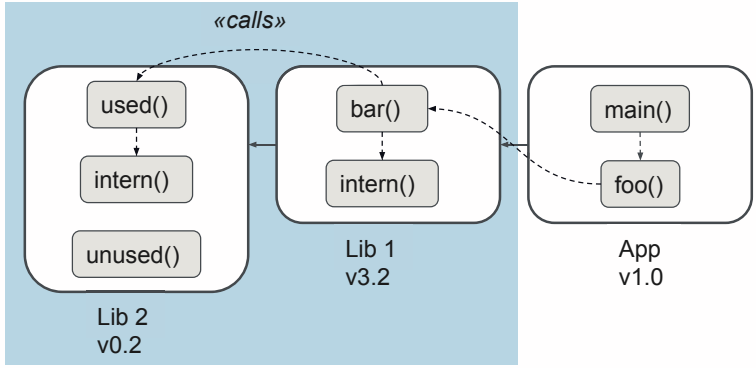
In a comprehensive study of the evolution of package repositories, Decan *et. al.,* [27] observe that three out of seven studied repositories have superlinear growth of transitive relationships, forming and strengthening new network hubs over time. Half of the packages in Crates.io, npm, and, NuGet had in 2017 at least 41, 21, and 27 transitive dependencies, nearly two times more than their respective number in 2015. Although Decan *et. al.,* [27] finds that the number of dependencies a developer declares in an application remains stable over time, the increasing number of transitive relationships in package repositories is still an active phenomenon after the `left-pad` incident. Apart from understanding the structure and evolution of package repositories, researchers have also studied known security vulnerabilities [7, 30], maintainability [28, 31, 32], software reuse [5, 33], and more recently breaking changes [34, 35] from a network perspective. Zimmermann *et. al.,* [7] report that 40% of npm include a package with a known vulnerability, suggesting that npm forms a large attack surface for hackers to exploit. Despite developer awareness on using trivial and simple packages after the `left-pad` incident, Abdalkareem *et. al.,* [33] still find a prevalent number of applications depending on trivial packages: 10% of npm and 6% of PyPI applications on GitHub depends on at least one package with less than 35 LOC.

Network analysis of packages commonly makes use of metadata from package manifests to calculate the impact and severity of measured variables. Ponta *et. al.,* [36]'s work on building a security dependency checker using call graphs highlights the limitations of using metadata and the importance of studying package dependencies with a contextual lens. Typically, a subset of an API is vulnerable—not the entire package—and how clients interact with API's is also highly contextual. Zapata *et. al.,* [37] observed through manual analysis that 75% of 60 warned JavaScript projects did not invoke the vulnerability. As an alternative to vulnerability detection through call graphs, Chinthanet *et. al.,* [38] explores the idea of building hierarchical structures of applications and their dependencies for Node.js. To pitch for code-centric instead of metadata-based representations of package repositories, Hejderup *et. al.,* [39] propose dependency networks based on function calls which we concretize in this work.

By embedding function call relationships into package dependency networks, we aim to also bridge the gap between API and package repository research. Notably, Präzi could resolve the limitation of studying immediate API calls to include chains of API calls (i.e., transitive calls) such as in [40]'s work on determining the ripple effects on deprecated APIs in the Smalltalk ecosystem. Similarly, combining qualitative studies such as looking into deprecation [41, 42], breakages [11, 16, 43], and migration patterns [44, 45] with network analyses could provide an additional empirical dimension in such studies. In support of this, Zhang *et. al.,* [46]'s need-finding study calls for tooling that supports API designers with data-driven recommendations, for example, on when to deprecate an API.



(a) State of the art: Package-based Dependency Networks.



(b) Our proposal: Call-based Dependency Networks.

Figure 2.1: Different granularities of dependency networks.

2.1.2 Rust Programming Language

Rust is a relatively new (first stable release 1.0 in 2015)[1] systems programming language that aims to combine the speed of C with the memory safety guarantees of a garbage-collected language such as Java. Rust is also unique because its package management system (Cargo) was designed from the ground-up to be part of the language environment [47]. Cargo not only manages dependencies but prescribes a build process and a standardized repository layout which helps facilitate the creation of automated large-scale analyses such as Präzi. Every Cargo package contains a file called `Cargo.toml`, specifying dependencies on external packages. Moreover, with Crates.io, there is one central place where all Rust packages (so-called "crates") live. As of 13 August 2020, Crates.io is the fifth most fast-growing package repository hosting over 44,745 packages and averaging 60 new packages per day.[2]

2.1.3 Call-based Dependency Networks

We distinguish two kinds of dependency networks, shown in Figure 2.1: i) coarse-grained *Package-based Dependency Networks* shown in Figure 2.1a, similar to what dependency resolution tools (e.g., Cargo or Maven) build internally or what researchers have used in the past, and ii) fine-grained *Call-based Dependency Networks* shown in Figure 2.1b, which we advocate in this paper.

[1] `https://blog.rust-lang.org/2015/05/15/Rust-1.0.html`
[2] `http://www.modulecounts.com/`

Figure 2.1a models an example of an end user application `App`, which directly depends on `Lib1` and transitively depends on `Lib2`. In such a PDN, each node represents a versioned package. An edge connecting two nodes denotes that one package imports the other, for example `App 1.0` depends on `Lib1 3.2`.

Figure 2.1b consists of three individual call graphs for `App`, `Lib1`, and `Lib2`. Each of these call graphs approximate internal function calls in a single package. Every node represents a function by its name. The edges approximate the calling relationship between functions, e.g., from `main()` to `foo()` within `App` in Figure 2.1b. However, the function identifiers bear no version, nor do they have globally unique identifiers (e.g., there are two `intern()` functions in Figure 2.1b). We merge these two graph representations to produce a CDN:

Definition 1. *A **Call-based Dependency Network** (CDN) is a directed graph $G = \langle V, E \rangle$ where:*

1. *V is a set of versioned functions. Each $v \in V$ is a tuple $\langle$`id,ver`$\rangle$, where `id` is a unique function identifier and `ver` is a float value depicting the version of the package in which `id` resides.*
2. *E is a set of edges that connect functions. Each $\langle v_1, v_2 \rangle \in E$ represents a function call from v_1 to v_2.*

Applying the above definitions, the function `used()` in Figure 2.1b becomes a node with the fully qualified identifier $\langle$`Lib2::used, 0.2`$\rangle \in V$. The dependency between `App` and `Lib1` is represented as $\langle\langle$`App::foo, 0.1`$\rangle, \langle$`Lib1::bar, 3.2`$\rangle\rangle \in E$.

CDNs offer a white-box view of the more coarse-grained PDNs. In particular, we can see that `unused()` is never called. If `unused()` was affected by a vulnerability, we can deduce from Figure 2.1b that we should *not* issue a security warning for `App`, since it does not use the affected functionality. In contrast to the CDN, the PDN in Figure 2.1 by its nature cannot provide such a fine-grained precision level.

2.2 Präzi: Generating CDNs from Package Repositories

In this section, we describe a generic approach, Präzi, to systematically construct CDNs for package repositories. Präzi can be applied to any programming environment that features i) a way of expressing dependency information between packages, and ii) tooling to generate call graphs for a package.

Präzi constructs a CDN in a two-phase process illustrated in Figure 4.1. In the first phase, *Call Graph Generation* (step 1, 2, and 3), Präzi generates a static dataset of annotated call graphs from packages in a repository. In the second phase, *Temporal Network Generation* (step 4 and 5), Präzi first generates an intermediate package dependency network by resolving dependencies between packages at a user-provided timestamp t, and then unifies the call graphs of resolved packages into one temporal call-based network, the CDN_t.

2.2.1 Call Graph Generation

Local Mirror Package managers keep an updatable index of package repositories to lookup available packages and their versions. Präzi uses such indices to extract and down-

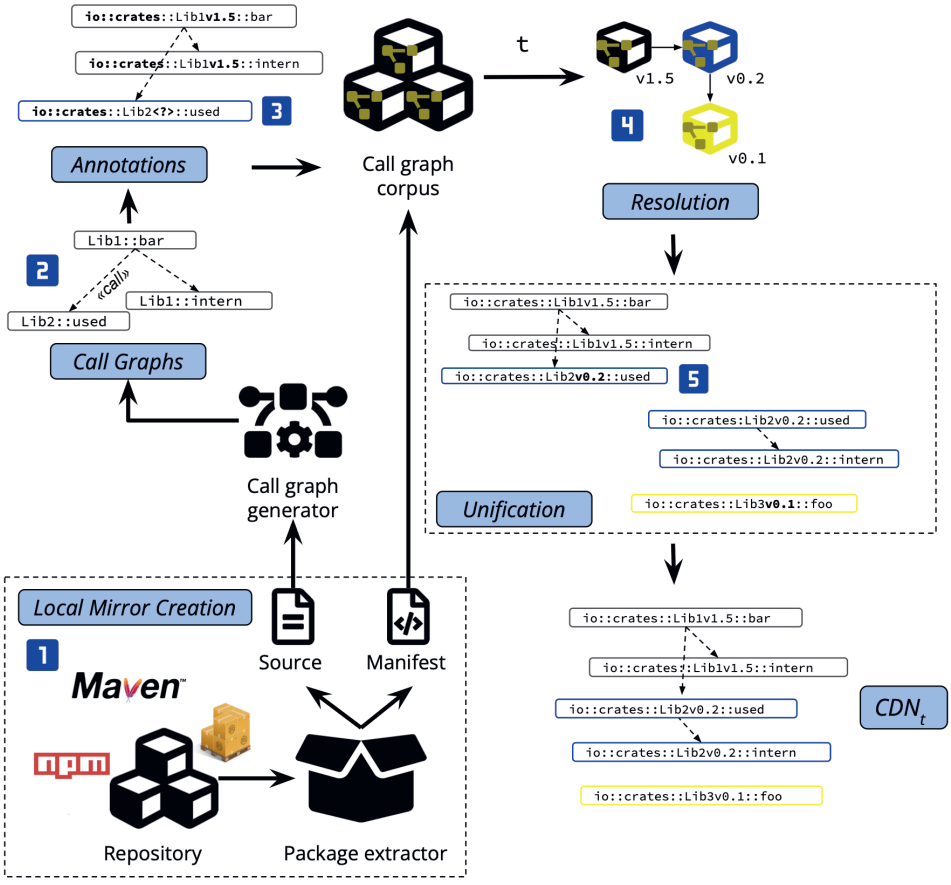


Figure 2.2: Generic approach to generate CDNs from package repositories

load available packages in step 1 in order to create local mirrors of repositories (i.e., clones of repositories). A minimal local mirror needs to contain the manifest and publication (or creation) timestamp for each version of a package.

Package Call Graphs A call graph is a data representation of relationships between functions in a program and serves as a high-level approximation of its runtime behavior [48, 49]. From a static analysis perspective, a call graph is useful for investigating and understanding interprocedural communication between code elements (i.e., how functions exchange information). In Präzi, we view a call graph as a partial graph of a resulting CDN. We increase the scope of a call graph from a single package (i.e., program) to a package and its dependencies. We denote inter-package function relationships as the actual specific code resources that packages use between each other (i.e., a dependency relationship at the function granularity) and are first-class citizens in CDN analyses. The call from `Lib1::bar` to `Lib2::used` in 2 exemplifies an inter-package function relationship. Präzi

requires nodes in call graphs to have function identifiers with fully resolved return types and arguments.

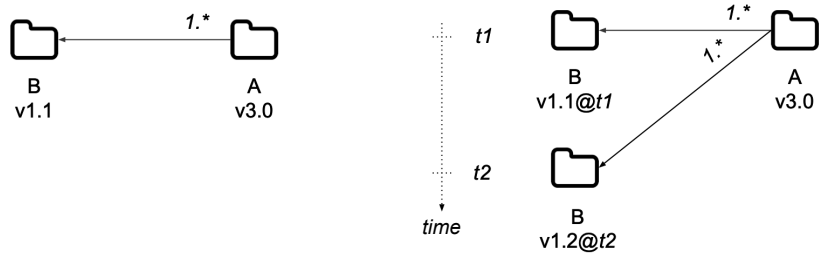
In the presence of dynamic features, such as virtual dispatch or reflection, there are implications to the precision and soundness of call graphs that indirectly also affect generated CDNs. Theoretically, it is impossible to have both a precise and sound call graph of a program. Thus, Präzi uses *soundy* call graph algorithms that follow a best-effort approach for the resolution of most language features [50]. Precise yet unsound call graph algorithms could miss actual inter-package function calls, making certain dependency analyses (e.g., security) of CDNs incomplete. Examples of soundy call graph algorithms for typed languages include subclasses of Class Hierarchy Analysis (CHA) [51, 52] and Points-to analyses [53–55] such as k-CFA [56]. In the case of untyped languages such as Python or JavaScript, a middle-ground is hybrid approaches combining both dynamic analysis and static analysis such as Alimadadi *et. al.,* [57]'s Tochal or Salis *et. al.,* [58]'s PyCG.

Annotating Call Graphs To prepare call graphs for unification, we need to rewrite function identifiers in each package call graph so that they are globally unique. Without annotating function identifiers, inconsistencies can arise from packages that have identical namespaces and multiple versions of the same package in a dependency tree. Präzi solves these issues by annotating the function names, return types, and argument types in function signatures with three components: i) repository name, ii) package name, and iii) static or dynamic (i.e., constraint) package version.

For each function signature in a call graph of a package version, Präzi maps each type identifier found in the signature to the package that declares it. There are three potential mappings of a type identifier to a package that do not reside in the standard library of the language:

- **Local**, resulting in an annotated qualifier with the repository name, and its package name and version as exemplified in `io::crates::Lib1v1.5::bar`.
- **Dependency with a static version**, resulting in an annotated qualifier with the repository name, and the name and version of the dependency.
- **Dependency with a dynamic version**, resulting in an annotated qualifier with the repository name and name of the dependency. However the version is missing as exemplified in `io::crates::Lib2<?>::used` in 3.

The first two mappings denote a resolved type annotation, and the last one is an unresolved type annotation. Function identifiers with unresolved type annotations have their dynamic versions resolved to a specific version at dependency resolution time (i.e., at the *Temporal Network Generation* phase). Finally, Präzi splits the annotated call graph into two sections, one immutable section containing resolved function signatures, and another section containing unresolved function signatures. The annotated call graphs are then stored in a dataset. The final dataset should contain all downloaded packages that include creation timestamp, manifest file, and annotated call graph with global identifiers.



(a) Package A depends on B version *1.**.

(b) Full dependency resolution tree with time.

Figure 2.3: Retro-active dependency resolution

2.2.2 Temporal Network Generation

Retro-active Dependency Resolution To study the evolution of the relationships between packages in a repository, we perform retroactive dependency resolution [4] that generates a concrete dependency network valid at a given timestamp t. The use of dynamic versions in package manifests complicates network generation of package repositories. During resolution time of package dependencies, a dynamic version instructs the dependency resolver to fetch the most recent version within its allowed version boundary, making the relationship between packages contemporary. Package A depending on the dynamic version $1.*$ of package B that satisfies any version with a leading *1.* (e.g., *1.0*,*1.8*, or *1.20.2*) in Figure 2.3a exemplifies a dynamic version. Given that Package B releases version 1.1 at t_1 and 1.2 at t_2 ($t_1 < t_2$) in Figure 2.3b. At t, where $t_1 < t < t_2$, a dependency resolver will select version 1.1. However, at $t > t_2$, it will select version 1.2, highlighting the temporal changes in package relationships.

Given a timestamp t, Präzi creates a subset $mirror_t$ of our local mirror (i.e., copy of the Crates.io index) containing packages and versions with a creation timestamp t_c satisfying $t_c \leq t$. Then, for each package version manifest file in $mirror_t$, we resolve its dependencies using a dependency resolver. Dependency resolvers are usually integrated into package managers and are available as independent libraries.

Call Graph Unification The unification is a two-phase process. In the first phase, we build a resolved dependency tree for each package version in $mirror_t$ and then perform a level-order traversal of each tree to merge call graphs of child nodes with their parent nodes. The output is a unified call graph of statically dispatched function calls for each package version in $mirror_t$. In the merge phase of a parent and a child call graph, we complete the unresolved function identifiers in the parent call graph with the resolved version available in the child node. The function `io::crates::Lib2v0.2::used` in [5] replaces the unresolved function `io::crates::Lib2<?>::used` in [3] with $v0.2$.

In the second phase, we need to deal with dynamically dispatched functions and localize call targets across package boundaries. To illustrate this process, we introduce the following scenario: Package `A` depends on package `B` and package `C`. Both `B` and `C` depend on the library `serde`. Furthermore, `B` has a class `Foo` that implements the function `serialize()` in the `Serialize` interface of `serde`. `C` has a function called `bar()` that takes a `Serialize`-like object as an argument and invokes the dynamically-dispatched `serialize()` call on the object.

Before merging the call graphs (i.e. first phase), `bar()` is only aware of call targets that are within `C`. In this example, there are no call targets available (i.e., there is no function implementing `serialize()` in `C`). Thus, in the second phase, we search for other compatible function implementations across packages that are available after merging their call graphs. Here, we would create a call target from `bar()` in `C` to the `serialize()` implementation in `Foo` in `B`. It is possible that `A` may never pass an object of `Foo` from `B` to function `foo()` in `C` in practice. However, the second phase is necessary to ensure that dynamically dispatched functions remain sound after merging all call graphs together.

After constructing a package-level call graph for each package version in $mirror_t$, we merge all partial call graphs into a single CDN. The process consists of aggregating all package-level call graphs and then merging them to remove duplicate nodes and edges. The result is a CDN corresponding to the package repository at the given timestamp t.

2.3 Implementing Präzi for Crates.io

We implement Präzi for Crates.io, the official package repository for Rust. Unlike mainstream package repositories such as Maven Central, PyPI, npm, and NuGet, Crates.io does not host pre-built binaries but the source code of its packages. To generate call graphs for Rust packages, we need to first perform a large-scale compilation of Crates.io and then extract call graphs from generated binaries. Attempting to reproduce the build of a piece of software is known to be challenging [59], Tufano *et. al.,* [60]'s compilation of 219,395 Apache snapshots yielded a success rate of 38%, and Martins *et. al.,* [61]'s compilation of 353,709 Github Java projects yielded a success rate of 56%. An overall low success rate could potentially endanger representative studies of Crates.io.

In the remainder of this section, we describe key implementation choices and results from our large-scale compilation of Crates.io.

2.3.1 Creating a local mirror

We clone two snapshots of the official git-based index of Crates.io[3]: one from the 16th of February 2018 at revision `b76c5ac`, containing 13,991 packages and 79,724 releases, and another from the 14th of February 2020 at revision `6c550c8`, containing 35,896 packages and 208,023 releases. By validating the dependency specifications in the index for incorrect names or dependency constraints, we can avoid building broken releases, thus saving resources.

In the newer revision, we identified 1,506 releases from 201 packages with dependencies that did not match existing packages and 5,667 releases from 4,427 packages with dependencies featuring unsolvable constraints (i.e., no available versions were satisfying

[3] `https://github.com/rust-lang/crates.io-index`

Table 2.1: LLVM call graphs and Rust call mechanisms.

Call Mechanism	Support
Standard function definition [64]	✓
Generic function definition [64]	✗
External function definition (e.g., FFI) [64]	✓
Standard method call (e.g., `Foo::m();`) [65]	✓
Standard method call with receiver (e.g., `Foo.m();`) [66]	✓
Statically dispatched method call (+/- receiver) [67]	✓
Dynamically dispatched method call (+/- receiver) [67]	✗
Macros (e.g., `print!("hello");`) [68]	✗

the constraints). The older revision found that 477 releases had dependencies with unsolvable constraints.

The documentation hosting service for Crates.io, `Docs.rs`,[4] provides Rust users API documentation for every published package release. In addition to automatically generating documentation for package releases, `Docs.rs` also documents the build log and compile status publicly. We create a web scraper that extracts the build status on `Docs.rs` for each release in our dataset. In total, we found that 43,893 indexed releases of the newer revision belonging to 10,154 packages have build failures, amounting to 20% of Crates.io. In addition to the Crates.io index, we use `Docs.rs` as externally validated metadata source in our study.

After subtracting build failures and invalid dependency specifications, our final index of the newer revision amounts to 156,484 releases from 29,480 packages and the old revision amounts to 72,947 releases from 12307. Lastly, we use the official API at `https://crates.io/api/v1/crates` to download all packages and their creation timestamp (not available in the index).

2.3.2 Choosing a Call Graph Generator

There are two approaches for constructing a call graph from a Rust program, the higher-level LLVM analysis,[5] and the lower-level MIR analysis [62]. Rust functions and its calls are either of monomorphized (i.e., static dispatch) or virtualized (i.e., dynamic dispatch) nature. From the documentation[6] and a comprehensive benchmark [63], we can learn that there are two monomorphized features, `macros`[7] and `generic functions`, and two virtualized features, `trait Objects`,[8] and `function pointers`,[9] that dispatch functions in Rust.

As part of the compilation process for Rust programs, we can use the optionally generated LLVM Intermediate Representations (IRs)[10] to construct call graphs. We can achieve

[4] `https://github.com/rust-lang/docs.rs`
[5] `https://llvm.org/docs/Passes.html`
[6] `https://doc.rust-lang.org/book/ch03-03-how-functions-work.html`
[7] `https://doc.rust-lang.org/stable/reference/macros.html#trait-objects`
[8] `https://doc.rust-lang.org/stable/reference/types.html`
[9] `https://doc.rust-lang.org/book/ch19-05-advanced-functions-and-closures.html`
[10] `https://doc.rust-lang.org/rustc/command-line-arguments.html`

Table 2.2: Build statistics for revision `b76c5ac`

Build Round	#Releases	#Packages	Time (hrs)
Crates.io index	72,947	12,307	—
1. Rustc stable	40,366 (55%)	9,376 (76%)	33.8
2. Rustc nightly	+4,972 (+7%)	+976 (+8%)	+13.5
3. Cargo.toml fixes	+2,644 (+4%)	+244 (+2%)	+5.8
4. Native dependencies	+1,862 (+3%)	+235 (+2%)	+16.5
Σ	49,844 (69%)	10,831 (88%)	69.6

this using the LLVM call graph generator[11] or `cargo-call-stack` [69]. Given that the LLVM call graph generator is a convenient tool for generating call graphs in Rust and other languages such as Swift, we assess its potential by considering all possible ways to invoke functions in [70]. We then create examples involving a specific function call or definition and generate a call graph to represent these cases. After examining the generated call graph, we record the support of each feature in Table 2.1. The analysis reveals that the LLVM call graph generator cannot infer non-static dispatch calls or macro invocations. Moreover, it can only infer generic function definitions if their instantiations exist. Due to the absence of Rust-specific type information in LLVM IRs,[12] call graph generators can only resolve monomorphic features and are unable to provide the complete type information needed in Präzi. By analyzing at the Mid-level Intermediate Representation (MIR) level rather than the LLVM level, `rust-callgraphs`[13] offers a more feature-complete call graph by implementing a Class Hierarchy Analysis (CHA) algorithm and is our preferred tool for building CDNs. In addition to monomorphic features, it can resolve function calls dispatched through `Trait Objects`, making it a sounder choice over LLVM-based call graph generators. Although `rust-callgraphs` does not support `function pointers`, this is a negligible trade-off, as the documentation[14] states that `function pointers` are primarily helpful for calling C code from Rust.

For annotating call graphs, the metadata in call graph nodes contains package information and access identifiers. Moreover, the complementary type hierarchy output contains complete type information for creating resolved function identifiers. We also keep the edge metadata that includes dispatch information (i.e., static, dynamic, or macro) in the annotated call graphs.

2.3.3 Large-Scale Compilation of Crates.io

Here, we present build statistics from two revisions, we use the old revision for evaluation in Chapter 2, and the newer revision for Chapter 3.

Table 2.3: Build statistics for revision `6c550c8`

Build	#Releases	#Packages	Time (hrs)
Crates.io index	208,023	35,896	—
`Docs.rs`	156,484 (-24.78%)	29,480 (-17.87%)	—
`rust-callgraphs`	142,301(-9.06%)	23,767 (-19.38%)	10 days

Compilation of revision `b76c5ac`

We perform the build step in several compilation rounds to achieve maximum completeness. We first use a stable version of the compiler, and then iteratively analyze compilation logs to tackle the common failure reasons. The compilation itself ran for three days. Table 2.3 shows the number of successful compilations along with the total time for each compilation round. In the first round, we successfully compile 51% of our set using the `rustc stable 1.22.1 (2017-11-22)` compiler. Unfortunately, a Rust package's build manifest does not specify the compatible compiler versions; a package may use unstable features from a nightly compiler release, which are not backwards compatible with the stable compiler. By swapping the stable compiler version for the nightly version `rustc 1.24.0-nightly (2017-12-06)`, we compile an additional 4,972 package version releases. Analysis of the remaining compilation errors reveals that a large number of builds fail due to missing path-based dependencies. These dependencies are incorrectly pointing the download source of a dependency to a local directory instead of Crates.io. To resolve this, we use Cargo's internal dependency source rewrite feature in and compile 2,644 additional package versions. Finally, we observe that several package releases are using native library dependencies which are not installed on Ubuntu 16.04; with them installed, we compile an extra 1,862 package releases.

Despite our best efforts, we could not compile 23,063 (31%) package releases. To understand why they fail to compile, we analyze the compiler errors and classify them into five categories in Table 2.4. The majority seem to relate to actual programming faults in the packages, in particular the Rust type checker (e.g., `E0277`, `E0599`, `E0425`), syntactical errors and invalid specifications for conditional compilation. A common reason for these error messages is the improper use of Traits. Overall, we can compile 69% of total releases and at least one release for 88% of packages, roughly double the ratio of previous attempts [60].

Compilation of revision `6c550c8`

Some Rust packages depend on external system libraries such as `libavcodec` or `libxml2` to successfully compile. Knowing which external libraries to install for compiling such packages is a manual and tedious process. Luckily, the Rust infrastructure team maintains a Docker image, `rust-lang/crates-build-env`, that bootstraps a Rust build environment pre-installed with community curated systems libraries, increasing the chances

[11] `http://llvm.org/doxygen/CallGraph_8h_source.html`
[12] `https://github.com/rust-lang/rust/issues/59412`
[13] `https://github.com/ktrianta/rust-callgraphs`
[14] `https://rust-lang.github.io/unsafe-code-guidelines/layout/function-pointers.html`

Table 2.4: Build failure reasons for package versions that did not build after installing native dependencies.

Failure reason	#Builds
Compile error w. error code	13,509 (58%)
Compile error wo. error code, of which	7,272 (31%)
... code parsing errors	1,486
... conditional compilation errors	1,058
... dependency resolution errors	719
... type checking errors	278
... other errors	3,711
Custom build script failure	2,127 (9%)
Missing system dependencies	137 (< 1%)
Miscellaneous errors	18 (< 1%)
Σ	23,063

for successful compilations. We use `Rustwide`, an API for spawning Rust build containers, and configure it to use `rustc 1.42.0-nightly` compiler together with `rust-callgraphs`'s compiler plugin. After compilation, we use the analyzer component in `rust-callgraphs` to generate and store the call graphs in our dataset.

We set up a compilation pipeline on four build servers running 34 docker containers to compile packages and build call graphs. It took 10 days to complete it. Table 2.3 shows the compilation results in comparison with index entries and `Docs.rs` results. Overall, our call graph corpus (CG Corpus) has a call graph for 90% of all compilable versions (70% of all indexed versions) and at least one version for 80% of all packages built by `Docs.rs`. The high success rate showcases the practical feasibility of Präzi for Crates.io.

Dependency Resolution For each Cargo.toml manifest in our downloaded dataset, we extract dependencies intended for source code use. These include library dependencies (i.e., `[dependencies]`), platform-specific dependencies (i.e., `[target]`), and also enabled optional dependencies in `[features]`. Both Kikas *et. al.,* [1] and Decan *et. al.,* [27] do not take into account both enabled optional dependencies and platform-specific dependencies, considering only library dependencies when analyzing Crates.io.

The Cargo.toml manifest supports specifications of dependencies using the `semver` schema [10]. A version is a three-part version number: major version, minor version, and patch version. An example of a version is 1.0.0. An increase in the major number denotes incompatible changes, an increase in the minor number denotes backward-compatible changes, and an increase in the patch number denotes small bug fixes. With the support of range operators (i.e., dynamic version) in dependency specifications such as caret (e.g., ∧1.0.0), tilde (i.e., ~ 1.0.0), wildcard (e.g., 1.∗), and ranges (e.g., > 1.0.0. <= 2.0.0), the dependency resolver in Cargo will attempt to resolve the latest version satisfying the constraint. When multiple constraints of the same dependency appear in the dependency tree, Cargo first attempts to find the most recent version satisfying all constraints. For example, for the two constraints, `log` *0.4.** and `log` *0.4.4*, the dependency resolver will resolve `log` *0.4.4*.

However, for example, if the resolver has to resolve `log` *0.4.** and `log` *0.5.**, there is no single compatible version that matches both constraints. Instead, the resolver will include two versions of the same dependency (e.g., `log` *0.4.4* and `log` *0.5.5*) through name mangling to avoid conflicts [47]. The resolution strategy of having multiple versions of the same dependency is similar to NPM.[15]

2.4 Evaluation

This section evaluates a CDN instance of CRATES.IO by comparing its performance against a traditional PDN. The benchmark focuses on structural variations, mainly where edges between packages differ, and performs a reachability analysis for known vulnerabilities. Additionally, we present applications that enable impact analysis of code structures, a task previously unfeasible with PDNs.

We evaluate a CDN instance based on the LLVM call graph generator using a CDN built from revision `b76c5ac`. We apply the same evaluation methodology for CDNs generated with `rust-callgraphs` in Chapter 3 using revision `6c550c8`.

2.4.1 Structural Comparison

The main objective of a CDN, and PRÄZI overall, is to offer a code understanding of dependency relationships between software components than what a traditional PDN provides. A direct comparison is impractical because of the distinct abstraction levels of both networks. However, by "uplifting" the CDN, we can approximate a PDN called $\widehat{\text{PCDN}}$ (a subset of PDN), which serves as a more precise version of the original PDN. Our comparison lies between this newly formed $\widehat{\text{PCDN}}$ and the original PDN. We construct $\widehat{\text{PCDN}}$ by including a package dependency $\langle \texttt{A}, \texttt{B} \rangle$ in its edge set if there is even a single function call from package *A* to package *B* in the CDN. Through this process, we intentionally diminish CDN's precision, thereby defining an absolute lower limit for CDN enhancements.

As Section 2.3.2 outlines, the call graph generation process in PRÄZI could potentially omit function calls across packages in the CDN. This omission could lead to missed dependency relationships in the $\widehat{\text{PCDN}}$. Hence, our evaluation primarily quantifies the disparities between the PDN and $\widehat{\text{PCDN}}$ and qualitatively explores the underlying reasons. In order to maintain a fair comparison, we eliminate from the PDN any packages that we could not compile, as these would not appear in the $\widehat{\text{PCDN}}$. We extract a subset of dependency relationships in the PDN but are absent in the $\widehat{\text{PCDN}}$. We carry out a manual code review for each of these dependencies to ascertain whether its absence in the $\widehat{\text{PCDN}}$ is appropriate or a misstep. The following outlines the protocol for the manual code review:

1. **Evidence of import statements:** Without the presence of import statements (i.e., `extern crate dependency;`) in the source code, the library is never used in code.

2. **Spot uses of imported code entities:** Look for uses of imported code entites within in the spotted module. If there are no function calls but uses of struct and traits, this constitues a *data-dependency*. If there is a macro invocation resembling a function call requires inspection of the defined macro. A macro performs code

[15] `http://npm.github.io/npm-like-im-5/npm3/dependency-resolution.html`

Table 2.5: Manual inspection and classification of 381 different dependency relationships between the PDN and the $\widehat{\text{PCDN}}$.

Categorization	#Samples
i) Dependencies absent in $\widehat{\text{PCDN}}$ (and should be)	**133** (35%)
... unused or no import statements	73
... invoked in non-library portion of a package (e.g., in `bin`)	53
... invoked in test code but not part of the library	7
ii) Dependencies absent in $\widehat{\text{PCDN}}$ (but should not be)	**248** (65%)
ii.1) Call graph generator	114 (46%)
... call inside a generic function	72
... dynamic function call	12
... missing generic definition	8
... C method invocations	22
ii.2) Type-only dependencies	50 (20%)
... imported `Trait` or `Struct`, no function call	50
ii.3) Preprocessor	84 (34%)
... part of functions with conditional compilation	58
... use of macro functionality	26
Σ	**381**

generation at compile time which may or may not be function calls. If there is a normal function call, evaluate whether this uses a conditional feature. Such calls are considered as *conditionally-compiled*.

3. **Localize the scope of identified import statements:** Identify which portion of the source code imported code entities such as structs, traits, or functions are used. Certain entities may exclusively be used in tests or example code which is out of scope.

The $\widehat{\text{PCDN}}$ contains 42,827 nodes and 110,762 edges. The PDN contains the same number of nodes but has 129,535 edges. A set difference on the edges of the two networks shows that 18,042 edges (i.e., 14%) are not present in the $\widehat{\text{PCDN}}$. Qualitative evaluation of all 18,042 different edges is practically infeasible, as it relies on manual work. Instead, we select a statistically representative subset of its edges using Cochran's sample size formula [71]. From a homogeneous set of edges, at a 95% confidence level with a confidence level interval of 5%, we need a sample of $n = 381$ edges to be statistically significant. In Table 2.5, we break down the results into dependencies that i) should be and are absent in the $\widehat{\text{PCDN}}$ and ii) should be present in the $\widehat{\text{PCDN}}$, but are not.

Our $\widehat{\text{PCDN}}$ identified several instances (35%) with our $\widehat{\text{PCDN}}$ where a conventional PDN might have falsely alerted developers. We resolve two shortcomings—i) unused or nonexistent dependency import statements and ii) dependency usage awareness beyond library code—to present notable precision benefits of the CDN at its fine-grained level. However, we observed that the $\widehat{\text{PCDN}}$ initially misses a substantial number of dependen-

cies. While missing 65% of dependency relationships may seem discouraging for Präzi initially, we categorized these errors manually and evaluated their practical trade-offs.

We attribute almost half of the missed dependency relationships (46%) to potential shortcomings of the call graph generator, particularly dynamically dispatched calls (12%). Only some of these limitations necessarily contribute to unsoundness, especially the absence of calls related to generics in Rust. Generics in Rust—parametric polymorphism in type theory—offer a mechanism for creating generic code or templates. If there are no specific instantiations of the generic type defining the function, the compiler does not generate any concrete functions, hence, no calls. Similarly, calls to wrapper packages of C-libraries are transformed into C-function calls, implying that the call targets the C library directly rather than the package. The remaining cases are potential areas for future improvements to Präzi. About one-fifth of the missed cases involve data-type-only dependencies that a call graph cannot capture, such as importing a struct from another package. Call graphs partially expose this information in the argument and return types of functions.

A further third of the missed dependencies stem from the conditional compilation of packages: a function is only included in the compilation if the appropriate feature toggles are on. We can address this issue by rerunning the call graph generation process for each possible feature toggle combination.

Depending on the CDN's end use, these limitations might be manageable trade-offs. For instance, use cases only concerned with instantiated types (e.g., what functions a package reuses from another) might find excluding generics and C-library wrappers acceptable. In such a scenario, the $\widehat{\text{PCDN}}$ approximates package relationships adequately. Conversely, strict security use cases, preferring to minimize false negatives, even at the expense of false positives, would need to consider calls from generic functions (even if unused).

To validate the broad applicability of these findings, the first two authors conducted an inter-rater reliability study, cross-validating 20 randomly selected pairs of dependencies. After independently assessing and comparing the results, both raters concurred that 19 ratings of the main rater were accurate ($p_0 = \frac{19}{20} = 0.95$). The a priori likelihood of random agreement is $p_c = 0.5$. This results in a Cohen's κ of $\frac{p_0 - p_c}{1 - p_c} = \frac{0.95 - 0.5}{1 - 0.5} = 0.9$[72], indicating nearly perfect agreement[73]. This strengthens our trust in the accuracy and generalizability of our manual inspection.

2.4.2 Reachability Analysis

We perform two examples of reachability analysis using the CDN structure, one focusing on security scanning, and a novel one centered on the deprecation study.

Security Scanning

Dependency networks often facilitate examining the spread of security vulnerabilities [1, 6, 30]. Companies like BlackDuck [74], Tidelift [75], and GitHub proactively help projects identify their exposure to publicly disclosed vulnerabilities in their dependencies using reachability analysis.

Our study seeks to evaluate whether a call-based representation of a repository could yield higher precision in the security vulnerability propagation case. We also aim to assess

the potential severity of soundness issues in real-world testing. To this end, we analyze nine security advisories from Rust's security advisory database, RustSec [76], and their effects on other packages using our CDN and PDN.

In Table 2.6, we present a detailed review of the advisories we considered, along with the number of impacted package versions per network, represented as PDN and $\widehat{\text{PCDN}}$ in the table. For each advisory, the table displays the count of packages impacted in each network (PDN and $\widehat{\text{PCDN}}$), the number of manually selected cases, and each network's precision, recall, and accuracy metrics. Three advisories from our initial set of nine were not analyzed due to operating system specificity, build failures, and advisory relevance. Moreover, we excluded two of the six remaining advisories: `cookie`, due to its lack of callers, and `smallvec`, because its vulnerability appears in a generic function.

The total number of packages in the PDN and $\widehat{\text{PCDN}}$ columns sum up to 8,016 and 649, respectively. In other words, these represent the total counts of packages that the PDN and $\widehat{\text{PCDN}}$ flagged as affected by the security advisories. Using the same review protocol defined in Section 2.4.1, we manually curated a ground truth by randomly selecting 482 cases. Please note that the number of cases manually reviewed varies for each advisory, based on the impact scope of the vulnerability, and the balance between diversity of cases and feasibility of manual review.

We established the accuracy of the two networks against this manually curated ground truth. For each security advisory, we identified the direct dependents of the vulnerable package and investigated whether a function call existed from a dependent to any function of the vulnerable package. We then compared the PDN and the CDN (converted to a $\widehat{\text{PCDN}}$, as in Section 2.4.1) against this ground truth, resulting in a confusion matrix that allowed us to derive precision, recall, and accuracy values:

1. *True Positive* (TP): A package correctly flagged as vulnerable when a vulnerable function call exists.

2. *False Positive* (FP): A package incorrectly flagged as vulnerable when no invocation of a vulnerable function occurs.

3. *False Negative* (FN): A package incorrectly flagged as not vulnerable when at least one call to a vulnerable function is present.

4. *True Negatives* (TN): All other cases where a package is correctly identified as not vulnerable.

Finally, we used standard binary classification metrics (precision, recall, and accuracy) to compare the performance of each network. Table 2.6 reveals that the $\widehat{\text{PCDN}}$ flags 83% fewer packages as affected compared to the PDN, which indicates significantly reduced false positives in the $\widehat{\text{PCDN}}$. This increased precision is consistent across all studied advisories, as highlighted by the perfect precision score of 1 for the $\widehat{\text{PCDN}}$. In contrast, the PDN had a varied precision across advisories, showing more false positives.

However, this precision gain comes with a caveat. While the recall of the PDN remained perfect, the $\widehat{\text{PCDN}}$'s recall dipped for a few advisories, indicating a higher false negative rate. Despite this, our analysis shows that a substantial percentage of affected

packages flagged by the PDN turned out to be false positives, demonstrating the precision advantage of our $\widehat{\text{PCDN}}$. If we took the additional step of including the entire transitive closure in our analysis, the advantage of our $\widehat{\text{PCDN}}$ would likely become even more pronounced. However, we avoided undertaking this step due to the substantial manual effort.

Regarding overall accuracy, our $\widehat{\text{PCDN}}$ outperforms the PDN by a factor of three, even with its slightly lower recall. This improvement in accuracy underscores the potential benefits of using CDNs over traditional PDNs. Our analysis reveals that even with suboptimal tools in real-world scenarios, CDNs can offer significant precision advantages, making them valuable in identifying and addressing security vulnerabilities.

Deprecation Impact Analysis

As packages evolve, their public API often changes to introduce new features or improvements, leading to some functions becoming obsolete. Many programming languages have mechanisms to mark such functions as deprecated, either through API documentation (e.g., Python) or as a language feature (e.g., Java). Although annotating functions as deprecated is common practice among developers, the removal of deprecated code poses a greater challenge. Sawant *et. al.*, report that API producers are generally hesitant to remove deprecated features from their APIs and often lack a formal protocol for their removal [77]. This is primarily because developers can't fully anticipate the impact of such cleanups.

By linking dependent functions, our method Präzi allows us to analyze the impact of changes at the package distribution level. Using a Präzi CDN, developers can estimate the impact of removing deprecated functions on their API clients, both directly and transitively. To illustrate this, we calculate the impact of function deprecation within the Crates.io ecosystem.

In Rust, functions marked for deprecation are annotated with a `#[deprecated]` attribute. The Rust compiler will stop the compilation process if a program links to a deprecated function, unless a `#[allow(deprecated)]` attribute is specified. We identified deprecated functions by extracting function signatures that were prefixed with a `#[deprecated]` attribute. We found 721 deprecated function signatures from 190 package versions in 43 unique packages. We considered only deprecated functions and their callers, i.e., functions that directly call the deprecated functions. Of the 190 package versions, only 42 versions had callers, which reduced our search space to 43 deprecated functions. We then manually matched these deprecated functions to our CDN and found 24 deprecated functions. The rest were not included due to reasons identified in Table 2.5. Our reachability analysis revealed that 13 of these 24 deprecated functions were called by other packages, which affected a total of 163 package versions, both directly and indirectly. This equates to $\frac{163}{42{,}827} = 0.38\%$ of the $\widehat{\text{PCDN}}$.

Table 2.7 presents a summary of our results. In the first column, we list the deprecated functions that belong to specific package versions in an encoded format. For example, texttttplatform_{window/display} represents the two functions `platform_window` and `platform_display`. We found that, on average, 48% of the dependents in the $\widehat{\text{PCDN}}$ subgraph of each respective package version call the deprecated functions either directly or indirectly. In other words, nearly half of their callers would break if the deprecated function were removed. This kind of information can provide library maintainers with crucial

insights into whether it is safe to remove a deprecated function, especially when many transitive consumers might be unknowingly using it due to a transitive call chain.

Table 2.6: Results for the security advisory propagation analysis.

Package	Function	#Packages		#Cases	Precision		Recall		Accuracy	
		PDN	$\widehat{\text{PCDN}}$	(Weight)	PDN	$\widehat{\text{PCDN}}$	PDN	$\widehat{\text{PCDN}}$	PDN	$\widehat{\text{PCDN}}$
`base64`	`encode_config_buf`	257	51	128	0.25	1	1	0.68	0.25	0.92
`cookie`	`parse_inner`	0	0	0	-	-	-	-	-	-
`hyper`	`Headers::set`	21	3	2	1.00	1	1	0.5	1	0.5
`smallvec`	`insert_many`	1,581	0	325	-	-	-	-	-	-
`tar`	`unpack_in`	502	31	61	0.62	1	1	0.71	0.62	0.82
`untrusted`	`skip_and_get_input`	5,655	564	291	0.25	1	1	0.43	0.25	0.86
Σ (or weighted average)		8,016	649	482	0.30	1	1	0.53	0.30	0.87

Table 2.7: All called deprecated functions.

Function	Package	$\widehat{\text{PCDN}}$	#Affected by Dep. Fn.s
`OwnedKVList::{new/id/root}`	`slog:::1.7.1`	93	63
`platform_{window/display}`	`winit::0.7.6`	91	50
`platform_{window/display}`	`winit::0.9.0`	31	16
`platform_{window/display}`	`winit::0.8.3`	44	14
`platform_{window/display}`	`winit::0.6.4`	36	12
`get_formats_list, get_name`	`cpal::0.4.6`	16	8
Σ 13	6	311	163

2.5 Threats To Validity

In the case of security analysis, Präzi's effectiveness crucially hinges on its ability to ensure the absence of false negatives – a core measure of soundness. The potential risk of this threat hinges on the soundness of the call graph generator used, an aspect that lies beyond the purview of this paper. Our initial Rust implementation guarantees soundness for statically dispatched method calls, while other calls are deemed "soundy" [50].

Alongside security warnings, meta-warnings attached to a program can encompass advisories on bugs, performance, and deprecation. Various advisory databases cater to these warnings, such as RustSec's security advisories and those pertaining to performance and semantic bugs [78, 79]. The quest for high precision is desirable in many instances, even if it means missing rare cases—an acceptable trade-off for many projects. Livshits *et. al.,* argues that this balance—sacrificing soundness for precision—is not just standard practice in static analysis tools like Fortify [80]. However, it is essential for their practical utility [50].

Furthermore, our evaluation of the discrepancies between the PDN and $\widehat{\text{PCDN}}$ is not comprehensive. It is a snapshot, limited in scope, and does not account for the entire package repository. Also, we do not evaluate identical dependency edges in both networks, which could manifest as false positives. To mitigate these threats, we employ statistical sampling, serving as a representative picture of the variations between the two networks. While not exhaustive, this approach rapidly pinpoints areas for improvement within the implemented approach and assesses the effectiveness of the call graph generator for dependency analysis.

2.6 Conclusions

In this chapter, we introduced Präzi, a strategy combining manifest data with package call graphs to create an enriched dependency network at the function level. Specifically, Präzi was implemented for Crates.io due to its unique challenges, including a need for large-scale repository compilation and the potential inaccuracies introduced by program analysis approximations.

Our preliminary evaluation objective was to evaluate the representational accuracy of call-based dependency networks for package repositories. Our CDN encompassed 90% of all compilable packages, almost entirely covering the Crates.io repository. Furthermore, our manual inspection of 381 package relationships revealed potential areas of improvement when using a call-based dependency network. Notably, a comparative evaluation of reachability analyses, focusing mainly on security scanning of known vulnerabilities, revealed that the precision of a CDN is 3.3 times greater than a PDN-based representation. However, the call-graph generator's accuracy in handling language features is an essential determinant of package relationship approximations. Misinterpreting these features could lead to erroneous addition or removal of dependencies, significantly impacting the dependency network's accuracy. The utility of such a network is also highly dependent on its specific application.

In summary, our research shows that implementing a CDN for a package repository is feasible and beneficial, as demonstrated through our work with Crates.io. It provides enhanced precision for use cases such as security in reachability analyses and can offer new insights into language-based mechanisms like deprecated methods. Future research

could enrich code-based representations of package repositories by considering references of data types between packages, as call graphs capture these only partially.

3

An Empirical Study Into the Structure and Evolution of Rust's Crates.io

This chapter analyzes the structure and evolution of Crates.io, the package repository for the Rust programming language. We evaluate three types of networks - metadata, compile-validated, and call-based - to assess their accuracy and reliability in various dependency tasks. Additionally, we study the evolution and impact of API reuse and dependency configuration bloat. Our investigation centers around three core questions related to Crates.io' network characteristics, its evolution, and the reliability of dependency networks. Our findings shed light on the trade-offs of these networks when conducting large-scale analysis of package repositories. We conclude by emphasizing the importance of static analysis in exploring transitive relations of package repositories.

Understanding package repositories' structure, evolution, and software reuse patterns can generate holistic yet crucial insights into a software supply chain by learning how packages reuse each other at the manifest and source code levels. This chapter delves into these dynamics by empirically conducting a study using three distinct types of package-based networks: metadata, compile-validated metadata, and call-based networks. Our primary objective is to compare the similarities and differences when utilizing these networks for various dependency tasks, such as counting the number of direct and transitive dependencies and performing reachability analysis. Understanding these differences and similarities offers valuable insights and trade-offs for researchers and practitioners aiming to conduct extensive studies of package distributions.

We also perform repository-wide code analysis of Crates.io by conducting impact analyses of critical APIs and studying the evolution and effects of bloating in dependency configurations. To structure our research, we have formulated three core research questions:

RQ 1 What are the network characteristics of Crates.io?

RQ 2 How does Crates.io evolve?

RQ 3 How reliable are dependency networks?

In **RQ1**, we discovered that a function in Crates.io typically has one static, one macro call, and a vtable with nine function targets. The Crates.io network exhibits a scale-free property, with some functions acting as connectivity hubs. In **RQ2**, we detected a trend of increasing transitive package dependencies without a corresponding growth in declared dependencies. Notably, despite a surge in direct and indirect API calls, approximately 60% of resolved transitive dependencies remain uncalled. Further, 28% of packages contain a function found in their dependencies, contributing to bloat between 1 and 10%. Most packages showed negligible to no reachability, while a minor fraction displayed high reachability. For **RQ3**, our study indicates that call-based dependency networks offer greater precision than their metadata-based counterparts, though data-only dependencies may compromise their accuracy.

3.1 Selecting a time window for dependency resolution

Instead of using a single fixed version at all times, version constraints allow developers to use a time-constrained version that updates itself at new compilations. Nearly all dependencies in Crates.io specify a dynamic version constraint—only 2.92% of all dependency specifications in Crates.io use a single (immutable) version [81]. Before studying the evolution and structure of Crates.io, we first decide the number of time points and a time window between each time point. Although popular studies such as Kikas *et. al.,* [1] and Decan *et. al.,* [27] use a time window of one year to study structural changes, we, instead, determine a time window based on the frequency of structural changes in Crates.io.

After resolving the dependency tree of a set of packages in Crates.io at a time t, we then re-resolve it using six different time points (i.e., one day, one week, one month, three months, six months, and one year) to find a time window where a large fraction of them have a changed dependency tree. We perform this using a set of packages having at least one non-optional dependency at the beginning of 2017 (5,252 package releases), 2018 (9,716 package releases), and 2019 (16,098 package releases).

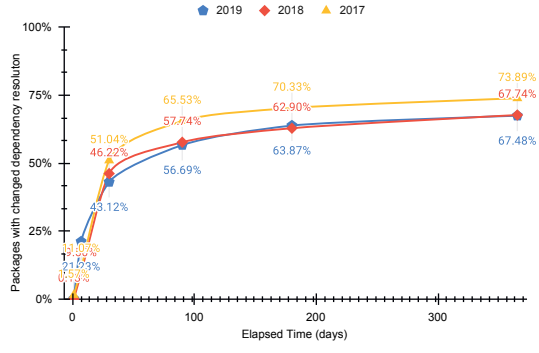


Figure 3.1: Retroactive resolution of dependencies over a time period of one year in 2017, 2018, and 2019

Figure 3.1 shows the fraction of packages with a changed dependency tree (i.e., a tree with at least one different version) over time. We observe a logarithmic trendline for each year group; a high increase of packages with changed dependency between time points before three months, and then it levels out. After one month, we already find that 40% of all packages have a changed dependency tree due to new releases of 148 packages in 2017, 190 packages in 2018, and 240 packages in 2019. In all year groups, we find that the dependence on `libc` triggers a new version resolution for most packages, followed by other popular packages such as `quote`, `serde`, and `syn`. A manual inspection of the release log for `libc`[1] and `serde`[2], suggests a frequency of at least two releases per month.

Finally, we also observe that 26% of all packages in 2017 have an identical dependency tree after one year. Among those unchanged packages, nearly all of them (2017: 83%, 2018: 93%, 2019: 90%) are outdated packages. With outdated, we mean that no recent releases for those packages in more than one year. Although packages may be outdated, they still could use flexible version constraints. In roughly one-third (2017: 31%, 2018: 34% 2019: 40%) of all dependency constraints, the dependencies are outdated packages (i.e., there are no recent releases). In the remaining cases (i.e., where more recent versions exist), the version constraints cover old releases (e.g., depending on `serde` 2.x when 4.x exists), and less than 1% are fixed versions. For example, `xml-attributes-derive::0.1.0`[3] depends on older versions of `syn`, `quote`, and `proc-macro2`, and `trie-root::0.11.0`[4] depends on an old version of `hash-db`.

Given these observations, we select a time window of one month and thus perform dependency resolution every month per year.

3.2 Research Questions

RQ1: What are the network characteristics of Crates.io?

We characterize the calling relationship between packages in Crates.io, and then identify various influential packages featuring a high number of callers and callees within the networks. Specifically, we describe our data corpus and the degree distribution to gain an overall understanding of the direct relationship between functions for a large package repository such as Crates.io.

RQ2: How does Crates.io evolve?

The frequent number of new package releases and the adoption of `semver` range operators in dependency specifications make the relationship between packages highly temporal in Crates.io. We capture these dynamics using both a package-level perspective and the more fine-grained, function-level perspective. In comparison to previous studies [1, 27], we use three different sources, namely metadata, compile-validated metadata, and control-flow data, to understand their differences and similarities for package-based dependency analysis.

[1] `https://crates.io/crates/libc/versions`
[2] `https://crates.io/crates/serde/versions`
[3] `https://docs.rs/crate/xml-attributes-derive/0.1.0/source/Cargo.toml`
[4] `https://docs.rs/crate/trie-root/0.11.0/source/Cargo.toml`

As all our snapshots deviate from a normal distribution according to Shapiro-Wilk ($p < 0.01 \leq \alpha$), we use the non-parametric Spearman correlation (ρ) coefficient for correlation analysis. Using Hopkins's guidelines [82], we interpret $0 \leq |\rho| < 0.3$ as no, $0.3 \leq |\rho| < 0.5$ as a weak, $0.5 \leq |\rho| < 0.7$ as a moderate, and $0.7 \leq |\rho| \leq 1$ as strong correlation We answer the following **sub-RQs** using a package-level and call-level perspective:

RQ2.1: How do package dependencies and dependents evolve?
RQ2.2: How does the use of external APIs in packages evolve?
RQ2.3: How prevalent is function bloat in package dependencies?
RQ2.4: How fragile is Crates.io to function-level changes?

For deciding on reasonable time points for evolution studies of package repositories, we include a guideline with analysis in Section 3.1.

RQ3: How reliable are dependency networks?

A dependency network approximates how packages use each other in a repository. Both metadata-based networks and call-based networks have trade-offs and limitations that affect how reliable they estimate actual package relationships. To understand how accurate these networks are in practice, we perform a manual analysis of 34 random cases where a metadata-based and call-based dependency network infers relationships differently. The cases involve both direct and transitive package relationships.

3.3 RQ1: Descriptive Analysis

3.3.1 Summary of Datasets

Before investigating the calling relationship among packages in Crates.io, we first describe our datasets of generated call graphs (i.e., **CG Corpus**) and our largest CDN dated February 2020 in Table 3.1. After removing all function calls to the standard libraries of Rust, the call graph corpus has over 121 million functions and 327 million function calls from 142,301 compiled releases. When merging call graphs into a CDN, we generate a compact representation with over 44 million functions and 216 million function calls, a sizeable reduction of 2.5 and 1.5 times of the **CG Corpus** (i.e., functions and calls), respectively.

Table 3.1 also breaks down function calls into their dispatch type, namely macro, static, and dynamic calls. Notably, nearly 80% of all edges in the **CG Corpus** are of a dynamic dispatch type, followed by static dispatch (18%) and macro invocations (2%). The high number of dynamically dispatched calls in the network indicates that Crates.io has a large pool of possible target implementations to virtual functions—not necessarily magnitude more function calls than statically dispatched calls. When comparing the access modifiers between functions, we can see that 40% of all functions inside Crates.io are publicly consumable. Also, we can see that calling functions in external packages is widespread in Crates.io; half of all the function calls invoke a function from an external package (i.e., inter-package call). Unlike the other two dispatch forms, 91% of all macro dispatched calls exclusively target macros defined in external packages. Overall, the high number of

Table 3.1: Summary of Datasets

	CG Corpus	CDN Feb'20
Functions	**121,825,729**	**44,190,643**
... public access	46,236,696	20,157,155
... private access	75,589,033	24,033,488
Call edges	**327,535,934**	**216,239,360**
Intra-Package Calls	**169,579,315**	**102,136,956**
... macro invocation	693,148	356,329
... static dispatch	28,570,266	20,650,000
... dynamic dispatch	140,315,901	83,130,627
Inter-Package Calls	**157,956,619**	**114,102,404**
... macro invocation	7,183,797	2,178,547
... static dispatch	29,650,173	13,319,367
... dynamic dispatch	121,122, 649	98,604,490

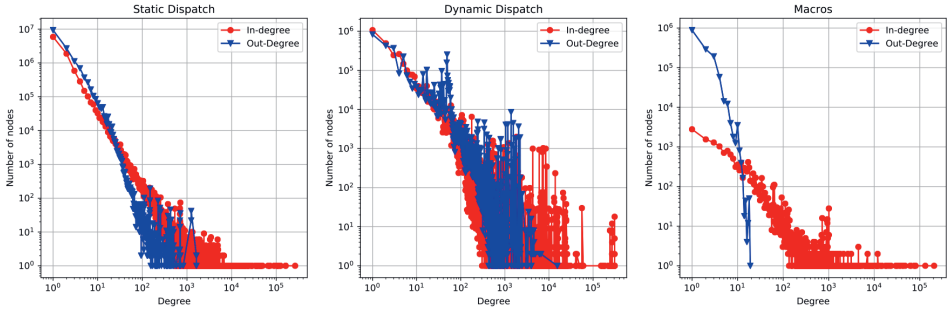


Figure 3.2: Degree distribution of all function calls

declared public functions and the large degree of inter-package calls indicate that code reuse in the form of functions between packages is a prevalent practice in Crates.io.

Function reuse is prevalent; 40% of functions are public and 49% of call edges target a dependency.

Function Call Distribution

Figure 3.2 presents the degree distribution for all function calls grouped by their dispatch type, and Figure 3.3 is a narrowed-down version looking at only inter-package function calls. The out-degree of a function is the number of function calls to other unique functions (i.e., number of caller-callee relationships). The in-degree of a function is the number of callers to a function across Crates.io (i.e., number of callee-caller relationships). Given a function `a()` in a package, the out-degree looks at what calls `a()` makes. The in-degree

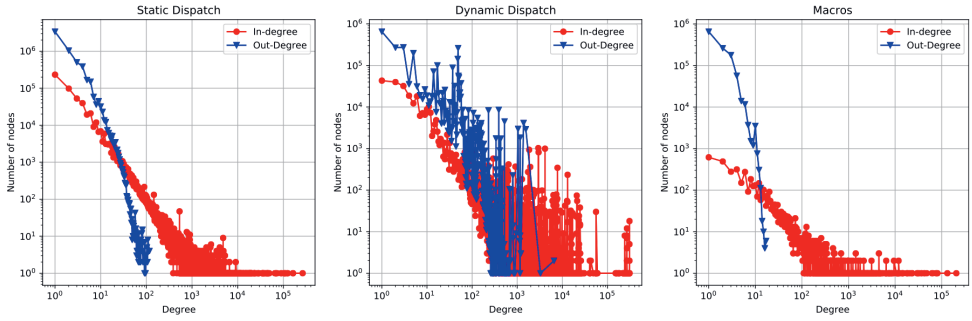


Figure 3.3: Degree distribution of inter-package function calls

Table 3.2: The top 5 functions with most statically-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
epoxy	`load_with`	1,625	serde	`missing_field`	264,281
sv-parser-syntaxtree	`next, into_iter`	1,243	log	`max_level`	162,747
python-syntax	`__reduce`	821	vcell	`set`	125,287
rustpython-parser	`__reduce`	720	serde_json	`from_str`	73,171
mallumo-gls	`load_with`	712	futures	`and_then`	65,043

looks at which functions in Crates.io call `a()`. As mentioned earlier, inter-package calls are only function calls between packages (i.e., pruning all internal calls). The out-degree distribution for dynamic dispatch represents the number of possible target functions in a virtual method table,[5] and, for static- and macro dispatch, the number of function calls. The in-degree distribution presents the aggregated number of callers for a function (i.e., callee) and implementations of virtual functions for dynamic dispatch, respectively. Overall, we can observe a long tail for both the in-degree and the out-degree of each dispatch mechanism, suggesting that the Crates.io CDN is a scale-free network with the presence of a few nodes that are highly connected to other nodes in the network (i.e., hubs). Finally, Tables 3.2 to 3.4 describe the top 5 functions with the highest in-degree and out-degree calls per dispatch type. The top 5 list is an aggregation of functions per package. For example, the `serde` package in Table 3.3 has over 300 serialization functions with an in-degree similar to 264,281. Thus, we present the top 5 functions as the top most called function(s) per package. In the following, we describe key results for each of the three dispatch forms.

Static dispatch The median out-degree for statically dispatched function call is 1 call (mean: 2.25) in both cases and at the 99th percentile being 15 calls (13 calls for inter-package calls). When comparing the out-degree between statically dispatched calls in Figure 3.2 and Figure 3.3, we can notice that there are 1865 functions (0.012%) that call more than 100 other internal functions in Figure 3.2. The highest number of calls made

[5]https://alschwalm.com/blog/static/2017/03/07/exploring-dynamic-dispatch-in-rust/

Table 3.3: The top 5 functions with most dynamically-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
hyperbuild	`match_trie`	15,460	serde	`deserialize_any`	307,976
heim-common	`to_base, from_base`	6,597	serde_json	`from_str`	268,887
uom	`to_base, from_base`	6,045	serde_urlencoded	`deserialize_identifier`	59,737
fpa	`I1F7, I2F6`	3,966	yup-oauth2	`token`	42,737
rtdlib	`deserialize`	2,470	cpp_core	`cast_into`	28,278

by a single function in both plots is to 1625 local functions and 116 external functions, respectively. The relatively high number of internal function calls among the outliers seems un-realistic at a first glance. Upon manual inspection of the source code of the only two packages having functions with an out-degree greater than 1000 (see Table 3.2), namely `epoxy`[6] and `sv-parser-syntaxtree`[7], we identify that this is the result of generic instantiations for creating bindings to the `libepoxy` (an OpenGL function pointer manager) and tokens for parsing SystemVerilog files.

The median in-degree for statically dispatched function calls are 1 (mean: 3.6) and the 99th quantile is 24. When omitting all internal calls and considering only inter-package calls, the median is 2 (mean: 24) and the 99th quantile is 208. There are three functions having over 100,000 external calls in Table 3.2, `serde` for *serialization*, `log` for *logging*, and `vcell` for *memory management*. While the first two are the most downloaded and depended upon packages in Crates.io, `vcell` stands out for only having nearly 300 dependent packages. After inspection of the source code of those packages for the specific `set` call, we could identify extensive implementations of low-level drivers to interface various microcontrollers such as the `Cortex-M` and `STM32` series.

Dynamic dispatch We use `vtable` to refer to all implementations of a virtual function of a Trait object. In practice, each Trait object points to compatible Trait Implementations (having a `vtable` with function and other member implementations). The median number of function targets function `vtable` is 9 (mean: 42 (all), 32 (inter-package)) for both all function targets and only inter-package function targets. The main deviation is at the 99th percentile, the outdegree for all function targets is 800 for all targets, two times higher than when only considering inter-package function targets. The highest out-degree function in Table 3.3 is `match_trie` in the package `hyperbuild v0.0.10`, a HTML minification library, having a `vtable` with 15,460 function targets. The function takes as an argument a `trie: &dyn ITrieNode<V>` Trait, invoking `get_child` and `get_value` of the Trait `ITrieNode`. The Trait is implemented for all forms of HTML entities, explaining this high outdegree value. In total, there are 38,352 (< 0.94%) functions that populate a vtable with more than 1000 function targets. Similarly, we can observe 11,906 (< 0.36%) inter-package function calls with over 1000 function targets.

The median in-degree for implementing a virtual (i.e. trait) function is 3 (mean: 53) and the 99th percentile is 608. When only considering inter-package relationships, the median

[6] `https://docs.rs/crate/epoxy/0.1.0/source/`
[7] `https://docs.rs/crate/sv-parser-syntaxtree/0.6.0/source/`

Table 3.4: The top 5 functions with most macro-dispatched calls

Outdegree			Indegree		
Package	Function	#	Package	Function	#
item	`path_segment`	19	log	`log!`	205,810
fungi-lang	`fgi_module`	18	bitflags	`__impl_bitflags!`	77,848
syn	`path_segment`	17	lazy_static	`__lazy_static_internal!`	64,161
device_tree_source	`parse_data`	17	trackable	`track!`	47,648
npy	`map`	17	serde	`forward_to_deserialize_any_method`	43,063

is 3 (mean: 64) and the 99th percentile is 875. As shown in Table 3.3, the most commonly implemented trait function stems from serializer packages such as `deserialize_any` in `serde`, `from_str` in `serde_json` and `deserialize_identifier` in `toml`. In addition to serialization functions, we can also observe that 42,737 functions implement the trait function `token` in `yup-oauth2` for user authentication with OAuth 2.0.

Macro dispatch When comparing the out-degree for both all and inter-package calls, we can observe a similar trend between them: the median is 1 (mean: 1.7) and the 99th quantile is 6, suggesting that macro-dispatched calls are largely inter-package calls. This resonates with our observations for macro-dispatched calls in Table 3.1. Looking at functions calling the most number of macros in Table 3.4, we can observe that the outdegree generally is relatively low in comparison to the other two dispatch types. The function `path_segement` in `item` makes in total 19 macro calls, the highest in Crates.io. The median in-degree is 7 (mean: 146) and the 99th quantile is 1427. When only considering inter-package calls, the median is 12 (mean: 391) and the 99th quantile is 6433. We can observe comparable numbers to the in-degree with the other two dispatch types in Table 3.4. With over 200,000 functions in Crates.io calling `log!`, it is the most called macro followed by `__impl_bitflags!` and `__lazy_static_internal!`. Generally, we can observe that the top most called macros belong to popular packages in Crates.io that are known to simplify logging (`log`), generate bit flag structures (`bitflags`), and wrapping error messages (`quick-error`).

The median function in Crates.io makes one static call, one macro call and has a `vtable` with nine function targets. The median function is also dependent upon by one static call, one macro call, and implemented by three functions.

Crates.io is a scale-free network, indicating the presence of a handful of functions or hubs that are highly connected to other functions in the repository

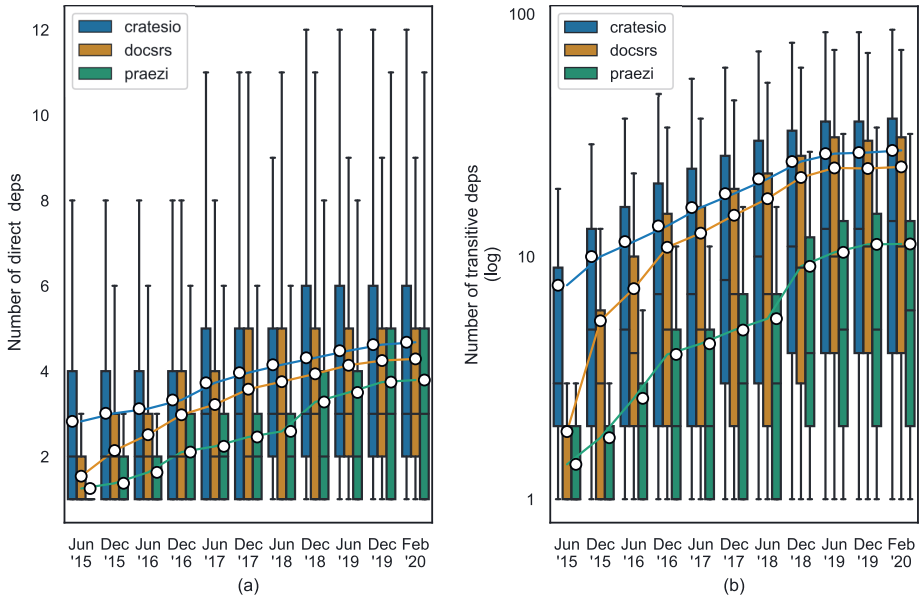


Figure 3.4: The evolution of package dependencies on two metadata-based networks, Crates.io and Docs.rs, and one call-based network, Präzi.

3.4 RQ2: Evolution

3.4.1 RQ2.1: How do package dependencies and dependents evolve?

Figures 3.4 and 3.5 present the number of direct and transitive package relationships split by network type over time. Each sub-plot also features line plots showing the mean with a circle for each snapshot. By using three different network representations, we can understand and contrast the differences between the three approximations of dependency relationships.

Direct dependencies Direct dependencies refer to the dependencies that a developer specifies in a package manifest. For each network group in Figure 3.4a, we see a marginal growth in the median number of direct dependencies over time. The median number of dependencies for a package grew from two to three between 2015-2020 for the Crates.io index network as an example. The median is also similar in the other two networks. Although there are notable differences in the overall spread in the formative years of Crates.io, the growth curve is relatively comparable between the networks. The correlation between the number of direct dependencies between the three networks (normalized) yields a significantly strong $\rho = 0.89$ between 2017 and 2020 (2015-2017: $\rho = 0.71$), indicating that the networks approximate each other.

When comparing the mean between the CDN and the Crates.io index network, we find the average package call at least one function in 78.8%[8] of its direct dependencies. As

[8]after normalizing the networks (i.e., inner join of common packages in all three networks)

the Crates.io index network has a higher overall spread than the `Docs.rs` network, and the `Docs.rs` network has a higher overall spread than the CDN, we can derive that the Crates.io network represents an upper-bound and the CDN a lower-bound on the number of direct dependencies. With 75% of all packages having less than six direct dependencies, the results are overall similar to the findings of Decan *et. al.,* [27] and Kikas *et. al.,* [1].

Package maintainers use 2 to 3 direct dependencies and are unlikely to increase their use over time. The three networks have comparable results.

Transitive dependencies Transitive dependencies represent the indirect dependencies of a package after resolving its specified dependencies. In comparison to the direct dependencies, in Figure 3.4b, we can observe an initial superlinear growth, followed by a period of stabilization (since 2018) for the three networks. The median number of transitive dependencies in 2015 is 5 for the Crates.io index network and 1 for the other two networks. The median number of transitive dependencies grew with a delta of 5 additional packages for the CDN, 9 for the `Docs.rs` network, and 12 for the Crates.io index network in five years. While we can find a similar continuing growth trend to Figure 3.4a, we observe higher degrees of dispersions between the CDN and the other two networks. The third-quartile in nearly all CDN snapshots is the same or below the median of the other two networks. Thus, half the packages in the Crates.io index network and `Docs.rs` network report a higher number of transitive dependencies than 75% of packages in the CDN. When normalizing the networks and comparing the mean between the CDN and the Crates.io index network in 2020, we find the average package call at-last one function in 40%[9] of all resolved transitive dependencies. The discrepancy indicates substantial differences between call-based and metadata-based networks in network analyses; CDNs will overall report a notably lower number of transitive dependencies than the metadata-based ones.

Finally, the correlation between the number of transitive dependencies between the three networks (normalized) is generally strong, with an average $\rho = 0.84$ between 2017 and 2020 (2015-2017: $\rho = 0.70$). In other words, the more resolved transitive dependencies a package has, the more transitive dependencies it will call (and vice versa). However, we identify a moderate average correlation $\rho = 0.62$ between the number of direct dependencies (i.e., either metadata-based or call-based) and the number of call-based transitive dependencies in 2017-2020. In 2015-2017, we observe a general weaker correlation, with $\rho = 0.47$. Thus, two packages with the same number of direct dependencies are likely to have different number of transitive dependencies.

The average dependency tree of resolved packages has nearly grown thrice (5 to 17 transitive dependencies) in 5 years. Substantial differences exist between the networks; packages are not calling 60% of their resolved transitive dependencies.

Direct dependents In addition to dependencies, dependents measure the number of consumers a package has. In the context of this study, we consider a consumer as an

[9]See footnote 22

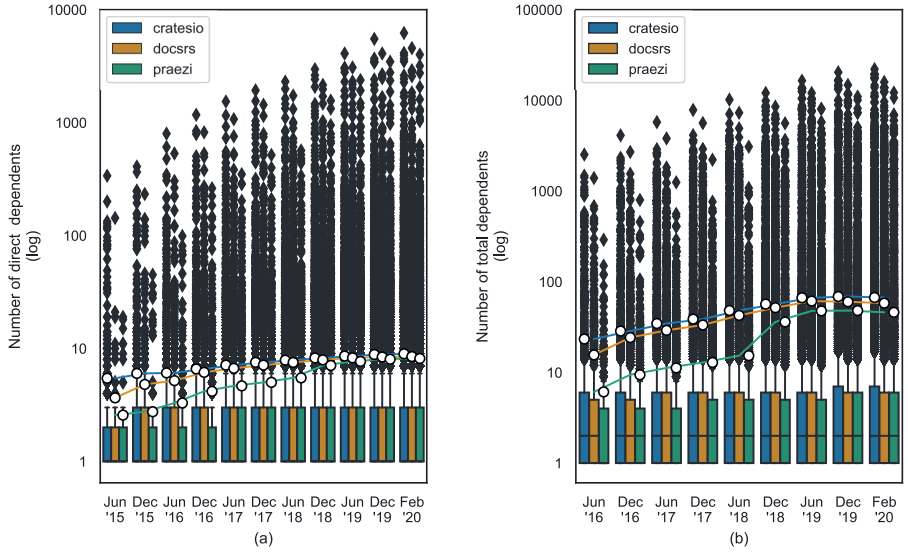


Figure 3.5: The evolution of package dependents on two metadata-based networks, Crates.io and `Docs.rs`, and one call-based network, Präzi.

internal consumer (i.e., a package making use of another package within Crates.io). Figure 3.5a presents the number of dependents over time. Irrespective of the network, we can see that the median number of consumers per package remains unchanged at one over time. Similarly, we can also find the interquartile ranges of the networks to be identical from June 2017 and onwards. In that period, the top 25% packages have at least three or more consumers. The correlation between the number of direct dependents for the three networks (normalized) yields a strong $\rho = 0.81$ between 2017 and 2020 (2015-2017: $\rho = 0.75$), indicating (similar to direct dependencies) that the networks closely approximate each other.

When comparing the mean over time, we see a steady growth of the number of direct dependents for all three networks. The growth pattern is a result of a few commonly used packages (e.g., `serde` and `log`) having the largest share of consumers in Crates.io (see also Figure 3.3). The outliers in the boxplot represents the top-most used packages for each network. Here, we can observe notable differences in the range and number of outliers between the networks. The number of top dependent packages in June 2018 is 651 for the CDN, 1245 for the `Docs.rs` network, and 1680 for the Crates.io index network. There are 2.5x more top-dependent packages for Crates.io than in the CDN. When comparing the top-most dependent packages in each network, the most consumed package has 566 dependents in CDN, 1735 in the `Docs.rs` network, and 2305 in the Crates.io index network. Although the gap between the outliers in the networks reduces over time (i.e., from 2.5x to 1.8x in 2020), there are notable differences between the networks when analyzing the top-most dependent packages in Crates.io.

Overall, the results are similar to the findings of both Decan *et. al.,* [27] and Kikas *et.*

al., [1], suggesting that an average Crates.io package has a relatively constant and low degree of consumers in general. While the networks seem comparable and interchangeable at large, there is a notable discrepancy between the outliers (i.e., topmost used packages in Crates.io) in metadata-based networks and call-based networks in earlier snapshots, potentially yielding differences in network analyses of top dependent packages.

The average number of consumers of a package remains at one over time. Similar to direct dependencies, the networks approximate each other (except for top-dependent packages).

Total dependents Figure 3.5b shows the total number of dependents per package. The total number of dependents include both direct and transitive dependents. We omit both June and December 2015 as these snapshots only have 19 and 47 transitive dependents in the CDN, respectively. Except for June 2016, the median number of total dependents remains constant at two for the three networks. Thus, in addition to the one median direct consumer in Figure 3.5a, packages also have one median transitive consumer. When looking at the top 25% consumed packages, the number of total dependents ranges from 8 or more consumers for the Crates.io index network and 7 or more consumers for the remaining networks. There is also a slight increase in the overall range at two occurrences for the CDN (Feb'17, Dec'19) and one occurrence for the `Docs.rs` network (Dec'17) and the Crates.io index network (Dec'19). When comparing the mean and outliers between the networks, we find a similar growth pattern and gap to Figure 3.5a.

Similar to transitive dependencies, we also find a general strong correlation between the number of transitive dependents between the three networks (normalized) ($\rho = 0.77$), and also a moderate correlation between the number of direct dependents and transitive dependents ($\rho = 0.54$).

Overall, we see that the total number of dependents remains stable over time with a few cases of gradual increase. Moreover, we see that the distributions of dependents are generally much lower in comparison to the transitive dependency relationships in Figure 3.4b. Thus, the results indicate that an average package in Crates.io has a handful stable number of consumers.

The average package also has one transitive consumer that remains unchanged over time. Similar to direct top-most dependent packages; indirect consumers are using them to a much higher degree than previously.

3.4.2 RQ2.2: How does the use of external APIs in packages evolve?

Figure 3.6 describes the evolution of the number of direct and transitive inter-package (i.e., API) calls per package for dependencies on the left-hand side and dependents on the right-hand side. When looking at the number of calls to dependencies over time, we make two major observations. First, the number of direct and transitive calls to dependencies has an initial superlinear growth, followed by a period where the growth slows down from December 2018 and onwards. From December 2016 to December 2019, the number of direct

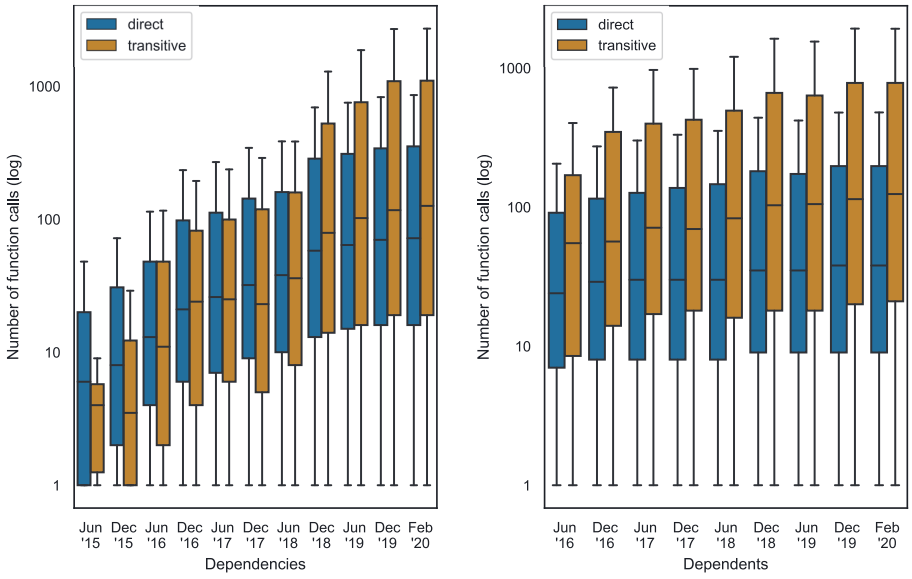


Figure 3.6: The evolution of the number of functions calls to dependencies and dependents

calls grew from 21 (transitive: 24) to 70 (transitive: 116), a three-fold increase in three years. On average, we also see a growth of 6.6 new function calls to direct dependencies and 12.2 new indirect calls to transitive dependencies every six months. Second, we can see that the median number of transitive calls overtakes the median number of direct calls in December 2018. Our findings unveil that the amount of calls to indirect APIs are comparable in numbers to calls of direct APIs. Recent snapshots further indicate that packages invoke more indirect APIs than direct APIs. The transitive median API calls in December 2019 is 1.6x larger than the median direct API calls.

The average API usage of transitive dependencies is both greater and comparative to direct dependencies in recent years.

Similar to the total dependents in Figure 3.4d, we also omit the two snapshots in 2015 due to an insignificant number of transitive dependents. Generally, we can observe a continuous growth of the number of direct and transitive consumers of package APIs over time. The median number of consumer grew from 25 callers in 2015 to 38 callers in 2020, an average growth of 1.6 new functions per year. The median of indirect consumers is larger than that of direct consumers, growing from 55 callers in 2015 to 124 callers in 2020, an average of 8.75 new functions every six months. When comparing the growth pattern between direct and transitive dependents, the gap between the median of direct dependents and transitive dependents expands over time. Moreover, we also find that the interquartile ranges and overall range is greater for transitive dependents than for direct dependents in all snapshots. A package with transitive dependents is likely to have more

indirect callers than direct callers of their APIs. Notably, the median number of transitive dependent callers (median: 114) is three times larger than the median of direct dependent callers (median: 38) in 2020. When also taking into account the findings of transitive dependency callers, our results strongly indicate that indirect users of library APIs is both highly prevalent in Crates.io, and comparable to direct users of library APIs. Despite the largely unchanged number of direct and total dependents (See Figure 3.5) over time, we see indications that developers are increasingly using more APIs over time.

Packages with transitive consumers have three times more API callers stemming from indirect consumers than direct consumers.

Below, we summarize the two perspectives of package relationships using both the metadata-based results with function-based results:

Dependencies: Packages depend on an increasing number of transitive dependencies over time. Package maintainers, however, are not declaring more dependencies. Although there is an increase of new direct and indirect API calls to dependencies over time, roughly 60% of all resolved transitive dependencies are not called.

Dependents: The number of total dependents, one direct and one transitive consumer, remains constant over time. However, consumers have a growing number of callers over time. For packages with transitive consumers, there is a higher number of calls stemming from indirect callers than direct callers.

3.4.3 RQ2.3: How prevalent is function bloat in package dependencies?

Packages depending on a growing number of external packages are also likely to introduce dependency conflicts. Conflicts arise when a dependency resolver is unable to eliminate the co-existence of a package in a dependency tree due to version incompatibility. For example, a resolver may arrive that there is no overlapping version when two packages in a dependency tree depend on package `A` where the former specifies a version constraint 1.* and the latter 2.*. Rust's Cargo package manager avoids such conflicts by allowing multiple versions of the same package to co-exist in a dependency tree using *name mangling* techniques [47]. A potential drawback of this strategy is the risk of bloating binaries due to multiple copies of identical yet obfuscated functions.

As a proxy for function bloat in binaries, we calculate the percentage of co-existing functions for all public functions in Crates.io. We denote a co-existing function as multiple copies of identical function identifiers loaded from different versions of the same package. It is important to note that the measure is an estimation and does not guarantee the semantic equivalence of functions. Before measuring the percentage of co-existing functions, we first inspect the presence of co-existing functions in all Crates.io packages. On average, we find packages having at least one co-existing function to be 5.4% of

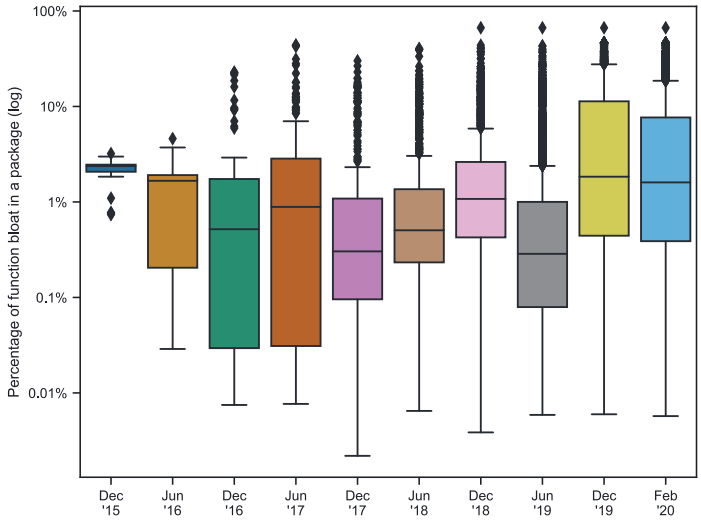


Figure 3.7: Percentage of co-existing functions (i.e., bloat) in Crates.io packages

Crates.io in Dec 2015-Dec 2017 and 28% of Crates.io in Jun 2018-Feb 2020. There are no packages with co-existing functions in June 2015. Largely non-existent in the formative years of Crates.io, we find that function co-existing among dependencies is relatively prevalent in recent years.

Among packages having co-existing functions, Figure 3.7 breaks down the percentage of co-existing functions in dependencies of packages over time. We can observe that the median fluctuates between 0.3% and 1.6% over time, indicating a constant yet insignificant amount of function co-existence in packages. 75% of all packages range between 1 to 10% co-existing functions in their dependencies, suggesting that a majority of packages have a small amount of possible bloat in their binaries. Thus, bloating of binaries from co-existing dependency functions are highly unlikely for packages with at least one co-existing function in Crates.io.

Finally, we find a small minority (i.e., outliers) of packages with a high degree of possible function bloat between December 2018 and February 2020. The package reporting the highest bloat of this time frame is `downward` with 67% bloat. However, it is an invalid outlier as it has a circular dependence on itself.[10] Thereby, the two packages with highest bloat is `const-c-str-impl` and `mpris` with 43% and 46% bloat, respectively. Upon manual inspection of their respective dependency tree, we identify that the packages have a dependence on multiple versions of `proc_macro`, `quote`, `syn`, and `unicode_xid`, common libraries for creating procedural macros. For example, `mpris` indirectly uses four different versions of `syn` and `quote`.[11]. We also make similar observations in three other outliers: `js-object` (33%), `js-intern-proc-macro` (41%), and `mockers_derive` (43%). Further investigation could perhaps reveal whether the combination of certain procedural macros

[10] `https://crates.io/crates/downwards`
[11] `https://docs.rs/crate/mpris/2.0.0-rc2/source/Cargo.lock`

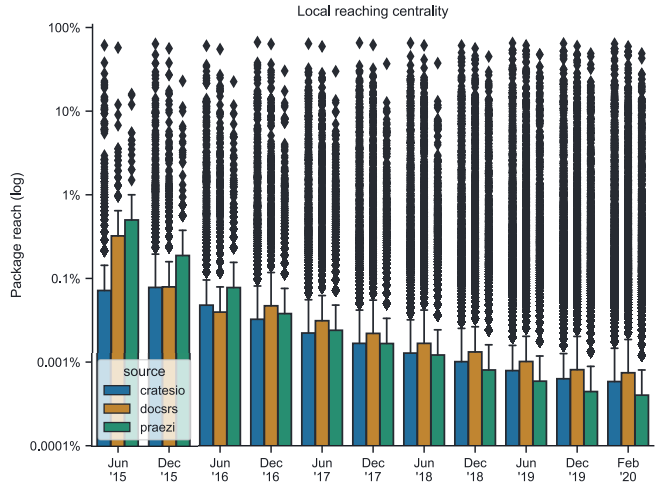


Figure 3.8: Distribution of Package Reachability

Table 3.5: Most central APIs in the largest component in Dec 2015, Dec 2017, and Feb 2020

Dec 2015 (size: 534)			Dec 2017 (size: 6,004)			Feb 2020 (size: 24,857)		
Package	Function	Reach	Package	Function	Reach	Package	Function	Reach
pkg-config::0.3.6	`find_library`	10%	log::0.3.8	`__log`	16%	log::0.4.10	`max_level`	30%
gcc::0.3.20	`Build::new`	6%	libc::0.2.34	`memchr`	12%	serde::1.0.104	`next_element`	24%
libc::0.2.1	`memchr`	6%	lazy_static::0.2.11	`get`	11%	bitflags::1.2.1	`__fn_bitflags`	23%
log::0.3.4	`__static_max_level`	6%	bitflags::1.0.1	`__fn_bitflags`	8%	lazy_static::1.4.0	`get`	21%
bitflags::0.3.3	`bitflags`	3%	unicode-width::0.1.4	`width`	8%	libc::0.2.66	`sysconf`	18%
gcc::0.3.20	`Build::compile`	3%	serde::1.0.24	`deserialize`	6.3%	libc::0.2.66	`isatty`	18%
log::0.3.4	`log::macros log`	6%	lazy_static::0.2.11	`lazy_static`	5.95%	memchr::2.3.2	`memchr`	18%
gcc::0.3.20	`compile_library`	4%	byteorder::1.2.1	`write_u32`	5.1%	itoa::0.4.5	`Buffer::new`	17.5%
time::0.1.34	`precise_time_ns`	2.5%	libc::0.2.34	`localtime_r`	5%	ryu::1.0.2	`Buffer::format_finite`	17%
libc::0.2.1	`sysconf`	2.5%	time::0.1.38	`num_seconds`	4.1%	serde_json::1.0.48	`from_str`	10%

libraries are highly likely to always result in bloated dependency tree configurations.

28% of all packages in Crates.io have a co-existing function in their dependencies. Among those packages, between 1-10% of imported functions from dependencies are bloated.

3.4.4 RQ2.4: How fragile is Crates.io to function-level changes?

Our goal is to identify packages that indirectly reach most of Crates.io and understand the differences and similarities in using different networks for impact analyses of package repositories. We use the local reaching centrality [83] to measure the reach of individual packages in the CDN, compile-validated metadata (i.e., `Docs.rs`), and regular metadata (Crates.io) networks. With reach, we measure the fraction of Crates.io packages that depend on a particular package (i.e., its transitive dependents).

Figure 3.8 presents the evolution of the reach of each package per network. When comparing the third-quartile between the snapshots, we can observe a gradual decrease in reachability over time. The decrease is a result of new packages being added to the

network and at the same time not being widely used by other packages. The top 25% of the distribution of the Crates.io index network has a ten-fold decrease of 0.07% in June 2015 to 0.008% in June 2019. Both the CDN and `Docs.rs` distribution also follow a similar pattern. In comparison to recent years, the higher reach of packages in the formative years reflects the small network size. In the remaining 75% of packages, they have no or limited reach of Crates.io irrespective of network choice, indicating that a majority of packages do not exhibit any influence in Crates.io. However, we can observe that the range and number of outliers expand over time, indicating that there is an increasing number of packages that exhibit a degree of influence in Crates.io. The number of outliers with greater than 10% reachability grew from 19 (`Docs.rs`/CDN: 3) to 92 (`Docs.rs`: 80, CDN: 66) packages as an example.

For each snapshot, we can see that the top-most outlier and the number of outliers is lower than that of the metadata-based network in each network. The most reachable package in June 2019 reaches 65% in the Crates.io index network, 61% in the `Docs.rs` network, and 47% in the CDN network.

Upon inspection of the top 10 highest reaching outliers in each network, we see that a similar set of packages such as `libc`, `log`, `lazy_static`, and `bitflags` remains prominent over time across the networks. These packages are also among the most directly called packages in Section 3.3.1. `libc`, one of the most downloaded packages in Crates.io, is the package exhibiting the highest all-time influence in Crates.io. There are also packages in decline: `rustc-serialize`, a serializer package, decreased in reach from its peak of 17% in 2016 to 2% in 2020. A potential explanation for its decline could be the adoption of `serde`, a rivaling serializer package, that grew its reach from 6% in 2016 to 42% in 2020.

We derive the ten most influential APIs by measuring the local reach centrality on functions of the CDN for 2015, 2017, and 2020 in Table 3.5.[12] Although `libc` exhibit the highest reach at the package-level, functions in `log` or `serde` exhibit higher influence than individual functions in `libc`. Moreover, we can see that `libc`, `log`, and `bitflags` have remained important since the inception of Crates.io. However, we can observe that the most called function changes over time. For example, `log` reports three distinctly different API functions. A possible explanation could be that new features or best practices over time change the use of APIs. Finally, we can also see a new fast-growing entrant in 2020: `serde` is second to `log`.

A large majority of packages in Crates.io have no or limited reachability; a handful of packages are reachable from 47% of Crates.io, and single functions are reachable from 30% of Crates.io.

3.5 RQ3: Reliability

We identify two occurrences with significant differences between the studied networks, namely transitive dependencies and outliers in the top-most dependent packages in **RQ2.1**. These differences have practical implications on dependency analysis use cases. For example, security-based dependency analysis such as `cargo-audit` would generally favor soundness over precision. Failing to account for an actual dependency relationship could

[12] due to presentation reasons, we showcase for only three years

lead to vulnerabilities being undetected. On the other hand, automated dependency updating such as GitHub's `Dependabot` would favor precision over soundness. False-positive updates steal valuable time from developers [84, 85]. Thus, our goal in **RQ3** is to obtain an understanding of how accurate and reliable metadata-based and call-based networks are in estimating actual relationships between packages.

Selection As packages can have many transitive dependencies and have complex use cases, manually mapping out how packages use each other in a dependency tree is a tedious and error-prone task. Attempting to scale the analysis to the entire Crates.io is also impractical. Thus, we sample dependency relationships in packages where both the metadata-based networks and the call-based networks report differently (e.g., between a package and a dependency, the metadata-network reports an edge between them, and the call-based network does not). We can then focus our manual investigation on whether call-based networks are missing function calls due to limitations with static analysis or whether metadata-based networks over-approximate unused dependencies. Moreover, analyzing a narrow set of direct and transitive dependencies further reduces the overhead of manually tracking uses of code elements across packages and their dependencies.

In the span of five workdays, we randomly sampled and reviewed 34 cases, 7 cases involving direct relationships, and 27 cases involving transitive relationships.

Review Protocol We initiate the review by first finding import statements of the direct library for the package under analysis and then track successive uses of imported items in variable assignments and definitions such as functions (e.g., return type) and `trait` implementations. After mapping out all use scenarios that trace back to the original set of import statements, we later can conclude whether a package reuses code from a dependency. The procedures for direct and transitive dependencies are slightly different. For direct dependencies, we investigate the entire package for any sign of reuse. For transitive dependencies, we inspect the context of how a package reuses its direct dependency, and whether the specific reuse of the direct dependency leads to reuse of the transitive dependency. Given the following example: Package `Foo` depends on `Bar`, and `Bar` depends on `Baz`. `Foo` also reuses `Bar`, and `Bar` also reuses `Baz`. A function `bar()` in `Bar` calls `baz()` in `Baz` and `foo()` in `Bar` does not rely on external code. If `Foo` only calls `foo()`, then `Foo` only reuses `Bar` and not `Baz` despite `Bar` reusing `Baz`. If `Foo` would call `bar()`, then there is an indirect reuse of the transitive dependency `Baz`. A step-by-step review protocol is available in the replication package.

Manual Analysis Table 3.6 tabulates the reasons for misclassification split by network and number of use cases. Overall, the metadata-based network over-approximates the dependency usage in 80% of the analyzed cases. Among direct dependencies where the metadata-based networks over-approximate, we identify seven instances where a package did not import any item from the dependency relationship under analysis. Moreover, metadata-based networks cannot distinguish dependency usage in non-runtime or conditionally compiled sections of the source code. We found two cases; one case where a developer uses a runtime dependency solely in test code and one conditional compilation case where a dependency code runs only on Windows environments.

Table 3.6: Manual inspection and classification of 34 dependency relationships between Präzi and the Crates.io index network.

Categorization	#Samples
i) Over-approximation in metadata-based networks	**27**
... no import statements	3
... import statement and no usage	4
... resides in a `#[cfg(...)] block`	1
... derive macro libraries	2
... test dependency	1
... non-reachable transitive dependency	16
ii) Under-approximation in Präzi	**7**
... importing a constant	1
... importing data type and usage	1
... importing data type in definitions	4
... handling C-function call	1
Σ	**34**

While Cargo has labels for build, test, optional, and platform-specific dependencies in the manifest file, *derive macro* dependencies are not distinguishable from runtime dependencies. A *derive macro* library performs code-generation at compile time. However, such libraries do not provide runtime functionality and are closer to the role of being a build dependency. We identify two such libraries, `cfg-if` and `thiserror`. Including such dependencies influences the count of runtime dependencies; for example, depending on the widely popular `serde_derive`[13] library would incorrectly add six dependencies to the total count of runtime dependencies. Without no specific metadata label or heuristic, a call-based dependency network avoids including such libraries.

The most prominent case with over-approximation by metadata-based networks are non-reachable transitive dependencies. The context of how a package uses its direct dependencies plays a central role in whether a package indirectly uses its transitive dependencies. As an example, the package `selfish` uses `nom v3.2.1` that then depends on `regex 0.2.11`. `nom` is a parser library and exports a set of regex parsers that uses the `regex` library. Although `selfish` enables the regex feature in `nom`, it does not import any of the regex parsers in `nom`, effectively making the `regex` library unused.

In the four cases where a developer imports type definitions from dependencies for use in function declarations. One such example is the case of importing `c_int` in `libc` for function declarations in `whereami v1.1.1`. Although a call graph does not track data references, we could still mitigate this by tracking the type declarations in argument and return types of functions in the call graph. Präzi embeds full type qualifiers including package information in functions belonging to call-based dependency networks (See Section 2.2.1).

Finally, we identify one instance where the call graph generator could not resolve a call from `subprocess v0.1.0` to the `libc` function `pipe()`. Although there is a `pipe` call

[13] `https://docs.rs/crate/serde_derive/1.0.106/source/Cargo.lock`

without clear identifiers in the call graph, it is not via the `libc` library. Thus, there are possible limitations with handling cross-language calls.

A call-based dependency network is more precise than a metadata-based network. Data-only dependencies could affect its soundness.

3.6 Discussion

We center our discussion on two key aspects; differences and similarities between using three different networks for network analyses and studying function relationships on a network level.

3.6.1 Strengths and Weaknesses between Metadata and Call-based Networks

As package repositories do not test whether a package can build or not, developers can by mistake or unknowingly publish broken versions to Crates.io. By verifying the build of package releases, the `Docs.rs` network excludes package releases that do not have a successful build record. When comparing the results of the network analyses in Figure 3.4 with Crates.io index network, overall, we find them to have comparable results except in the formative years of Crates.io. The diverging results in the initial years show that a large number of releases are not reproducible and consumable, stressing the importance of performing additional validation besides the correctness of packages manifests. Thereby, we urge researchers to minimally validate package manifests with external information such as publically available build and test data for network studies of package repositories.

When comparing the network analysis results in Figure 3.4, we find notable similarities and differences between metadata-based and call-based networks for Crates.io. Except for the formative years of Crates.io, the distributions of recent snapshots for direct dependencies, direct dependents, and total dependents are mostly similar between the networks. Thus, a network inferred from Crates.io metadata closely approximates the presence of function reuse relationships between packages without needing to construct and verify with call graphs. Recent snapshots of Crates.io further indicate that recent package releases are highly likely to be reproducible and compile as well. On the other hand, there are also significant differences between the networks, specifically for transitive dependencies and outliers in dependent distributions. By taking into account that a developer does not make use of all APIs available in a package, we identify a two-fold difference between call-based and metadata-based networks. These differences also manifest among the most popular dependent packages (i.e., outliers)—despite the networks reporting similar results for the average dependent package.

Based on these similarities and differences, we conduct a manual analysis to understand which network has a more accurate representation of package repositories. Our investigation indicates that call-based dependency networks are more precise than metadata-based networks; the prominent finding is that the number of transitive dependencies a package uses is highly contextual and moderately correlates with the number of declared dependencies. From a statistical viewpoint, we identify a strong correlation between the number of dependencies derived from a metadata-based network and the number of called

dependencies. In other words, the more resolved transitive dependencies a package has, the more transitive dependencies it will call. On the other hand, we only observe a moderate correlation between declared (direct) dependencies and called transitive dependencies, indicating that the number of called transitive dependencies potentially varies for the same number of direct dependencies. Based on our studied use cases, we find examples of packages only importing non-core functionality from libraries or specific modules of packages that use individual libraries by themselves. Despite limitations with data-only dependencies, we argue that calculating the number of transitive dependencies should not be generalized to the sum of all resolved dependencies. In line with previous work on the fine-grained analysis of known security vulnerabilities, we also argue both researchers and practitioners interested in understanding how developers or programs use dependencies should account for its context—not the number of compiled dependencies.

As a summary, we make the following recommendations based on the trade-offs and costs for constructing a call-based dependency network:

- **Direct dependencies:** Given the relative proximity of results between a metadata-based and call-based network, a metadata-based network is sufficient for use cases involving direct dependencies if precision is not crucial. The cost of building a call-based dependency network would be overly expensive.
- **Transitive dependencies:** Where transitive dependencies are central in any analysis, we recommend call-based dependency networks over metadata-based networks.
- **Data-dependencies:** Where data references are crucial to track or studying data-centric packages in Crates.io, we recommend metadata-based dependency networks or use additional (cheap) static analysis to identify data dependencies. Although metadata-based networks are imprecise, they will not miss such relationships.

3.6.2 Transitive API Usage

For studying the evolution, impact, and the decision-making of deprecation [40, 77] and refactorings [86] of library APIs, datasets such as `fine-GRAPE` [87] provide valuable insights into how a large number of clients in the wild make use of a few popular libraries. These datasets extract API usage by mining direct invocation of library APIs (i.e., a client calling a public API function). By analyzing the use of APIs in transitive dependencies of clients (i.e., indirect API use) in addition to direct dependencies, we find that there are more calls to transitive dependencies than direct dependencies in recent years. Thus, the transitive relationship where either an intermediate client or library relays a call between a client and a library could potentially present new confounding variables and implications to the evolution and decision-making of APIs. Although developers do not have control of transitive package dependencies, they have the same execution rights and follow the same laws of software evolution [88] as direct dependencies. Thus, API decisions in transitive dependencies can equally impact clients as direct dependencies.

As package managers allow the same dependency (albeit different versions of them) to co-exist in a client, our results in **RQ2.4** show growing signs that more and more copies of the same function identifier from multiple versions exist in a client. In cases where

such a function is dependent on the environment (e.g., a specific implementation of an OpenSSL library), there is a potential risk for introducing unexpected incompatibilities. Such problems that arise from the use of transitive dependencies can directly influence the decision-making of APIs. For example, a user in PR #20 of IDnow SDK,[14] an identity verification framework, is persuading the maintainers to drop dependence on Sentry, an application monitoring platform, due to the user having problems with Sentry as several versions of that dependency exist in its application.

Given the increasing growth of indirect API calls and a slight increase of multiple copies of the same function identifier appearing in clients, we call for researchers to also account for the dynamics of dependency management—particularly transitive dependencies—when studying the evolution and decision-making around APIs.

3.7 Threats to Validity

In this section, we discuss limitations and threats that can affect the validity of our study and show how we mitigated them.

3.7.1 Internal validity

For CDNs to closely mirror actual package reuse in Crates.io, we only consider packages specified under the `#[dependencies]` section and optionally-enabled packages as these are consumable in the source code. As packages in `#[dependencies]` are also available in the test portion of packages, developers could potentially specify packages for testing purposes that do not attribute towards package reuse. We mitigate the risk of inferring test specific calls by restricting the build of packages to compilation without further execution steps such as tests.

The `rust-callgraphs` generator can resolve function invocations that involve static and dynamic dispatch except for function pointer types. Although the documentation[15] states that function pointers have a specific and limited purpose, we acknowledge that we cannot make any claims around the completeness of generated CDNs due to the general absence of ground truth for package repositories. When limiting the scope to the features that the call graph generator supports, the generated CDNs represent an over-approximation of function calls in Crates.io. It is an over-approximation as function targets in dynamic dispatch may never be called by the end-user in practice (i.e., it is inexact). Using additional analysis such as dynamic analysis to remove all unlikely function targets is error-prone and could result in unsound inferences. Thus, we avoid considering both static (i.e., exact) and dynamic (i.e., inexact) function calls as the same. Instead, we view the results of dynamically dispatched calls from the perspective of virtual method tables (i.e., its concrete representation during runtime).

Real-world constraints such as non-updated caches of the repository index, user-defined dependency patches, and deviating semver specifications could influence the actual version resolution of package dependencies. The selection of packages and their versions for creating snapshots has additional implications on the representativeness of Crates.io and its users. To mitigate the risk of making incoherent versions resolutions,

[14] `https://github.com/idnow/de.idnow.ios.sdk/issues/20`

[15] `https://doc.rust-lang.org/book/first-edition/trait-objects.html`

we use the exact resolver component implemented in Cargo, ensuring the same treatment of version constraints.

Kikas *et. al.,* [1] report the highest package reach to be up to 30% in 2015 while our Crates.io metadata network report over 60%, nearly twice the number. The difference lies in the selection of packages when creating the networks: Kikas *et. al.,* [1] build a dependency tree for all available versions of a package valid at timestamp t and we build a tree for the single most recent version of a package at a timestamp t. As there is no consensus on best practices for which packages and releases to include in a network, we take a conservative approach that avoids including dormant and unused releases. For example, we argue that it is rare that a user today would declare a dependence on version dating back to 2017 when newer versions from 2019 exist. Kikas *et. al.,* [1] would include such versions.

3.7.2 External and reliability validity

We acknowledge that the results of network analysis are not generalizable to other package repositories and only explain properties of Crates.io. Due to differences in community values [16] and reuse practices of packages, we expect network analyses to yield different results. However, based on Decan *et. al.,*'s [4] comparison of seven package repositories, we believe certain repositories, for example, npm and NuGet may share some similarities with Crates.io than with CRAN and CPAN.

The Präzi approach to constructing a CDN is general applicability as long as the programming language has a resolver for package dependencies and a call graph generator. However, the soundness of generated CDNs may vary depending on the programming language. For example, CDNs generated for Java are more accurate and practical than CDNs for Python due to limited call graph support. Therefore, evaluating trade-offs in terms of precision and recall plays an important role in whether a study scenario is suitable for CDN analysis.

3.8 Future work

Our work opens an array of opportunities for future work in data-driven analysis of package repositories, both for researchers and tool builders.

3.8.1 Enabling data-driven insights into code reuse with network analysis

As functions are not the only form of achieving code reuse, we aim to explore how we can model reuse of interfaces, generics, class hierarchies, and wrapper classes as networks. In a similar spirit to enabling data-driven insights of APIs, language designers can use data-driven models to understand patterns and adoption of certain code reuse practices. As Rust advocates developers to prefer using generics over trait objects and limit the use of unsafe code constructs, language designers can verify such premises with feedback through network- and data-driven analyses of package repositories.

Following Zhang *et. al.,*'s [15] need-finding study on data-driven API design, we are investigating possibilities to mine program contexts and error-inducing patterns using Präzi to extract API usage patterns beyond syntactic features and frequencies. Insights

into involved API usage patterns can help library maintainers to make changes echoing improvements that simplify code reuse and strengthening the stability of a package repository.

3.8.2 Modeling socio-technical risks of package abandonment

Package repositories are successful in attracting developers to release new packages. However, they are less successful in keeping these packages maintained on a long-term perspective. As a result of developers abandoning packages due to shifting priorities, unmaintained packages are increasingly jeopardizing the security and stability of package repositories. Notably, the *event-stream* incident [12] is emerging as a textbook example of how the abandonment of a package turned itself into a bitcoin stealing apparatus affecting thousands of users. While survival analysis of packages can yield insights into the stages of abandonment [31], understanding the social-technical motives behind developer abandonment could potentially help develop a risk control model that package repository owners can exercise. As an example, when a package repository recognizes the slowdown of development activities of popular yet central packages, they could explore incentives such as monetary support, developer assistant in resolving long-running bug reports, or discuss possible handover to a network of trustful developers. We are exploring both quantitative and qualitative strategies on how to model and mitigate risks around package abandonment using PRÄZI.

3.9 Conclusions

In this chapter, we have comprehensively analyzed the structure, evolution, and software reuse patterns in package repositories, using CRATES.IO as our case study object. This analysis involved exploring metadata, compile-validated metadata, and call-based networks, helping us better understand the accuracy and reliability of these networks in executing various dependency tasks.

Despite the increased number of imported transitive dependencies, including the increase of API calls to direct and indirect dependencies, we find that packages do not invoke 60% of their transitive dependencies. When comparing the three networks approximating CRATES.IO, we find that networks approximate each other and are interchangeable for analysis relating to direct dependencies. However, we identify substantial differences in analysis involving transitive dependencies where the context of how a package uses its dependencies influences which transitive dependencies it calls.

By utilizing a CDN, we can conduct a detailed code-centric analysis of package repositories, tracking code bloat, monitoring deprecation, and studying dispatch types such as monomorphic and polymorphic call sites. Comparing the three networks reveals that CDNs can potentially cut down on false positives in dependency checkers like Rust's `cargo-audit` and GITHUB's `Dependabot`, especially when it comes to analysis tasks involving transitive package relationships. Here, we establish that networks that do not infer from source code significantly overstate package relationships.

4

Can We Trust Tests to Automate Dependency Updates? A Case Study of Java Projects

Automated dependency update services, such as Dependabot, are increasingly popular, but developers find them unreliable due to their heavy reliance on test coverage to detect conflicts. We analyzed the test coverage of direct and indirect dependency uses in 521 well-tested Java projects to investigate this issue. Our study reveals that tests cover 58% of direct and 20% of transitive dependency calls. Further, we created over 1.1 million artificial updates with simple faults across 262 projects, assessing the effectiveness of test suites in detecting dependency-related semantic faults. The results show that tests only detect 47% of direct and 35% of indirect faults on average. To improve this, we explored the use of change impact analysis to reduce false negatives. Our tool unveiled 74% of faults in direct dependencies and 64% in transitive dependencies, nearly doubling the detection rate of conventional test suites. We then applied our tool to 22 real-world dependency updates, identifying three cases of semantic conflict and five unused dependencies. These findings suggest integrating static and dynamic analysis in future dependency updating systems for more reliable results.

Modern package managers facilitate reuse of open source software libraries by enabling applications to declare them as versioned dependencies. Crucially, when a new version of a dependency is made available, package managers will automatically make it available to the client application. This mechanism helps projects stay up-to-date with upstream developments, such as performance improvements or bug fixes, with minimal fuss. Typically, package managers implement a set of interval operators (dependency version ranges) on top of the SemVer protocol [89] that developers use to declare update constraints. For example, a dependency declared with the range >= 1.0.0 < 1.5.0 restricts updates to backward-compatible changes up to 1.5.0. On the other hand, >= 1.0.0 welcomes automatic updates of all new version releases starting from 1.0.0. Given a new library release with version 1.5.0, the latter constraint will allow an update but the former will not.

In practice, most package managers use a liberally interpreted version of the SemVer protocol with no vetting, allowing library maintainers to release new changes based on their self-interpretation of backward compatibility [16, 89]. As a consequence, client programs may unexpectedly discover regression-inducing changes, such as bugs or semantic changes that break code contracts. Discovering, debugging and resolving such issues, as exemplified in Figure 1.4, remains a challenging task for development teams [16]. In fact, unexpected regressions are one of the main reasons that deter developers from upgrading dependencies to new versions [6].

Developers can mitigate the risk of integration errors by either using restrictive strategies, such as *version locking*, or permissive strategies involving *dependency update tooling*. Version locking effectively makes the dependency tree of client programs immutable and disables automated updates. This strategy offers maximum stability but is prone to incurring technical debt due to outdated dependencies. Moreover, developers need to manually discover and apply security hotfixes. On the other hand, dependency update checkers analyze version compatibility before deciding to update. There are two main techniques for deciding version compatibility, *breaking change detection* [34, 90, 91] and *regression testing* [92, 93]. Detecting potential breaking changes (i.e., backward API incompatibilities) prevents client programs from updating to versions that will result in compile failures. A major shortcoming of this technique is that it depends on the compilation and the existence of a static type system; many of today's most popular languages are dynamically typed. A more popular option among developers is the use of services providing automated dependency updating, such as `greenkeeper.io` [94], `Dependabot` [95], and `renovate` [96], that use project test suites to detect regression changes on every new update.

The effectiveness of such services depends highly on the quality of end-users test suites [97]. Poor test coverage of dependency usage in client code can lead to missing update-induced regressions. Recent studies [98, 99] suggest that high statement coverage in test suites does not guarantee to find regressions in code changes. Failing to detect regressions stemming from updates can have dire consequences for client programs: for example, users dependent on NPM's `event-stream` package did not notice a malicious maintainer planting a hidden backdoor for stealing bitcoin wallets inside the library's source code [12]. Moreover, a recent qualitative study [84] also revealed that developers are generally suspicious of automatically updating their dependencies. One of the prime reasons is that developers perceive their tests as unreliable.

To mitigate the risks of undetected regression changes, this chapter proposes the use of static change impact analysis for dependency updates, referred to as UPPDATERA. By statically identifying changed functions and approximating call-relationships between an application and its dependencies, change impact analysis can fill in gaps where test suites have limited coverage or cannot reach.

In this chapter, we set out to empirically understand how reliable developer tests are in automated dependency updating by addressing the following research questions:

- **RQ1:** *Do tests cover uses of third-party libraries in projects?*
- **RQ2:** *How effective are project test suites and change impact analysis in detecting semantic changes in third-party library updates?*

- **RQ3:** *How useful is static analysis in complementing tests for compatibility checking of new library versions?*

This work examines the prevalence of tests in projects exercising dependencies and measures the coverage of these tests. The study uses systematic mutation of dependency usage in multiple projects to determine the adequacy of test suites and change impact analysis in detecting artificial updates with simple faults. We evaluate the performance of test suites and Uppdatera on multiple pull requests that update dependencies to understand the strengths and weaknesses of static analysis as a complement to tests.

The findings of this work indicate significant gaps in test coverage of function calls in projects that use library dependencies. While static analysis shows a higher degree of effectiveness in covering these gaps, it is prone to false positives due to difficulties evaluating over-approximated execution paths. These results underscore the risks of automated dependency updating and suggest that tool creators should consider incorporating hybrid workflows to cover gaps in regression testing with static analysis and assist developers in prioritizing testing efforts.

This chapter contributes to the ongoing discourse around automated dependency updating. It puts forward empirical evidence to advocate for a more considered approach to dependency updating that balances automated tooling with human oversight and expertise.

4.1 Background

4.1.1 Package Managers

Package managers such as Java's Maven, JavaScript's npm, or Rust's Cargo provide tooling to simplify the complexities of maintaining, distributing, and importing external third-party software libraries in development projects. As a community service to its users, package managers also host a public Online Package Repository (OPR) where developers can freely contribute with new packages (e.g., a database driver) or build upon existing packages (e.g., use a parser library to build a JSON parser). This helps package manager users to reduce development efforts by benefitting from existing functionality in their language environments. In a nutshell, a package is a distributable, versioned software library.

Because of the relative ease of building packages on top of each other, OPRs today grow quickly and become evermore inter-dependent [1, 100]. As a consequence, package manager users experience a dynamic growth of new hidden dependency imports in their projects and frequent dependency updates that increase the risk of build failures due to breaking backward compatibility [2, 4, 16, 30]. The risk of breaking backward compatibility varies between OPRs: npm and Maven Central move the burden of checking incompatible changes on its users while R/CRAN minimize this risk by requiring a mutual change-cost negotiation between library maintainers and their users [16]. Users of OPRs, such as npm or Maven, either uses additional tooling or disable dependency updates through version-locking as a protective measure. Version-locking dependencies guarantee a stable build environment. Additional tooling provides an extra layer control by scanning dependencies for vulnerabilities [30], freshness [101] or update compatibility [34, 91].

```
1  package client {
2      import p2.B;
3      class Main {
4          void int main {B.b(); B.z();}
5      }
6  }
7  package p2 {
8      import p1.A;
9      class B {
10         int b() {
11             int y = 1;
12             if A.v(y){y+2;}
13             int x = A.a();
14             if x > 0 {return 0;}
15             return x + y;
16         }
17         //...
18     -   bool z() {return false;}
19     +   bool z() {return make_false();}
20         //...
21     +   bool make_false() {return false;}
22     }
23 }
24 package p1 {
25     class A {
26         //...
27     -   int a() {return 0;}
28     +   int a() {return 1;}
29         //...
30     -   bool v(int a) {a > 0 ? true : false}
31     +   bool v(int a) {a == 0 ? true : false}
32     }
33 }
```

Example 4.1: Changes in dependencies that break client semantics

4.1.2 Safe Backward Compatible Updates

Update checkers such as `cargo-crusador` [102], `JAPICC` [103], and `dont-break` [104] typically determine backward compatibility by ensuring that the new version is consistent with the public API contract of the old version. Removals or changes in method signatures, access modifiers, and types (e.g., classes and interfaces) are examples of inconsistencies that can lead to compile failures in client code [11, 105].

Checking dependency updates for API inconsistencies is a necessary precondition to a safe update, but not a sufficient one. From Example 4.1, we consider an additional class of changes, *semantic changes*, that are API-compatible (i.e., respects the public API contract) but introduces incompatible behavior (i.e., regression changes) for clients after dependency updates. The code example illustrates a `client` that depends on `p2` which in turn depends on `p1`. There are two changes that are not semantic preserving in `p1`: `a()` returns 1 instead of 0 (line 27-28) and `v(int a)` compares variable `a` with a different comparison operator (line 30-31). On the other hand, the change in `p2` is semantic preserving: `z()` still returns `false` despite replacing it with a method call to `make_false` (line 18-21). Given a scenario in which `client` automatically updates to the next release of `p1`, and `p1` updates to the next release of `p2`. The changes made in `p1` will indirectly impact the behavior of

`client` despite seeming hidden and distant. The change in `a()` of `p1` results in `b()` to match the if statement on line 14 and return 0 instead of doing an addition of `x` and `y` in `p2` (line 15). This further propagates to the `client` where `b()` is called. Similarly, the change in `v()` flips the condition to `false` instead of `true` in `p2` which result in skipping `y+2` at line 12. These two code changes illustrate how the client behavior or the execution flow is not honored after updating to a newer version.

Unlike breaking API contracts, semantic changes are not inherently bad: the refactoring of `z()` in `p2` introduces a new execution path (e.g., new behavior) to `make_false` which continues to return `false` after the change. Source code changes that preserve the same behavior before and after an update are semantic backward compatible changes. Deciding semantic backward compatibility is also a contextual problem: Given another client, `client2` that use the same dependency `p2` as `client` but don't call `b()` and `z()` (line 4). The same update we illustrate for `client` is semantic backward compatible for `client2` as it functions the same way before and after the update.

Following the observations in Example 4.1, we denote a semantic backward compatible update or *safe update* as the following: We denote $Lib_1, Lib_2 \in Library$ as two versions of the same library and a client C with dependency tree as $T_C = (V, E)$ where V is a set of resolved versioned libraries used by C, and E is the directed dependence between them. Let PDG_{TC} represent a sound *program-dependence graph* [106] of T_C connecting data and control dependencies between program statements in both client and dependency code. The transition $[Lib_1 \longrightarrow Lib_2]_C$ represents replacing Lib_1 with Lib_2 in client C. We arrive at the following definition of a safe update:

Definition 4.1.1. Given that $Lib_1 \in T_C$ and a request by a package manager to perform $[Lib_1 \longrightarrow Lib_2]_C$, let $D = Lib_1 \setminus Lib_2$ be a source code diff mapping between Lib_1 and Lib_2, and function $f : D \longrightarrow Y$ determine semantic compatibility for diff $d_i \in D$ in client C where $Y \in \{true, false\}$, an automatic update (or safe update) can only be made if and only if $\forall d_i \in D, f(d_1) \wedge f(d_2) \ldots \wedge f(d_n) = true$ where i varies from 1 to n and n is the cardinality of set D.

4.2 Research Questions

The goal of this paper is to understand how reliable test suites are as a means to evaluate the compatibility of updated library versions in projects. To that end, we study a large number of test suites from Maven-based Java projects that depend on external libraries.

Bogart *et. al.,* [16] report that developers create strategies to select high-quality libraries based on signals such as active contributors, project history, and personal trust in project maintainers to reduce the exposure of unwanted changes. Thus, in our first research question, we investigate whether testing of third-party libraries is prevalent and a strategy to minimize the risk of breaking changes:

RQ1: Do test suites cover the uses of third-party libraries in projects?

Mirhosseini *et. al.,*'s [84] qualitative study suggests that developers have trust issues with automated updates and perceive tests as unreliable. A compelling complement to evaluate the effect of dependency changes is the use of change impact analysis. We set to measure how capable both test suites and change impact analysis can catch simple semantic faults in both direct and indirect uses of third-party libraries:

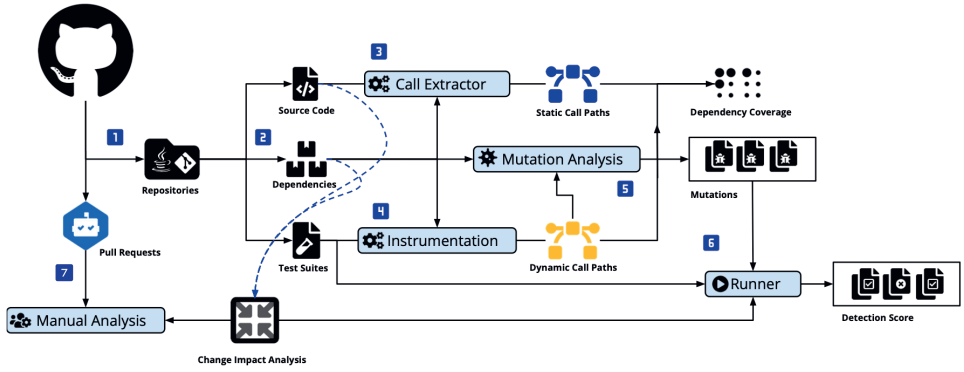


Figure 4.1: Overview of our study infrastructure

RQ2: How effective are project test suites and change impact analysis in detecting semantic changes in third-party library updates?

While static analysis can yield higher coverage, it is also more prone to false positives by classifying safe updates as unsafe. To understand the strengths and weaknesses of static analysis in a practical environment, we ask:

RQ3: How useful is static analysis in complementing tests for compatibility checking of new library versions?

We extract a set of real-world update cases from pull requests generated by the popular service `Dependabot` and manually investigate the correctness of each pull request. Then, we analyze each pull request using change impact analysis to compare the results with the test suite and our ground truth.

4.3 Research Method

We follow the study design depicted in Figure 4.1 to evaluate the reliability of test suites for automated dependency updating and the potential of using static analysis. First, we select Java repositories with high-quality assurance badges and at-least one test class from GitHub [1]. Then, we build each repository to infer a complete dependency tree of the project along with its source- and test classes in [2]. Second, we feed the source classes together with the dependencies of a project to the call extractor and statically extract all its direct and indirect uses of third-party libraries [3]. Third, we use instrumentation to learn all invocations from a project to its dependencies via its test suite [4]. Then, we use the information from the previous step to calculate the dependency coverage of a project. Fourth, we generate mutations of dependencies by inserting simple faults (See Table 4.1) in dependency functions executed by tests. Here, we use dynamic call paths (from [4]) to identify such functions [5]. We can then run both the test suite and the change impact analysis to measure the detection score [6]. Finally, we harvest `Dependabot` pull requests in a real-time fashion and then manually evaluate how both test suites and change impact analysis perform in practice [7].

4.3.1 Identifying Usages of Third-Party Libraries

We refer to the use of third-party libraries as using functionality from externally-developed libraries in software projects. Specifically, we focus on functionality exposed as functions in libraries as they are among the most widespread forms to achieve code reuse. Thus, we consider a function call from a project to a library dependency as third-party library use. As projects depend on an ordered tree of library dependencies, there are both implicit and explicit third-party uses. An explicit use is a direct function call between a project and one of its declared libraries. On the other hand, implicit use is when a function in a project transitively calls underlying libraries in a dependency tree. Given the following example scenario: project `A` depends on library `B`, and library `B` depends on library `C`. If there is a function call path between a function `a()` in `A` to a function `c()` in `C` via called functions in `B`, project `A` is implicitly using functionality in library `C`.

To identify explicit use of third-party libraries, we statically extract all function calls to functions that are neither part of the project under analysis or the Java standard library. By deduction, all such method invocations represent calls to third-party libraries. For implicit use of third-party libraries, we statically derive call graphs capturing call paths between a project and its dependency tree, similar to Ponta et al [36]. Finally, we prune function and call sequences belonging to the Java standard library to derive a graph representing interactions between a project and its transitive dependencies.

To measure dependency coverage in a project in **RQ1**, we use instrumentation to record all project invocations to third-party libraries during test suite execution. Using the recorded set, we calculate the proportion of statically inferred functions covered by the test recorded set of function as dependency coverage ($Recorded\ functions \subset Declared\ functions$):

$$Cov_{dep} = \frac{Recorded\ functions}{Declared\ functions}$$

Effectively, dependency coverage is function coverage [107], but only restricted to dependency calls.

4.3.2 Heuristics for Static Impact Analysis

The central task of automated dependency updating is to facilitate the continuous integration of new compatible library versions with minimal developer intervention. Unlike static analysis that may contain false warnings [108], automated updating suffers instead from false negatives. A faulty update has a potentially high maintenance penalty if merged into the project and could cascade into breaking the build of externally depending projects.

As a step towards reducing false negatives, we are investigating change impact analysis as a means to potentially reduce coverage gaps where tests are not able to reach in dependencies. Change impact analysis estimates the reach and fraction of affected execution paths in a program given a set of code changes [109]. While there are advancements towards inference of semantic changes in static analysis such as data flow analysis with equivalence relations [110] and mining techniques [45], precise static interpretation of semantic changes such as faulty updates is an undecidable problem [111]. Moreover, most of these techniques only analyze method bodies, and thus not practical for inter-procedural analysis of projects and their dependencies.

Table 4.1: Mutation operators (based on Papadakis et al [114])

Names	Description	Example
ABS	Absolute Value Insertion	$v \longmapsto \texttt{abs}(v) \mid \texttt{-abs}(v) \mid 0$
AOR	Arithmetic Operator Replacement	$x\ \texttt{op}\ y \longmapsto x\ \texttt{+}\ y \mid x\ \texttt{\%}\ y \mid x\ \texttt{/}\ y$
LCR	Logical Connector Replacement	$x\ \texttt{op}\ y \longmapsto x\ \texttt{||}\ y \mid x\ \texttt{\&\&}\ y \mid x\ \texttt{\^{}}\ y$
ROR	Relational Operator Replacement	$x\ \texttt{op}\ y \longmapsto x\ \texttt{>}\ y \mid x\ \texttt{!=}\ y \mid x\ \texttt{>=}\ y$
UOI	Unary Operator Insertion	$v \longmapsto v\texttt{++}, \texttt{++}v, \texttt{!}\,v$

Without possibilities to precisely determine if an update is faulty or not, we approximate a faulty update (or semantic change) as a change to the execution flow of a project. We use control flow graphs (CFGs) [112] to represent all possible execution paths of functions. There are two types of statements in CFG terminology that affect the execution flow of a program, namely *control* and *write* statements [113]. A change to a write statement can affect the program state (i.e., assign a value to a variable). A change to a control statement changes the program counter (i.e., determine which statement to be executed next). By reading the program state, changes to the two other statements passively impacts *read* statements. Thus, we derive the following heuristics to classify unsafe updates:

Definition 4.3.1. Given a diff mapping $D = Lib_1 \setminus Lib_2$ between code entities in Lib_1 and Lib_2, we consider a code change as not *semantic preserving* if and only if $d_i \in D$ has a source location with a reachable control flow path to client C and maps to the following potential actions in a CFG:

1. d_i translates to change in the expression of *write* or *read* statements (data-flow change)
2. d_i translates to moving a statement from position x to y (control-flow change)
3. d_i translates to removing or expanding with new control flow paths (control-flow change)
4. d_i translates to changes in branch conditions (control-flow change)

The definition is an over-approximation; code changes such as refactoring would result in being classified as an unsafe update if and only if affected functions are reachable. As services such as `Dependabot` present only the outcome of test results and a changelog between the old and new version of a library, change impact analysis instead precisely pinpoint affected execution paths in an update. Such information help project maintainers prioritize testing efforts or determine the potential risk of the update.

4.3.3 Creating Unsafe Updates in Project Dependencies

For seamless integration, it is important for automated dependency updating to detect incompatibilities that arise when updating a library dependency. By using mutation analysis to seed artificial faults in all uses of third-party libraries in a project, we can derive

an adequacy test of detecting incompatibilities in automated dependency updates. We first dynamically extract a set of called third-party functions in a project and then apply mutation operators defined in Table 4.1 to construct a set of artificial updates that are false negatives. As static analysis can over-approximate execution paths (i.e., risk creating false-positive cases), we resort to dynamic analysis to ensure mutations of truly invoked functions. For the selection of mutation operators, we choose operators common in mutation testing studies [114, 115] that focus on simple logical flaws and exclude mutation operators with a limited effect such as deleting statements [115].

In comparison to using actual update cases, the mutation setup provides a systematic way to introduce simple faults in **all** uses of third-party libraries in a project to measure the effectiveness of detecting faulty updates. Manually curating false-negative cases of dependency updates limits to specific project-library pairs and may not generalize to other projects that use the same library. Moreover, finding such pairs for all libraries in a project to create an overall assessment may not be possible in most projects.

For **RQ2**, we denote *mutation detection score for dependencies* (an adaption of *mutation score* [116]) as a tool's ability to detect a mutated reachable dependency function as (mutants):

$$Detection\,Score = \frac{Detected\,mutants}{All\,mutants}$$

4.3.4 Manual Analysis of Pull Requests

As the artificially created updates address only false negatives, we also need to understand how static analysis performs in practice. Thus, we manually analyze the applicability of static analysis using pull requests through a lightweight code review. Due to the absence of established ground truth or a benchmark, we resort to manually creating a ground truth of libraries under update. As understanding the use context of a project-library is challenging, we also, attempt to corroborate our findings by posting our assessment as pull request comments. Below, we define our setup for the manual analysis:

Selection criteria We select pull requests generated from the popular service `Dependabot` on GitHub that supports automated updates of Java projects using the Maven-build system. To select significant and high impactful projects and increase the chance for a response by a project maintainer, we harvest newly created pull requests using GHTorrent's event stream [117] and adopt the following filter criteria: (1) *high stargazer, watchers or forks count* indicate popularity, (2) *no passive users* indicate projects that assign reviewers and frequently merge `Dependabot`-pull requests, (3) *dependency type* indicates that we only consider Maven compile and runtime dependencies, and (4) *project buildability* indicates that we can compile the project out of the box.

Code review protocol After a pull request meets the selection criteria, we first inspect the diff in the pull request to identify the old and the new version number of the library under update. Then, we download the source jar of the old and new version from Maven Central and use a diffing tool to localize the set of changes. By reviewing the change location, consulting the changelog, inspecting the tests of the library, we classify the nature

of a change as refactoring, structural (i.e., breaking change), or behavioral (i.e., semantic change). Next, we check out the project at the commit described in the pull request and manually localize uses of the library by first performing keyword search of import statements leading to the library under update. Then, we track the data- and control-flow of imported items (e.g., object instantiations, function invocations, and interface implementations) to map out how the project uses the library under update. If the library under update is a transitive dependency, we first trace how the project uses its direct dependency and then how the used subset of the direct dependency uses the transitive dependency. After mapping out uses of the library under analysis, we can then establish whether a project directly or indirectly uses any of the changed classes and function signatures identified in the diff and whether those changes make the update safe or not. If the changes do not alter the logic (e.g., refactorings) of the project, we consider the update safe. Refactorings are in some cases highly contextual and can yield different outcomes as exemplified in the following: the changed function `foo(x)` adds a new IF-statement with the condition `x > 50` that breaks the original functionality. Project `A` uses `foo(x)` indirectly, and through the manual analysis (including inspection of its tests), we can establish that the threshold is $x < 20$ in all cases, and thus the update is safe to make. On the other hand, project `B` has a public function `bar(x)` that passes `x` in a function call to `foo(x)`. Here, we cannot assume anything around `x` as users of `B` could call `bar(x)` with any `x`. In this case, we consider the update unsafe.

After manually evaluating pull requests, we classify them using one of three categories:

- *Safe*: the update is safe to perform and will not negatively impact the functionality of the project.
- *Unsafe*: the update is risky and could lead to potential unexpected runtime changes.
- *Unused*: the update of an unused dependency (i.e., it is only declared in the project but not used).

Based on the outcome of the update tooling, we compare it with the classification above and consider the following:

- False Negative (FN) when classifying an unsafe update as safe.
- False Positive (FP) when classifying a safe update as unsafe or falsely updating an unused dependency.
- True Positive (TP) when both our manual classification and update tooling has the same conclusion.
- True Negative (TN) when not creating an update for an unused dependency.

4.3.5 Dataset Construction

We sample 1,823 repositories from GitHub that have Java as the primary language, Maven as the primary build system, and have at-least a high-quality assurance badge (i.e., Travis CI, CodeClimate, coveralls, and CodeCov) as a signal for having tests [118]. Services such as `Dependabot` can update dependencies in projects as long as there is a valid `pom.xml`

Table 4.2: Descriptive Statistics for 521 GitHub projects (each variable aggregated per project)

Variable	Unit	$Q_{0.05}$	Mean	Median	$Q_{0.95}$	Histogram
Project Methods	count	20	668	210.5	2320.5	
Test Coverage (function calls)	%	7	72	64	97	
Direct Dependencies	count	1	10	7	31	
Transitive Dependencies	count	1	31	16	105	

file. Next, we build and then dry-run projects on both the instrumentation and mutation pipeline to eliminate incompatible projects. In total, there are 818 repositories that compile to Java 8 bytecode and have at least one compiled test class. Out of the 818 built projects, 521 projects successfully run the instrumentation pipeline, and a subset of 262 projects are compatible with the mutation pipeline. The number of projects in the mutation pipeline is nearly double the ratio of a recent previous study [119]. Table 4.2 presents descriptive statistics on four aggregated variables for projects belonging to the instrumentation pipeline. The median number of declared methods is 210 (mean: 668) with a heavily positive skewed distribution. 75% of all projects in our sample cluster around 588 or less declared methods with 36 projects having more than 1400 methods. The largest project is `oracle/oci-java-sdk` with 22,264 methods. As per Section 4.3.1, we measure test coverage of all function calls made in a project. We can observe that the test coverage is generally high: half of the projects have coverage of 67% or more. For the number of dependencies, we can observe that the distribution does not drastically change: the median changes from 7 to 16, indicating a small expansion of transitive dependencies. Overall, our dataset represents mid-sized projects that use a significant number of dependencies with varying test coverage.

4.3.6 Implementation

We discuss the implementation of Uppdatera, a tooling for performing change impact analysis of library dependencies in Maven, and our pipeline to run our experiments. We have open-sourced the tooling and docker images for automation and reproducibility of our study (see Section 4.5.3).

Uppdatera

Given a request to update a dependency to a new version in a `pom.xml` file, Uppdatera first performs AST differencing of the current and new version of the dependency to identify a list of functions with potential behavioral changes using `SpoonLabs/GumTree` [120]. Then, Uppdatera computes a call graph inferring all control-flow paths between client and dependency functions following Ponta *et. al.,* [36] approach for call graph construction (using WALA). Finally, Uppdatera performs a reachability analysis using the list of possible behavioral changes on the call graph to find reachable paths to the client code. Figure 4.2 demonstrates an example of using Uppdatera for updating the library `io.reactivex:rxjava` from version 1.3.4 to 1.3.8 in `opentracing-contrib/java-rxjava`. The report features a call stack to the changed function along with a set of AST diffs. In this

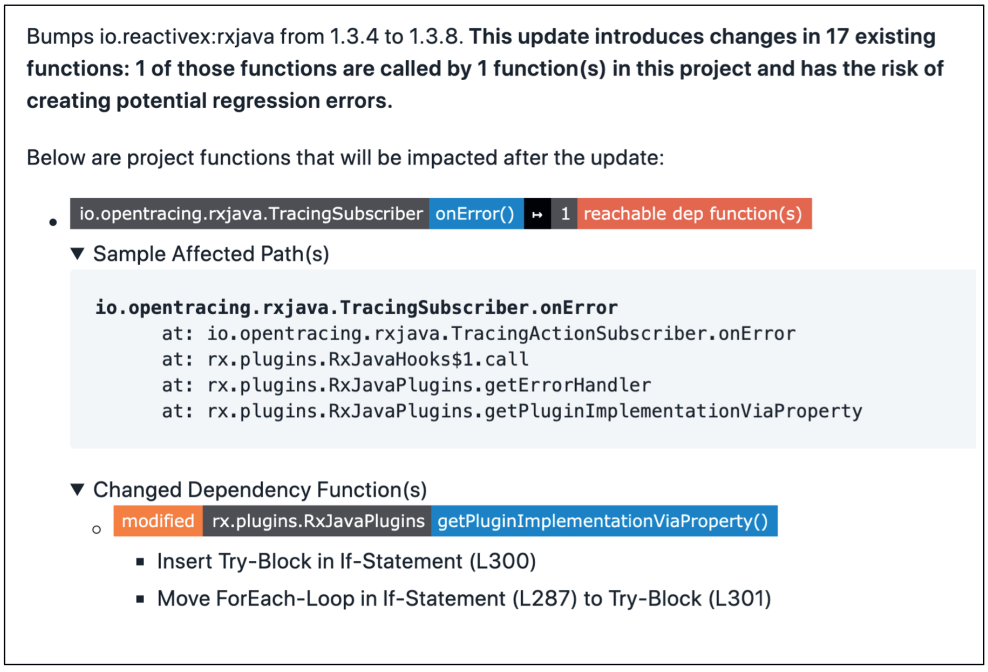


Figure 4.2: Example of updating `rxjava` from 1.3.4 to 1.3.8 in the project `opentracing-contrib/java-rxjava`

particular case, the `onError()` function in the class `TracingSubscriber` transitively calls `getPluginImplementationViaProperty()` in the dependency class `RxJavaPlugins`. The addition of a `try-catch` block in the function takes care of unhandled exceptions which may have been handled by clients in previous versions (i.e., potential regression change)

In the following, we motivate our implementation choices for a change impact analysis tool designated for updating library dependencies.

Diffing Uppdatera performs source code differencing at the abstract syntax tree (AST) level of both the current and the new version of a dependency to identify functions with code changes. AST differencing algorithms [120, 121] produce fine-grained and accurate information about the type and structure of source code changes. Following *Definition* 4.3.1, we capture AST transformations at the statement level and map the following as regression changes:

- Any method-level *move* operation mirrors moving a statement from line x to y.
- *deletion*, *update* or *insertion* of *Expression* ASTs mirrors data-flow changes.
- *deletion*, *update* or *insertion* of control struct ASTs such as *IF*, *While*, *FOR* mirrors control-flow changes.
- *deletion*, *update* or *insertion* of *Call-Expression* ASTs represents changes mirrors control-flow changes.

As an alternative to AST differencing, we could consider bytecode differencing. Bytecode (e.g., LLVM's IR or JVM code) differencing compute edit scripts at the instruction level. Although this technique offers a fine-grained and a compelling alternative to AST differencing, instruction-level changes can be difficult to understand for developers not familiar with low-level details.

Call Graph Construction Uppdatera constructs a call graph capturing inter-procedural control-flow paths between client and dependency functions. Each node in the call graph represents a fully resolved function identifier and should be identical to the identifiers in the changeset of the **Diffing** phase.

We advocate the use of call graph algorithms that are both *soundy* [50] and scalable for analyzing projects in the wild as a general guideline. The call graph algorithm should support and resolve as many language features as possible. Limited support of language features could potentially leave gaps in the coverage of projects making use of unsupported features. Similar to static analyses of security applications, achieving high recall is more crucial than precision to avoid recommending faulty updates.

As recent studies [1, 4] suggest that irrespective of the OPR, the majority of packages have a small number of direct dependencies, but a high and growing number of transitive dependencies. For example, 50% of all packages in Crates.io have a dependency tree depth of at least 6 [4]. Therefore, performing static analysis at the boundary of a project and its dependency tree can become computationally expensive and impractical in DevOps environments. Moreover, as Uppdatera can expect to analyze any compatible project in the wild, the algorithm should be scalable to cater large projects and cheap to construct to cut down computation time.

Finally, a potential trade-off of using call graphs instead of CFGs is the loss of analysis precision due to the absence of data-flow paths in the graph. However, taking into account program features such as aliases, arrays, structs, and class objects in dataflow analysis adds additional complexity and scalability problems when moving the analysis boundary to include project dependencies. Supporting such analysis adds extra precision but may not yield extra actionability.

Reachability Analysis For each changed function identified in the **Diffing** phase, Uppdatera performs a reachability analysis on the call graph to detect paths connecting changed dependency functions to functions in the analyzed project. If Uppdatera finds such paths, it marks the update as potentially *unsafe*. If no such paths are found, Uppdatera marks it as a potential *safe* update and recommends the update to the package manager. Finally, Uppdatera also reports the impacted paths between dependencies and project functions, to inform developers of the program paths that need to be inspected in response to an update in a dependency.

Experimental Pipeline

To implement our methodology, we first develop a call extractor that records complete call sequences between a project and its library dependencies. The implementation builds on instrumenting library classes using ASM [122] and the Maven Dependency Plugin. To

infer function calls to libraries from a project (**RQ1**), we use ASM to statically extract call sites for direct dependencies. We generate call graphs using WALA [123] configured for the CHA algorithm for transitive dependencies. Following Reif *et. al.,* [124]'s comprehensive benchmark of call graph algorithms for Java, we find that the CHA algorithm supports the most language features and has a lower runtime in comparison to more precise points-to analysis algorithms such as 0-1-CFA or N-CFA.

For **RQ2**, we implement the update emulation pipeline (i.e. mutation analysis) on top of PITest [125], a popular in-memory-based mutation testing framework that works with the popular test runners JUnit and TestNG by limiting mutations to library functions identified from the call extractor. We exclude the use of experimental mutation operators that cannot guarantee non-equivalent mutations. For each mutated class, we use Procyon [126] to decompile into a source file for AST diffing in the case of Uppdatera.

4.4 Results

Here, we report the results of our research questions.

4.4.1 RQ1: Dependency coverage

Figure 4.3 presents a violin plot of dependency coverage on the left-hand side, and dependency coverage including transitive dependencies on the right-hand side. Overall, 13% (67/521) projects have less than 10% coverage, suggesting at large that a majority of projects have some tests exercising at least one dependency use. We observe that the median coverage is 58% (mean: 55%): half of the GitHub projects miss coverage of more or at least 42% of all dependency function calls. In practice, there is a risk that automated dependency updating may not have tests that exercise changes in dependencies.

The right-hand side of Figure 4.3 shows the dependency coverage taking into account reachable paths to transitive dependencies in projects. The distribution has a bimodal shape with two peaks at, 9%, and at 52%, suggesting two classes of projects. In the first class, half of the projects have a median dependency coverage of 21% (mean: 26%), indicating that project test suites at large do not exercise dependencies in depth. This is not surprising: an ergonomic factor of third-party libraries is that they are well-tested and should in principle not need extra tests [127]. In the second class, we can observe that projects have tests that exercise dependencies in-depth, suggesting the presence of projects with adequate test suites. As mentioned in Section 4.3.1, these results are indicative as we compare against statically inferred call paths, which, being over-approximating, may not be representative of actual calls.

Findings from RQ1: *Half of the 521 projects exercise less than 60% of all direct dependency calls from their tests; this drops to 20% if paths to transitive dependencies are considered.*

4.4.2 RQ2: Detecting Simple faults in Dependencies

Our benchmark generated in total 1,122,420 artificial updates for 311 Maven modules belonging to 262 GitHub projects. Figure 4.4 shows a violin plot of the mutation detection score for both direct and transitive dependencies, split by project test suites on the left-

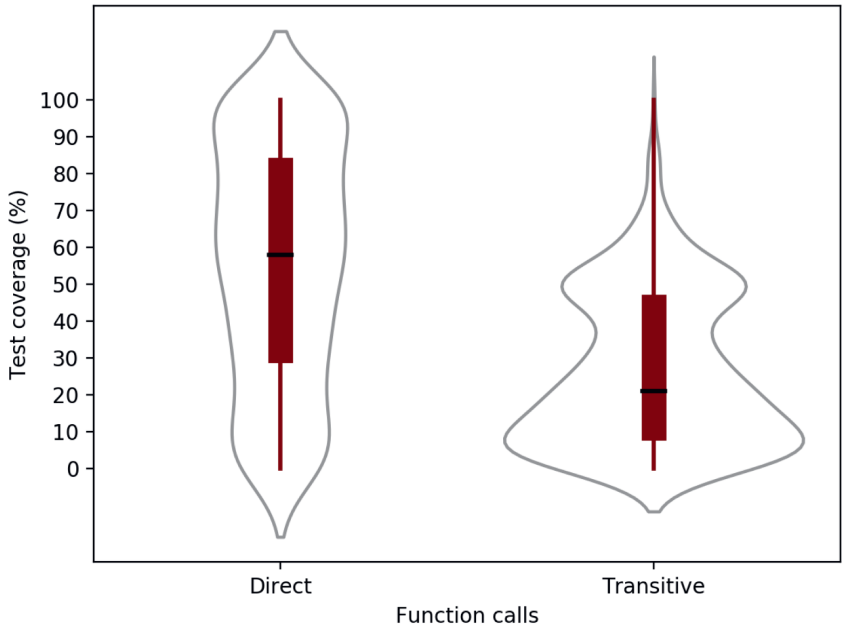


Figure 4.3: Test coverage of dependencies

hand side and UPPDATERA on the right-hand side. The median detection rate score is 51% (mean: 47%) for direct dependencies and 36% (mean: 35%) for transitive dependencies. We can observe that 25% of the projects have a high test suite effectiveness greater or equal to 80% for direct dependencies. When looking at transitive dependencies, the median of direct dependencies and the third-quantile of transitive dependencies are similar, showing that only one-fourth of the test suites remain effective in detecting faults in transitive dependencies. Moreover, we can also see more dispersion in effectiveness among direct dependencies than transitive dependencies, half of the projects have a detection score raging between 16 to 54% for transitive dependencies. Overall, the results indicate that tests are effective for a limited number of cases and dependencies. At large, however, a small minority of projects have test suites that can comprehensively detect faulty updates.

On the other hand, UPPDATERA, has a median detection score of 97% (mean: 74%) for direct dependencies and 88% (mean: 64%) for transitive dependencies. Generally, we see that static analysis is highly effective in detecting simple faults with a slightly decreased effectiveness for transitive dependencies. Half of the projects with a low detection score (< 50%) using tests now have detection score greater than 80%. In the lower half of the median for both direct and transitive dependencies, we see large variations between the projects. As change impact analysis is largely a generic technique, we manually investigate why UPPDATERA was unable to detect changes in 76 modules having a low detection score

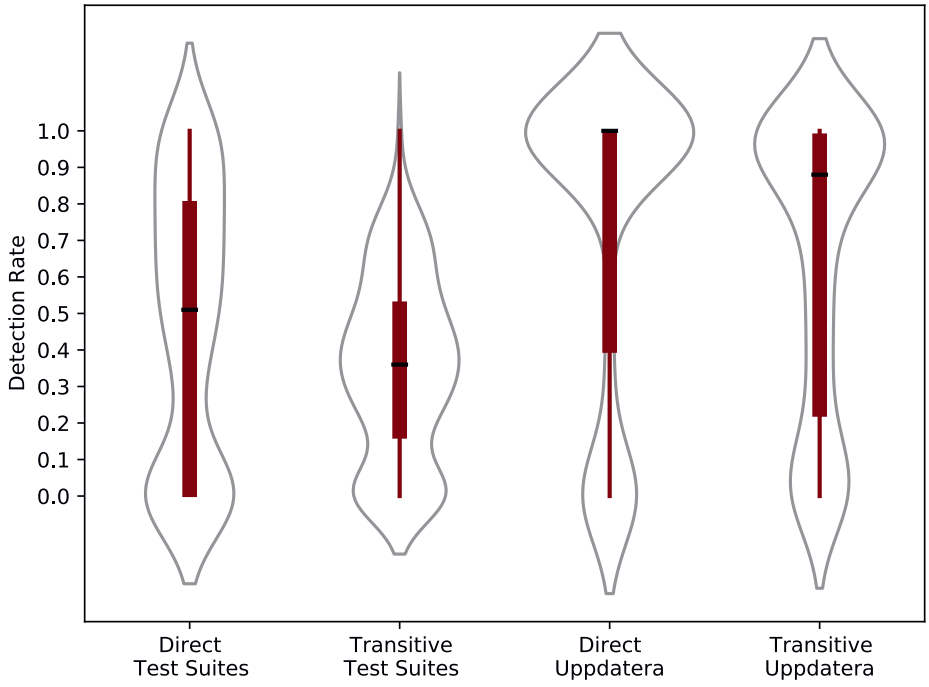


Figure 4.4: Mutation detection score

of less than or equal to 39% and 22% for direct and transitive dependencies, respectively. We perform a manual investigation using the following protocol: (1) back-track from the dynamic call trace to test suite, (2) identity potential tests cases that invoke the path in the call trace, and (3) investigate both the test case setup and source code in-depth to understand how Uppdatera could miss the regression change in the update.

In total, we identified four potential reasons for Uppdatera to miss faulty updates: 29 cases involving code generation, 26 cases involving class loading, 19 cases involving instrumentation, and 2 cases of instantiations of generic methods. Dynamic class loading along with code generation makes use of Java's Reflection API such as `Class.forName("DynClass");`. A majority of the inspected cases stem from libraries such a FasterXML Jackson-databind, Jersey REST framework, Spring framework, JAI ImageIO, Hibernate Validator, and Google Guice. Reflection is useful in cases such as the creation of data bindings (jackson-databind), data validation (hibernate or guice) or generation of HTTP endpoints from annotated user methods (jersey or spring framework). Resolving cases involving reflection is a known limitation of static analysis [124].

Although we do not instrument JUnit and Maven (which we use to power our setup), projects can bypass our exclusion filter by putting those libraries under a different namespace, a practice known as *class shading*. We identify several instances of bypassing the filter, an effect we cannot easily control. Finally, in two cases, generic methods defined

Table 4.3: Results of running UPPDATERA on 22 Dependabot pull requests

Pull Request	Update Type	Class	Confirm	Test Suite	UPPDATERA	Test Runtime	UPPDATERA Runtime
spotify/dbeam#189	Patch	**S**	✓	FP	FP	3.11	2.31
airsonic/airsonic#1622	Minor	**S**	✓	TP	FP	77	7.5
bitrich-info/xchange-stream#570	Patch	**S**	✓	TP	FP	2.78	1.93
CROSSINGTUD/CryptoAnalysis#245	Major	**S**	✓	TP	FP	12	2.6
dbmdz/imageio-jnr#84	Patch	**S**	✓	TP	FP	-	0.7
dnsimple/dnsimple-java#23	Minor	**S**	✓	TP	TP	2	11
smallrye/smallrye-config#289	Patch	**S**	✓	TP	TP	1.1	0.6
dropwizard/metrics#1567	Patch	**S**	✓	TP	TP	2.6	8.73
s4u/pgpverify-maven-plugin#96	Minor	**S**	✓	TP	TP	4	1.2
JanusGraph/janusgraph#2094	Minor	**N**	✓	FP	TN	365	33
UniversalMediaServer/UniversalMediaServer#1989	Major	**U**	✓	FN	TP	11	8.3
premium-minds/pm-wicket-utils#71	Patch	**N**	✓	FP	TN	1.56	0.51
UniversalMediaServer/UniversalMediaServer#1987	Minor	**N**	✓	FP	TN	11	7.7
CSUC/wos-times-cited-service#36	Patch	**S**	✗	TP	FP	0.55	0.5
Grundlefleck/ASM-NonClassloadingExtensions#25	Major	**S**	✗	TP	FP	4	0.5
RohanNagar/lightning#211	Major	**U**	✗	TP	TP	2	7.8
zalando/riptide#932	Minor	**U**	✗	FN	TP	7.5	20.5
pinterest/secor#1273	Patch	**S**	✗	TP	TP	390	13.5
michael-simons/neo4j-migrations#60	Patch	**S**	✗	TP	TP	3.45	0.8
zaproxy/crawljax#115	Minor	**U**	✗	FN	TP	17.35	1.3
hub4j/github-api#793	Minor	**S**	✗	TP	TP	1.3	4
zalando/logbook#750	Patch	**S**	✗	TP	TP	6.1	18.38

in user projects were only instantiated in tests but not in the project source code. Generally, call graph generators do not resolve generic methods unless there is a concrete instantiation of it.

Findings from RQ2: *Project tests are effective in a limited number of cases but not at large. UPPDATERA can detect twice as many faulty artificial updates as opposed to project test suites. Libraries making use of Java's reflection API could affect its applicability.*

4.4.3 RQ3: Change Impact Analysis in Practice

We conducted our online monitoring for two weeks between 13-27 Apr 2020 evaluating in total 22 Dependabot pull requests. On average, we harvested around 350 pull requests per day between Mondays and Wednesdays, 150 pull requests per day between Thursday and Fridays, 50 pull requests per day on the weekends. While the number of pull requests may seem high, a majority of them were updates of MAVEN plugins or test dependencies, uncompilable, or superseding previous pull requests. Thus, we posted on average two pull requests per day taking anywhere between one to four hours to manually evaluate pull requests and post our findings as comments.

Table 4.3 presents the analyzed pull requests along with the update type, ground truth class (i.e., **Class** column), external confirmation (i.e., **Confirm** column), results from the tooling, and execution times (in minutes). In total, our ground truth consists of 15 pull requests where the update is safe (i.e., **S** class), three pull requests where the dependency under update is unused and only declared (i.e., **N** class), and four pull requests where the updates that are unsafe (i.e., **U** class). The test suites of the analyzed pull requests classified 15 update as true positives(TP), four update cases as false positives (FP), and three update cases as false negatives (FN). UPPDATERA classified 12 update cases as true posi-

tives (TP), seven cases as false positives (FP), three cases as true negatives (TN). There are 12 cases where the two techniques report differently as highlighted by the colors in Table 4.3. Most notable are false positives; Uppdatera incorrectly reports six updates (highlighted yellow in the table) as unsafe that test suites can detect as safe. In those cases, the heuristics failed to account for refactorings or falsely derived call paths due to dynamic dispatch. In four cases, Uppdatera could not detect that the changes were refactorings (i.e., semantic-preserving changes). One such example is a confirmed minor update of the Apache `commons-lang3` library refactoring array length and null checks into a new function. In the two remaining cases, all reachable call paths were over-approximations. The update of `org.eclipse.emf.common` in one project included changes to List structures implementing methods of Java's List Interface (such as `addAll()`), resulting in unrelated interface calls being linked to it. This is a limitation of the CHA algorithm as it links interface calls to all available implementations. In three confirmed **N**-cases (highlighted blue in the table) where tests would falsely pass the updates, Uppdatera correctly identified no use of the dependency under update in the projects. The project maintainers in two of the reported cases have started refactoring work to remove those identified dependencies.

Uppdatera was able to complement test suites in three false-negative cases (highlighted red in the table). In our confirmed case of an unsafe update, Uppdatera identified the Apache `commons-lang3` library to break the application logic of a project due to changes in calculating string edits using the Jaro–Winkler distance. Generally, we can observe that solely using static analysis may risk falsely classifying safe updates as unsafe. Finally, we also make a comparison of execution times between running tests and Uppdatera. The results reveals that Uppdatera has faster or comparable times in 16 out of 22 cases, suggesting that change impact analysis can be a viable option to complement tests in CI environments.

Findings from RQ3: *Semantically equivalent changes (refactorings) and over-approximated function calls are the main sources of false positives in Uppdatera. However, Uppdatera helped project maintainers identify risky updates and unused dependencies.*

4.5 Discussion

4.5.1 Evaluating Library Updates

Updating to a new version of a third-party library is not a trivial task, and for good reasons: interface refactorings induce additional maintenance burden and integrating untested behavior can jeopardize project stability. Services such as `Dependabot` advocate a modest update strategy focusing on project compatibility: only update if the tests pass with the new library version. Effectively, developer-written tests act as the first-line defense against library updates introducing regression changes.

A key insight in our work is that automated dependency updates are not reliable. Our results strongly suggest that existing developer-written tests lack specifications that exercise dependencies in depth. This finding is in line with the work by Mirhosseini *et. al.,* [84], where developers report being suspicious of integrating automated updates due to fear of breakage. When selecting to adopt a third-party library, Bogart *et. al.,*, report that developers look at aspects such as reputation, code quality standards and active

maintenance to build up trust [16]. Perceived high-quality libraries can eliminate the need for extensive testing. In our case, we found evidence against this practice. The minor backward-compatible update of `org.apache.commons:commons-lang3`, a high-quality library, had changes that would break the application logic in one project if the pull request was merged in our manual analysis. In addition, the practice of testing third-party libraries is not common among popular testing books [107, 128, 129], very few research papers suggest testing of third-party libraries [130, 131]. When unit testing involves mocking, it often results in third-party libraries not being thoroughly exercised.

Directing testing efforts to dependencies would be a potential solution to the problem. Therefore, we recommend practitioners to use automated updating services cautiously and complement with tests for critical library dependencies. For tool creators in the domain, we argue for increased transparency in automated updating. With a small minority of projects having both coverage and tests capable of detecting simple regressions, pull requests could feature a confidence score on how well it is able to test new changes in a library under update. As a first step, tool creators can make use of our study setup to measure both coverage and quality of tests as an indication of confidence. A confidence score could also help reduce false negatives: if no tests are exercising a changed functionality of a dependency under update, `Dependabot` could avoid recommending it.

4.5.2 Strengths and Weaknesses of Static Analysis

Without needing to maintain additional dependency-specific tests, static analysis can be effective in deterring updates with potential regression changes. For a large number of projects with limited test quality, change impact analysis can fill the gap where tests are unable to reach and would be a compelling option for tool creators to consider. For a minority of projects, however, we identify certain third-party libraries that impede the overall analysis accuracy. Libraries heavily relying on code generation such as the Spring framework makes use of the Java Reflection API that are known to be statically difficult to analyze [132], could miss critical execution paths in projects that make use of them. Moreover, by linking interface calls to all its implementations, call graphs contain over-approximated call paths. We could observe non-existing interface calls from functions in the unused dependency to classes implementing the interface in the project during the manual analysis. As Ponta *et. al.,* [36] approach base on building a call graph with the project and its dependencies together, we make preliminary observations that projects having library dependencies with several common interfaces between them are likely to have many unrelated function calls. Exploring improvements such as using type hints with data flow analysis could potentially eliminate such function calls. Overall, we argue that static analysis is a useful complement in use cases where tests lack coverage. By also revealing and presenting gaps and quality issues in test suites, static analysis can help developers in prioritizing testing efforts of dependencies.

4.5.3 Threats to Validity

Sampling random projects from GitHub pose threats to our results: tests or dependencies in projects may not exercise production classes. To mitigate this risk, we configure our call extractor to only record call paths originating from the project source code. Call paths that do not traverse via project source code are excluded (e.g., test class directly calling a

dependency).

The use of mutation analysis to emulate source code changes in dependency functions has several potential threats to validity. First, we acknowledge that the applied mutation operators do not substitute actual regression changes in library updates. Our objective is to exercise all uses of libraries in a project by injecting simple faults to uncover potential coverage gaps in updating tools. Using real-world cases for this purpose would be challenging and potentially adding hidden uncontrolled factors. Second, our ground truth in **RQ2** represents reachable call paths inferred from running project tests, making it a subset of all possible executions and is a limitation of the benchmark. A potential avenue to explore is the use of test generation techniques such as EvoSuite [133] to discover new call paths. However, EvoSuite generates tests at the class level without considering its interaction with other classes or dependencies, generating artificial tests that may not represent valid use cases.

The false-positive rate in **RQ3** is indicative and not representative. Without domain knowledge of the interplay between a project and a dependency, the code reviews may state incorrect or incomplete information. To mitigate this risk, we post our code review assessment in the update for the project maintainer to react in case of incorrect analysis. Finally, for the reproducibility of our study, we have made the source code,[1] the experimental pipeline,[2] and our data publicly available [24]. Specifically, we include the examined projects, applied mutation changes, and their dynamic and static call graph.

4.6 Related Work

Updating library dependencies in projects To assist developers with updating dependencies in projects, researchers have studied practices around updating dependencies [6, 11, 16, 81, 84, 105] and proposed tools leveraging both static- and dynamic analysis [34, 91, 134]. Kula *et al.* [6] empirical study of 2,700 library dependencies in 4,600 Java project found that 81.5% remain outdated, even with security problems. The study found that factors such as uncertainties around estimating refactoring efforts and other task priorities as reasons for developers to not update dependencies. To address the update fatigue for developers, automated dependency updaters such as `Dependabot` and `greenkeeper.io` actively reminds and suggests dependency updates to developers through the use of pull requests. A study by Mirhosseini *et al.* [84] found that pull requests encourage developers to update dependencies more frequently but the frequency of updates and lack of convincing arguments defer them from updating. On similar lines, the work of Bogart *et al.* [16] also suggests that developers perceive the use of monitoring tools to have a high signal-to-noise ratio than giving actionable insights. Finally, the empirical work of Dietrich *et al.* [105] suggests that 75% of emulated library updates in the Qualitas dataset has breaking changes. However, only a few updates resulted in an error, motivating the need for contextual analysis.

Recently researchers have started to explore the use of static- and dynamic analysis to identify library updates with breaking changes, saving developers time, and review efforts of library updates. `NoRegrets` [34, 134] is a tool that detects breaking changes in test suites

[1] `https://github.com/jhejderup/uppdatera`
[2] `https://github.com/jhejderup/uppdatera-pipeline`

of dependent NPM packages before releasing an update of the library. Although helpful in minimizing the chances of breaking changes for clients, the identified subset of clients may not be representative of other clients. Similarly, Foo *et al.* [91] describes a static approach using simple diffing and querying Veracode's SGL [135] graph to find clients affected by breaking changes. In contrast to this approach, UPPDATERA analyzes at the project level (e.g., does not search for affected clients), targets diff with data- and control flow changes (i.e., not only interface changes), and includes a benchmark to compare updating tools.

Change Impact Analysis Change Impact analysis is a widely studied problem in program analysis research [136, 137]. Propagation of changes in package repositories have become an important research area in light of incidents such as the *left-pad incident*, and recent moves to emulate these problems on package-based networks [1, 5]. Several techniques [138–142] use call graphs as an intermediate representation for change impact analysis. Alternative techniques to call graphs are static and dynamic slicing [109, 143], profiling [144, 145] and execution traces [146]. Due to cost-precision trade-offs, several proposed approaches use a combination of these techniques. One such example is Alimadadi *et al.*'s work on Tochal, that leverages both runtime data and call graphs to more accurately represent changes to dynamic features such as the DOM. For a comprehensive overview of impact analysis techniques and change estimations, we refer the reader to Li *et al*'s [136] survey on code-based change impact analysis techniques

An application of change impact analysis is regression test selection techniques [147] (RTS) such as class-based STARTS [148, 149] and probabilistic test selection [150] that find relevant tests for evaluating new code changes. We found in our evaluation that test suites have limited coverage of dependencies, thus RTS may not be able to find tests relevant for changes in dependencies or have enough test data to build a prediction model for average GITHUB projects. Finally, Danglot *et al.* [151] and Da Silva *et al.* [152] investigate the use of search-based methods such as test amplification and automated test generation for detecting semantically conflicting changes. Although search-based methods are effective in reducing false positives and to some degree eliminating false negatives present in static analysis, they are limiting for integration test scenarios such as automated dependency updating. Da Silva *et al.* [152] found that automated test generation such as EvoSuite [133] have difficulties in generating effective tests for complex objects with internal or external dependencies.

4.7 Conclusions and future work

In this chapter, we delve into the empirical investigation of test suite reliability in the context of automating dependency updates. As developers increasingly rely on services that automate dependency updates, it is critical to determine how much project tests cover utilized functionality in library dependencies, their effectiveness in detecting simple regressions, and their performance in real-world scenarios. Furthermore, given the call for more conservative techniques in recent research, we also explore change impact analysis's role in minimizing false negatives.

Our analysis reveals that half of the 521 well-tested projects we investigated cover less than 60% of their function calls to direct dependencies with their tests. When looking at

call paths to transitive dependencies, coverage dwindles to a mere 20%. Furthermore, by simulating simple faults in library dependencies across 262 projects, we found that only one-fourth of the projects can detect 80% or more faults in functions of direct libraries. For transitive dependencies, this number plummets to one-eighth.

However, change impact analysis can detect 80% of potentially breaking changes in both direct and transitive dependencies, showing double the efficacy of using test suites. While change impact analysis shows promise in flagging faulty updates, our manual investigation of its potential to complement tests in 22 `Dependabot` pull requests reveals both advantages and drawbacks. The analysis could prevent unsafe updates in three cases where tests were unsuccessful and detected unused libraries in two instances. Nonetheless, the approach had a higher false-positive rate due to its relative imprecision than tests.

These findings underscore the need for developers who utilize automated dependency updating to be mindful of the risks of relying on project tests for compatibility checking. Updates can incrementally introduce unintended functionality without sufficient coverage or adequate tests for all usages of library dependencies. Given that services like `Dependabot` do not explicitly communicate the risks associated with updating dependencies, we recommend that tool creators incorporate reliability measurements like scoring test suites in pull requests. Moreover, given our exploration of change impact analysis, we advocate for tool creators to consider combining dynamic and static analysis to establish verification techniques independent of users' test suites.

We recommend that researchers and practitioners establish best practices for updating third-party libraries. An initial step in this direction is to understand how developers direct testing efforts toward dependencies and their strategies around them. Furthermore, tool makers should investigate hybrid workflows through data-driven methods for efficient update checking by combining dynamic and static analysis.

5

Evaluating the Impact of Third-Party Library Reuse in Java Projects: An Empirical Study

The use of open-source libraries is common in the software industry; however, reliance on external code introduces operational and security risks, some even present with large shares of unused imported libraries, such as deserialization gadgets, malicious contributions, and supply chain attacks. To minimize risks and address regulations, organizations implement rigorous policies, like producing software bills of materials, meeting test coverage targets, or defining quality criteria for open source. To better understand how projects utilize these libraries, we conduct an empirical study involving 3,182 releases from 176 Java projects in the Census-II dataset, representing widely used libraries in production applications. Our results show that, in these projects, 87% of lines of code stem from third-party libraries, but they do not use all of the imported code. Our reachability analysis uncovers that a range between 12-38% of external lines of code is reused within these libraries. Over time, these numbers steadily increase, signifying tighter reuse. However, we find that the reuse largely remains stable between consecutive versions. The high degree of unused third-party code shed light on the operational and maintainability risks associated with third-party code imports and emphasize the necessity for a more fine-grained evaluation of dependency usage in software development.

In typical software development projects, developers strive to avoid rewriting code that has already been written in the past. Encouraging modularity and promoting the sharing of functionality is the norm, especially since the advent of the open-source software (OSS) movement. The concept of code reuse refers to employing pre-built components, such as reusable collections of functions or data types, in both source or binary form, that are easily importable in projects [153, 154]. The practice of reusing open-source components can be studied in various levels of granularity, starting from the application level (for example, using a ready-made webserver), to the library level (for instance, using a released open-source library), down to the more detailed aspect of fine-grained code-based reuse.

Importing and using third-party libraries incurs operational and maintenance challenges. Library developers often build third-party libraries on top of existing third-party libraries, effectively creating a network of dependencies between libraries [1, 8, 155]. For example, importing a single `npm` library in a JavaScript project can pull in an average of 80 other `npm` libraries, [7] each with its own set of release cycles, development standards, and testing practices [16]. These differences among imported libraries complicate developers' efforts to address vulnerabilities effectively [7, 30] and detect semantically breaking changes when updating libraries [156, 157].

In response to these challenges, organizations, including government agencies, are increasingly mandating more responsible reuse of third-party libraries [158, 159]. Tools such as Software Bill of Materials (SBOM), dependency vulnerability scanners and license compliance checkers are increasingly being used, by OSS third-party library users to safeguard their products. On the development front, practices such as higher test coverage[160], mapping potential attack surfaces, code security analysis and reproducible builds from source to binary[161], are being promoted to ensure that the produced components are of high quality.

A common characteristic of most of the approaches described above is that they work at the library or component level. However, reuse does not happen at the library level, but at the code level: functions in the client project interact with APIs in the imported library and, transitively, in its dependencies. Thus, a disconnect arises between what tools (and developers) perceive as reuse and what the actual code reuse really is. Gaining insight into the degree of reuse at the code level is crucial for decision making and risk management, as it can help inform policies such as attack surface monitoring, insourcing of components with very low reuse ratio or updating to newer versions in response to vulnerability disclosures.

Toward this end, we conduct a study on library reuse at the code level. Our high-level goal is to establish *code reuse baselines*: learning code reuse ratios from high-quality OSS components used in enterprise applications to inform decision-making. To guide our investigation, we formulate a set of research questions, presented below:

RQ1 How much third-party code do projects import?

RQ2 How much of the imported third-party code do projects reuse?

To study how projects reuse code in third-party libraries, we create a dataset by starting from Census-II [162] and iteratively removing cases that might skew our analysis. The dataset comprises libraries frequently used in production applications. To study RQ1, we analyze the dependency tree of each release in terms of lines of imported code. Our analysis is longitudinal: we study averages in six-month intervals and changes between subsequent major and minor versions of those libraries. To study RQ2, we construct call graphs for each analyzed library version to quantify the degree of reuse of third-party libraries We compute upper and lower bounds for our results, to cope with imprecision of our program analysis technique.

Our main finding is that while library-based reuse computations indicate that, on average 87% of the code in an application is being imported, the actual reuse is in the range of 12% to 38%. Moreover, those numbers remains largely at the same level in the last ten years, effectively constituting code reuse *invariants*.

Our work makes the following contributions:

- A method for identifying library reuse using program analysis, including thorough analysis of its limitations.
- A dataset on code reuse for client applications and libraries written in the Java programming languages.
- Baselines that can be used to evaluate code reuse in real projects.

5.1 Background

5.1.1 Software Reuse

The concept of software reuse dates back to the early days of software engineering [163] and generally signifies the reuse of design patterns, source code, or packaged components such as libraries or frameworks to increase developer productivity and software quality at reduced efforts and costs.

The open-source movement contributed significantly to producing reusable software components in the last two decades. Package managers have greatly facilitated their consumption, which exists for most programming languages, and automate the identification, resolution, and download of third-party components (called dependencies) based on declarative manifest files. For example, in the case of Maven, the most prominent package manager for Java, developers specify (direct) dependencies in `pom.xml` files. Maven takes care of locating those dependencies in package repositories, recursively resolving their dependencies (indirect ones for the development project at hand) and any potential version conflicts, and downloading a consolidated set of Java archives (JARs) to the local development environment.

Although software reuse significantly accelerates the development process, it also introduces risks that potentially manifest during later phases of the software lifecycle. An example of such a risk is the discovery and fixing of functional bugs in an imported third-party library that is no longer maintained by its community; hence, the need to receive updates can result in significant development costs down the line.

5.1.2 Program Analysis

Program analysis refers to the set of techniques that are used to prove whether programs satisfy specific properties. Static analyses perform analysis without executing the code, whereas dynamic analysis techniques focus on the runtime behavior of programs. A form of static analysis we use in this work is the representation of a program using *call graphs*, graphs whose nodes represent program functions and edges represent calling relationships [48]. Being a static analysis technique, call graphs offer *approximations* of the program's runtime behavior. Specifically, they may be missing some function calls (*e.g.*,due to the use of dynamic code loading) or they may include function calls that may not happen in practice (*e.g.*,due to runtime configuration restricting parts of specific functionality).

Generating precise call graphs is computationally expensive. In realistic program analysis scenarios, an analyzed client program under development uses code from multiple third-party libraries. As the client code changes much faster than its library dependencies, the analysis might be sped up if each dependency is only analyzed once, the analysis

results of third-party libraries are cached and composed on request with the client program. In our work, we use the idea of callgraph stitching, first proposed by Keshani [164], to scale our analysis to thousands of call graphs.

5.1.3 Related Work

Existing works use and combine various program analysis techniques in the context of Java/Maven dependency management in order to study software reuse and address the above-mentioned risks.

Reuse metrics. The use of call graphs to establish reuse metrics is mentioned by Norman[165], based on which Bieman[154] defines various properties and metrics for software reuse in object-oriented programming languages. Using his terminology, we study *direct and indirect verbatim public reuse*, *i.e.,* the direct or indirect consumption of *servers* (software entities of a library) produced externally (public), and which have not been modified (verbatim). Some metrics proposed by Bieman[154] and Devanbu *et. al.,* [153], however, investigate the frequency of reuse, *e.g.,* the number of paths to indirect servers. At the same time, we are solely interested in whether a library component is called (at least once) or not.

API use. Many studies investigate API use and its evolution across releases of both libraries and their clients, *e.g.,* to carve out libraries' core API [166, 167], to discover the introduction of breaking changes [11], or to understand clients' update lag [87, 168] or their use of deprecated APIs [169]. Compared to these works, we look at a client's direct use of a library's public API, typically declared as a direct dependency, and investigate indirect code reuse in transitive dependencies.

Software Bloat. Automated dependency management results lead to software bloat, *i.e.,*, the accumulation of functionality and code not used by given downstream projects. Soto-Valero *et. al.,* [170] studies 9,639 artifacts hosted on Maven and finds that 75.1% of their dependencies are bloated, *i.e.,*, none of their types are used directly or indirectly by the classes of the artifact. Moreover, in another study[171] the same year, the number of bloated dependencies grows over time, particularly for transitive dependencies. Both works focus on establishing the number of unused classes found in third-party libraries. In contrast, we aim to understand the amount of reuse made by the projects at the granularity of LOCs and functions.

Dependency networks. Package repositories like PyPI for Python or Maven host hundreds of thousands of packages with millions of versions. Empirical studies like [1, 27, 30, 172] describe properties of such package dependency networks, *e.g.,*, the number of direct or transitive package dependencies, study their evolution and compare them across ecosystems. Hejderup *et. al.,* [155] performs a large-scale analysis of the Rust ecosystem to compute call-based dependency networks to study trends and insights of the ecosystem on a function level, thereby acknowledging that networks built on manifest data only represent a coarse-granular view on reuse. In contrast to those works, we focus on a small subset of libraries that are reportedly the most-used Java open-source libraries in *production applications at thousands of companies* [162], thereby ignoring a long tail of packages of little relevance to the broader community.

Security. Plate *et. al.,* [173] uses runtime traces to assess whether known-vulnerable methods in project dependencies are reachable in the context of given projects. Ponta *et.*

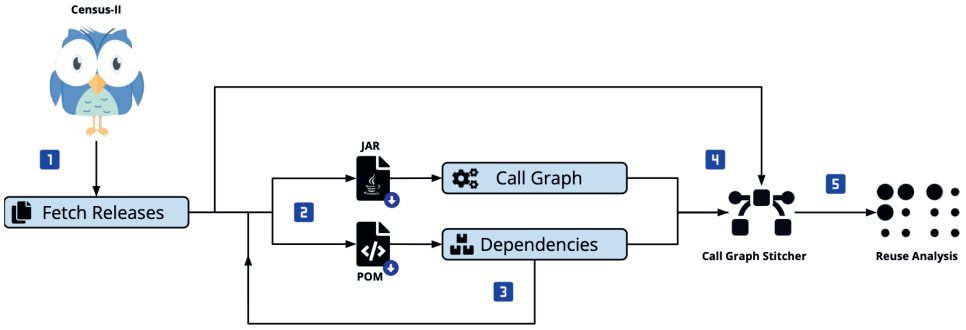


Figure 5.1: Study Infrastructure Overview

al., [174] extends this approach and determines vulnerable methods' reachability through static and dynamic techniques. They demonstrate the complementarity of those techniques, as 11.7% of reachable vulnerable methods are only discovered through their combination. Those works, however, do not investigate code reuse in general terms, independent of security vulnerabilities. Ohm *et. al.,* [175] provides an overview of various attack vectors aiming to infect open-source projects, which motivates the internalization and removal of dependencies for risk reduction.

5.2 Research Method

We follow the research approach outlined in Figure 5.1 to examine how popular Java projects reuse third-party libraries. Initially, we select Java-based projects from the Linux Foundation Census-II dataset [162], which contains libraries used in production applications. For each package, we retrieve information about all releases using the Maven Search API[1] 1. Then, we download the associated POM (dependency descriptor) and JAR (compiled code) files for each release 2. Using Maven, we build each POM file's dependency tree and extract package identifiers of resolved third-party libraries. We recursively repeat step 2 for each third-party library until no new library versions can be obtained.

The previous steps generate a set of library versions to perform analysis on, at the package level. However, answering all our RQs requires precise identification of the code that is reused. For this reason, we build call graphs for each individual release 3 and then *stitch* those callgraphs together to compose a global callgraph for each release 4. On top of those global callgraphs, we perform reachability analysis to track code reuse by third-party libraries 5.

In the following sections, we describe each analysis step in detail.

5.2.1 Analyzing Code Reuse in Third Party Libraries

Our unit of analysis is a library released with a version in an open source package repository. We consider that a client application (or another library) reuses the library if there is a function call from the client into the library; this omits reuse cases where a client

[1] `https://search.maven.org/solrsearch`

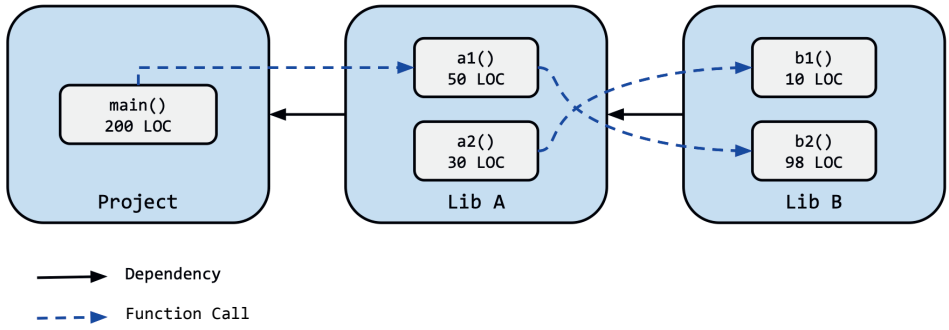


Figure 5.2: Example: Project depending on `Lib A` and `Lib B`

only uses a data type exposed by the library. In such cases, reuse can be both implicit and explicit, as projects can have direct and transitive library dependencies. The term explicit reuse refers to function calls to first level library dependencies (typically declared in dependency descriptor files); while implicit reuse refers to function invocations in transitive dependencies. Implicit reuse arises when a client calls a function in a direct dependency which then calls a function in a transitive one. In Figure 5.2, we find that the `project` explicitly uses functionality from the `Lib A` by calling the `a1()` and implicitly uses `Lib B` by calling the `b1()` via the `a1()`.

In **RQ1** and **RQ2**, we study reuse both from a static and an evolutionary perspective. Initially, we analyze the distribution of imported code across all releases in our dataset. To study evolution, we consider two cases: i) evolution of reuse over time in six-month intervals, and ii) evolution of reuse across library versions. The time-based analysis exposes aggregate reuse trends at the ecosystem level, while the version-based analysis quantifies how library import patterns change across consecutive major and minor versions.

RQ1: Imported Code Recent research reveals that software projects increasingly import third-party libraries, often through transitive dependencies [27, 155]. However, the impact of this trend on the proportion of insourced and outsourced source code available in projects is less known. To address **RQ1**, we estimate the size of client projects and third-party libraries by counting the percentage of i) lines of code (LOC) and ii) the number of functions of imported third-party code over the total size of the project. Using the example in Figure 5.2, the LOC import ratio is 188 / 388 = 0.48, suggesting that about half the code in `Project` comes from external libraries. On the other hand, the function import ratio is 4 / 5 = 0.8, highlighting to a higher degree that reusable code stem from external libraries.

RQ2: Reused Code Importing a third-party library does not imply full reuse of the imported code. To answer RQ2, we construct a call graph (see Section 5.2.3 for construction details) of a project and its imported third-party libraries to approximate reuse. Using a call graph, we perform reachability analysis: if functions in third-party libraries are reachable from functions in the analyzed project, this indicates reuse. We count the number of both reused functions and their size.

Assuming we have a call graph of the project and its dependencies, we calculate the

Table 5.1: Summary of packages and releases at each stage of the dataset processing.

Stage	Packages	Releases
Initial dataset (Maven-based projects)	334	27,210
Semver schema validation	307	6,113
Dependency tree pruning (invalid dependencies)	302	5,661
Selecting packages with dependencies	213	3,997
Call graph generation and LOC annotation	194	3,765
Final dataset (removing incomplete LOC info)	176	3,182

reused LOC ratio (**RLR**) as the sum of the LOC of all reachable dependency functions divided by the sum of all LOC functions of all third-party libraries as the following:

$$\text{RLR} = \frac{\text{reachable\_LOC(Project)}}{\sum_{i=1}^{N} \text{size}(\text{Lib}_i)}$$

The function `reachable_LOC` computes the sum of LOC of all reachable external functions. Using the example in Figure 5.2 we see that `Project`, explicitly calls 50 LOC in `Lib A` and implicitly reuses 98 LOC in `Lib B`. Therefore, the reused LOC ratio is 148 / 188 = 0.78, indicating that a large proportion of imported LOC is actively being reused by the project. Similarly, we compute the ratio of reused functions (**RFR**) by substituting LOC with the number of functions. Thus, for the same example, the function reuse ratio of `Project` is 3 / 5 = 0.6. The ratio is comparable to the **RLR**.

5.2.2 Dataset Construction

We base our study on Linux Foundations' Census-II dataset, which contains the most-used third-party libraries in production applications according to comprehensive data provided by several Software Composition Analysis (SCA) tools such as Snyk and FOSSA [162]. From the 348 Maven-based projects in the dataset, we can retrieve 27,789 releases from the Maven Search API. We exclude three invalid group ids and artifact ids unavailable from Maven Central and `org.scala-lang:scala-compiler` as it is a non-Java project. After removing all non-JAR releases, our initial dataset is 27,210 releases belonging to 334 projects.

Table 5.1 describes the results of the various processing steps we used to create our dataset. Initially, and to answer the RQs, we need to find a way to order releases. Even though Maven does specify a version ordering scheme, Raemaekers *et. al.,* [172] found that this is not followed in practice. Semantic Versioning (semver) is more widely accepted, but there is no strict adherence. Nevertheless, we decided to follow semantic versioning as our version ordering scheme. After validating releases according to semver, we filtered out 27 projects (21k releases). The significant reduction of releases is due to packages such as `aws-java-sdk` and `aws-java-sdk-kms`, which release on Maven on every commit in their repository.

We then built the dependency tree for each release and removed releases with invalid dependencies on third-party libraries, such as non-existing names and constraints, result-

ing in 302 projects with 5661 releases. Inspecting the remaining dependency trees revealed that 119 packages and 952 releases do not have any dependencies. Moreover, some packages and releases only declare test and non-production dependencies; thus, we selected packages with compile, runtime, and provided dependencies on third-party libraries, bringing the number down to 213 packages with 3997 releases.

Overall, we identified 17,775 unique third-party library releases from the resolved dependency trees. We could not download a JAR for 11 third-party libraries; however, these were test dependencies, and thus we did not reduce the dataset further.

5.2.3 Call graph construction and stitching

We used the OPAL call graph generator for the call graph construction as it has been shown to handle most language features and call types [124]. We configure OPAL to perform Rapid Type Analysis (RTA) [176] analysis in library mode. Effectively, this means that OPAL treats all public functions in the analyzed library as entry points. The call graph is constructed by following all calls from the entry points to all internal functions; if an external (dependency) function is called, OPAL will create a placeholder. The call graph is saved in an intermediate format that records internal and external types, the called functions and call site information including dispatch type (static or virtual).

We built call graphs for 18,377 releases (all identified third-party libraries and the analyzed client programs). OPAL could not process 368 JAR files, while for 264 JAR files we could not extract line count information as this information could not be extracted from their JAR files. Thus, we have annotated LOC information for 17,745 JAR files, corresponding to 194 packages and 3765 releases. As some third-party libraries may need LOC information, we must validate the dependency tree of releases and remove releases with incomplete information. After removing releases with incomplete LOC information for their dependency tree, our final dataset comprises 176 packages with 3182 releases.

Each analyzed release is treated as a data point in our analysis: we first resolve all dependency versions and then we stitch the individual callgraphs into a global callgraph to perform reachability analysis on. The stitching process is an industrial-grade re-implementation of the process proposed by Keshani [164]. Effectively, stitching takes a set of intermediate callgraphs and resolves (creates edges to) all external function references to their concrete implementations in other callgraphs.

5.2.4 Treating Dynamic Dispatch & Reflection

A common issue when constructing (and stitching) callgraphs is the handling of virtual dispatch call sites: given a method call on an interface (*e.g.,* the Java standard interface `Iterator.next()`), what is the actual implementation that will be called at runtime? The most straightforward approach, called Call Hierarchy Analysis (CHA) [177], creates edges toward all concrete implementations of the function in the object hierarchy delimited by the type of the receiver object. However, this creates an overapproximation of the actual code execution; in the case of popular interfaces (*e.g.,*the aforementioned `Iterator`), this might lead to *edge explosion* situations where a method may be linked to 100s of functions (and their calls, transitively).[2] In turn, this affects any reachability analysis significantly.

[2]Such call sites are refered as *megamorphic* in the static analysis literature.

RTA, which both OPAL and the stitching code implement, attempts to limit the effect of this overapproximation by tracking object allocations along call paths; in practice, we noticed that their effect on precision is minimal when considering third-party libraries in the analysis scope.

To deal with the issue of overapproximation, we decided to take a pragmatic approach: treat the reachability results as an *upper bound* (*i.e.,*code reuse cannot exceed that calculated by the unmodified reachability analysis), but we also introduce a *lower bound.* The lower bound is calculated by constructing callgraphs where virtual dispatch call sites are not stitched if they resolve to more that n targets. We calculate n empirically by analyzing the distribution of target numbers and taking the median (five in our dataset). The resulting lower bound callgraph is, by definition, unsound, but it gives a more realistic view of the actual code reuse as it avoids megamorphic call sites by construction. A sensitivity analysis for various values of n (three, five, seven, ten) indicated that the results did not differ radically.

A further issue with static callgraph construction is using reflection to load and call code. Static analysis cannot effectively deal with reflection. However, various popular frameworks in Java (*e.g.,*Spring, Hibernate and Guice) make extensive use of reflection and versions of those frameworks exist in our analysis set, thus making our results imprecise. Both, Sui *et. al.,* [178] and Liu *et. al.,* [179] find that not all of Java's reflection APIs equally contribute as a source of unsoundness in analysis. Therefore, following the guidelines of Landman *et. al.,* [180], we aim to quantify which reflection APIs appear most frequently as part of **RQ2**.

5.3 Results

The results of our analysis deviate from a normal distribution according to the Shapiro-Wilk test ($p < 0.01 \leq \alpha$). Consequently, we employ the non-parametric Spearman correlation coefficient (ρ) for our correlation analysis. We follow Hopkins's guidelines for interpreting the correlation coefficient [82]: a value of $0 \leq |\rho| < 0.3$ indicates no correlation, $0.3 \leq |\rho| < 0.5$ suggests a weak correlation, $0.5 \leq |\rho| < 0.7$ signifies a moderate correlation, and $0.7 \leq |\rho| \leq 1$ implies a strong correlation. Our results including analysis scripts are available as a replication package [25].

5.3.1 RQ1: Import of Third-Party Libraries

Distribution Figure 5.3 illustrates the percentage of third-party code in the total code available for all project releases, considering both external LOC and functions. The medians are similar: 84% for external LOC (mean: 71%) and 88% for external functions (mean: 71%). Although the distributions of reused LOC and functions in the codebase are comparable, we observe a marginally higher proportion of external functions in projects than external LOC, particularly in the 90-100% range in the figure.

By dividing the data at the 50% threshold, we deduce that most releases (74% or 2359 out of 3182) contain more than 50% of external code. This underscores that a significant part of modern software heavily depends on external components. However, one-fourth of the releases consist of more internally developed code. Examples include `joda-time:joda-`

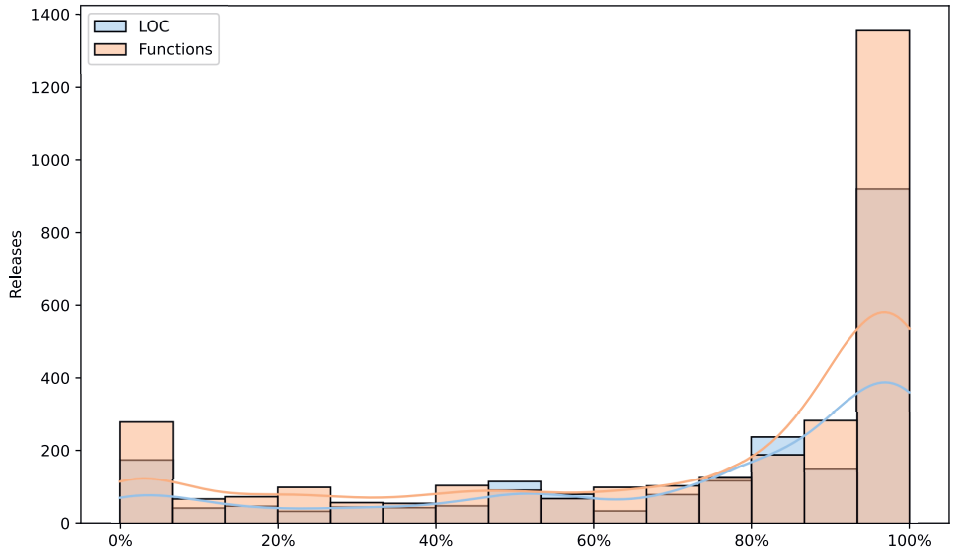


Figure 5.3: Import Ratio of all project releases

`time:2.9.9` (2%), `com.sun.xml.bind:jaxb-impl:2.2.3` (15%), and `org.ehcache:ehcache:3.6.1` (3.2%).

Evolution To understand the changes in external code import over time, we analyze the latest version of each package at six-month intervals from 2005-11 to 2023-04 for both the number of functions and LOC in Figure 5.4. Over a decade from 2012-11, the median has steadily increased from 81% to 87%, indicating that projects are progressively importing more external code. When comparing the median for the external code ratio between the number of functions and LOC over time, we notice that the gap between them narrows. From around 2013-05 onwards, the medians are comparable, indicating that median ratios approximate each other.

The variability and spread of the boxes show a consistent contraction and upward shift, signaling an increased reliance on external code. In recent years, we have observed that releases consist of at least 20% or more external code (excluding outliers). This increase could be attributed to the growing number of dependencies over time. Therefore, we investigate the correlation between the total number of dependencies and the percentage of third-party code over time, resulting in a moderate $\rho = 0.67$ correlation. This finding suggests that the rise in external code is likely due to an increased dependence on declaring or resolving more third-party libraries or third-party libraries expanding in size.

Version Diff When diffing the import of external code between consecutive minor and major releases of packages, we find 8.36% (266/3182) of the releases to have a change more significant than 1%, suggesting that there are generally no dramatic changes in the import of external code between consecutive releases. Figure 5.5 depicts the percentage

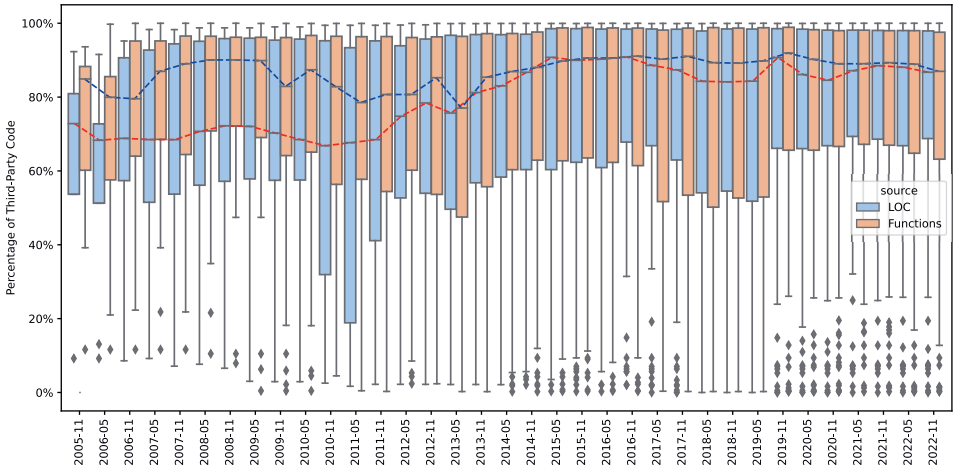


Figure 5.4: Evolution of Import Ratio between 2005-2023

5

diff for releases that have a delta greater than 1% of imported external LOC and number of functions. Overall, we identify a bell-shaped distribution where most delta changes range between -25% and +25% between consecutive versions. Furthermore, we also notice more releases with more significant diffs for functions (362 cases) compared to LOC (226 cases). Among the outliers, we find that `org.ehcache:ehcache` grew from version 3.7.0 to 3.8.0 with 52% as it started to depend on `org.glassfish.jaxb:jaxb-runtime`, and `org.mockito:mockito-core` reduced from version 1.3.0 to 1.5.0[3] with 72% by removing `cglib:cglib-nodep`. Overall, we find that between releases, there are no major changes in the inclusion of external code in projects. The likely reason for release changes with a high delta percentage is to include or remove a third-party library.

Findings from RQ1: *The import of third-party code accounts for approximately 87% of projects, driven likely by the growing number of external libraries imported in projects. Comparing versions, we observe minimal fluctuations in the inclusion or exclusion of third-party library code. Overall, projects predominantly consist of external code.*

5.3.2 RQ2: Reuse of Third-party Libraries

Use of Reflection APIs Third-party libraries often utilize reflection methods to enable dynamic behavior, such as creating objects or calling methods at runtime [180], which may impede our ability to find all instances of code reuse. Thus, to understand how the presence of Java's reflection API affects our analysis, we first identify the most frequently invoked reflection APIs to determine potential sources of unsoundness and then examine the number of library dependencies that use reflection in projects. Table 5.2 presents the most frequently invoked reflection APIs in our set of third-party libraries. Among

[3]there is no 1.4.0 for this particular project

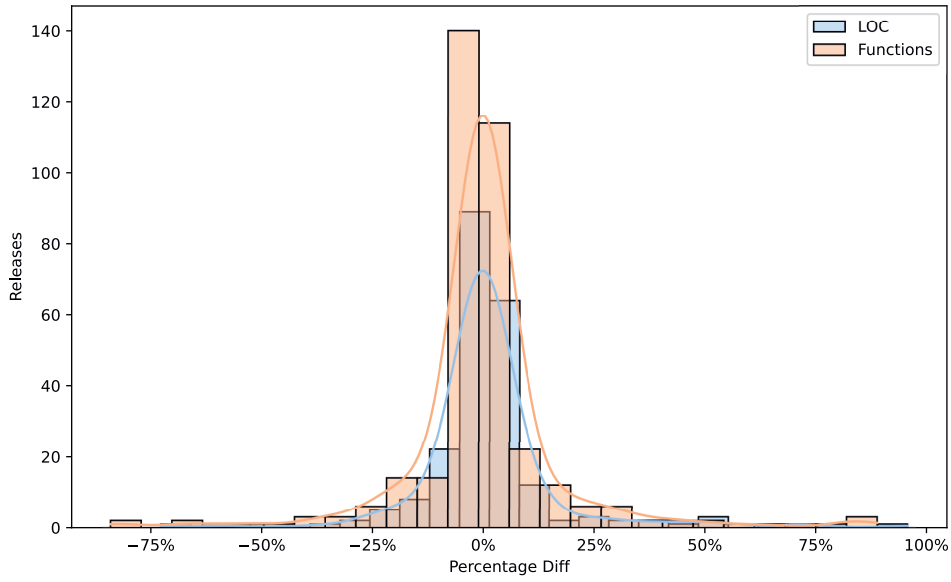


Figure 5.5: Percentage Diff between Major and Minor releases (excluding -1% to 1% changes)

Table 5.2: Most Frequent Reflection APIs

Reflection API	Count
`Method.invoke`	5729
`Class.getDeclaredMethods`	5634
`Class.forName(String)`	5093
`Class.getMethods`	5052
`Class.getMethod`	4878
`Constructor.newInstance`	4602
`Class.newInstance`	3741
`Class.getConstructors`	3394
`Class.getConstructor`	3284
`Class.getDeclaredConstructors`	3112
`Class.forName(String,Boolean,Classloader)`	2885
`Class.getDeclaredMethod`	2662
`Class.getDeclaredConstructor`	2178
`ServiceLoader.load`	1463

them, all except `Method.invoke`, `Constructor.newInstance`, and `Class.newInstance` are non-problematic, as they do not construct or invoke dynamically invoked methods necessary to identify reuse. Using only the three shortlisted APIs, we find in Figure 5.6 that most releases import a third party using these APIs. In fact, nearly half of all releases depend on only library dependencies invoking the reflection API.

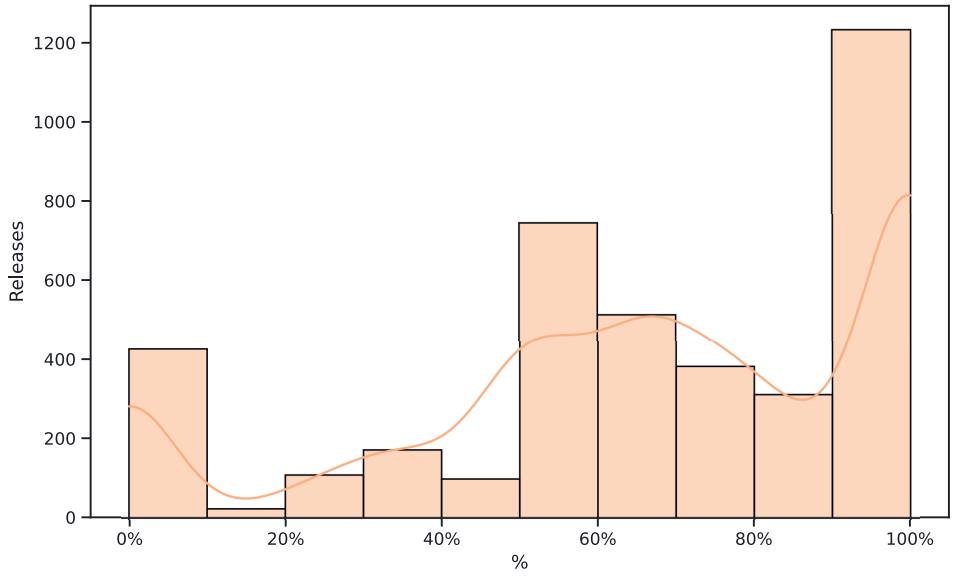


Figure 5.6: Effect on Reflection APIs

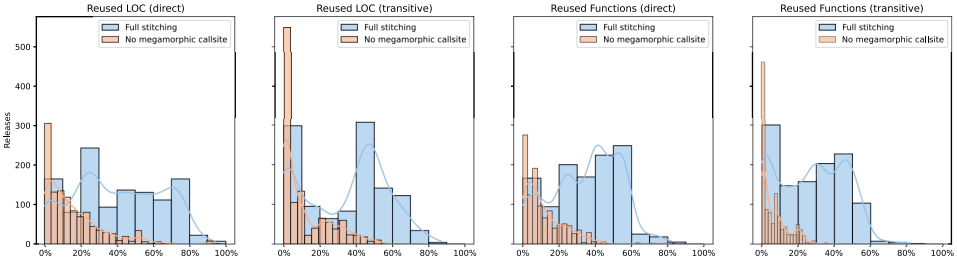


Figure 5.7: Distribution of Reused Third-Party Functions and LOC

However, despite the high presence of reflection APIs in third-party libraries, according to Sui *et. al.,* [178], dynamic invocations are a minor source of unsoundness for the three shortlisted APIs. By adding dynamic analysis, Liu *et. al.,* [179] show that the average increase of calls is 15 by recording calls of such APIs, which implies that the overall effect on our analysis is limited.

Distribution Figure 5.10 depicts the reuse of third-party functions and LOC per direct and transitive library dependencies. When comparing the plots between direct and transitive dependencies, we find a generally higher percentage of reuse of directly declared third-party libraries than transitively resolved ones. For example, looking at the histogram for the portion of reused functions of transitive library dependencies, we find that very few releases use 70-80% of their transitive dependencies. Soto-Valero *et. al.,* [170], and Hejderup *et. al.,* [155] also observe low utilization of transitive library dependencies by

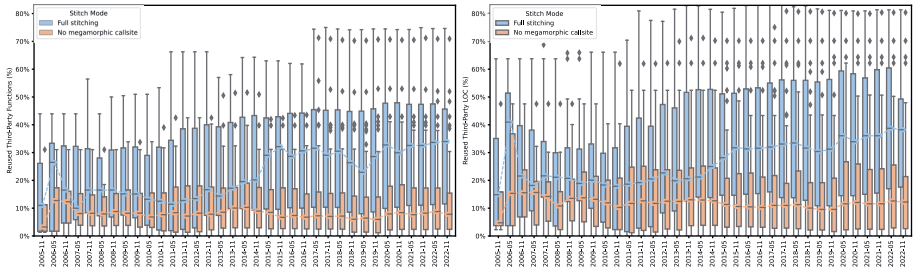


Figure 5.8: Evolution Reused Functions and LOC of Third-Party Libraries

finding most transitively resolved library dependencies are mainly unused.

When comparing the two stitch modes, we see a right-skewed distribution (no megamorphic call sites) and a largely multimodal distribution (full stitching). In other words, the full stitch mode suggests reuse in terms of invoked functions or LOC is more evenly spread, notably in the figure for direct reused LOC, whereas the no megamorphic call site mode suggests that reuse is overall very low (between 0-20%). The contrast in the distribution shape highlights the over-approximation of third-party function reuse due to including all targets in interface calls. Particularly in the range between 30% to 60% reuse of either direct or transitive dependencies, we find a substantial number of releases with this amount of reuse in the full stitch mode. In contrast, in the no megamorphic call sites, we find almost no or a limited amount of releases. One example is `com.typesafe.netty:netty-reactive-streams:2.0.6`, which invokes 47% of all third-party (transitive) functions in the full stitch mode and only 2.1% in the other case.

Evolution How projects reuse third-party libraries over time provides insights into patterns and changes in how developers and organizations use imported third-party library code. Between 2005 to 2022, we identified an overall growing trend of increased reuse over time, both in terms of function and LOC, in Figure 5.8. While the median remains similar over time for the no megamorphic calls, we can observe an increase in the variation over time. Notable are the whiskers and outliers in recent time points. For the full stitching mode, we can see a significant increase over time in the median and spread of the box plots; for instance, the median reuse from 2012-11 to 2022-11 grew from 22.7% to 38%, an increase of nearly 25%. We can also see the gap expanding when comparing the gap between the median of the two different stitch modes. Earlier time points are more comparable to later ones, particularly the median and lower quartile. For example, we find the median for the full stitching to be 20% and 13% for no megamorphic calls in 2009-11, and 38% and 12.2%, respectively, in the latest time point. Finally, when comparing the function reuse with LOC reuse, we notice that the results complement each other. The correlation between the two is strong per stitch mode, ρ = 0.93 for full stitch mode, and ρ = 0.76 for no megamorphic calls. Suggesting that either using LOC or functions approximate each other well when estimating the reuse in terms of size.

Version Diff Similar to Figure 5.5, we find that most releases do not have a more significant delta increase or decrease than ≤ 1%, suggesting that the reuse of third-party

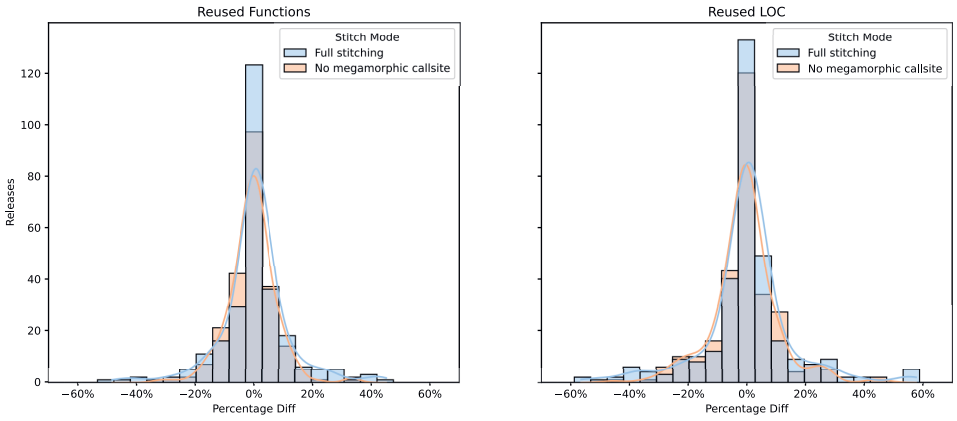


Figure 5.9: Percentage Diff Between Releases - Functions and LOC

libraries remains broadly the same. Both Sawant *et. al.,* [169] and Soto-Valero *et. al.,* [170] also observe that the use surface of third-party APIs in projects remains the same over time with no significant changes. When narrowing down to releases outside the -1% and 1% range in Figure 5.9, we find a similar set of 266 releases in the full stitching mode and a reduction to 233 in the other mode when reducing megamorphic call sites for reused functions. For reused LOC, we found 319 releases, with a decrease to 286 releases when removing megamorphic calls. We see a nearly matching bell-shaped distribution when comparing the percentage diff between the actual reuse of third-party functions and LOC. There is a slight deviation between the two; the reused LOC diff has releases with more than 50% delta, which is not present in the reused function diff. When comparing the stitch mode, we find that they largely follow the same distribution, unlike previous comparative plots. The correlation after normalizing the data is $\rho = 0.62$ and $\rho = 0.67$, indicating a moderate-strong correlation.

Looking at the two previous cases, `org.ehcache:ehcache` between version 3.7.0 to 3.8.0 had an increase in function reuse of 8.5% when looking at the full stitch mode. However, it had a LOC reuse decrease of -0.4664% despite importing more LOC. Similarly, removing megamorphic call sites suggests a more drastic reduction in reuse, namely around -20% for both modes. While surprising at first, the significant drop in reuse is not a response to drastic changes in how the project reuses third-party libraries; it is a result of adding `org.glassfish.jaxb:jaxb-runtime` that brings in more external code (*i.e.,* going from 405 to 8511 declared functions). The attributed increase in function reuse is likely due to new interface implementations being made availalbe

For `org.mockito:mockito-core` between version 1.3.0 to 1.5.0, there is a -4.6% reduction in LOC reuse (function reuse: -8.5%) on the full-stitch mode, and in contrast, when looking without megamorphic calls, we observe an increase of reuse, a 28% increase in LOC reuse (function reuse: 11%). In the case of mockito, removing a library dependency should generally bring up the reuse ratio, showing how not treating interface calls can potentially bring misleading results.

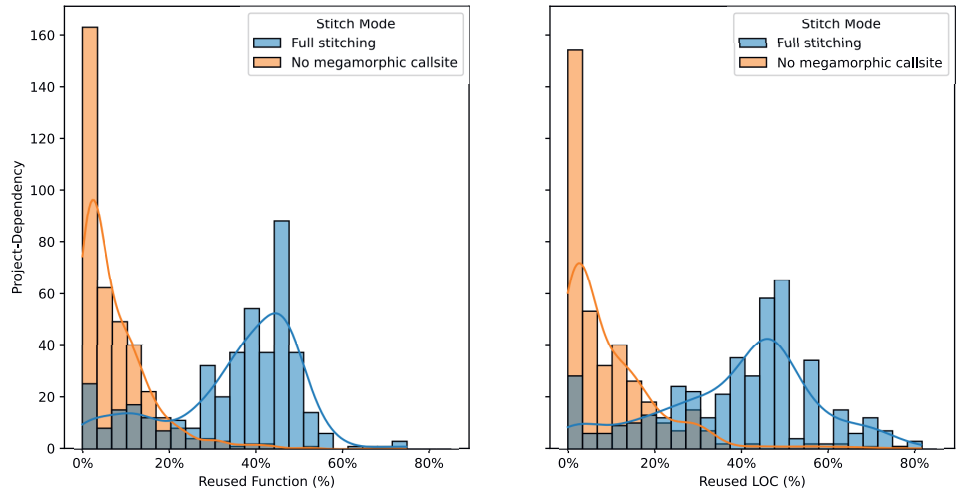


Figure 5.10: Third-Party Reuse per Project-Dependency Relationship (latest version)

Third-Party Reuse per Project-Dependency Relationship Figure 5.10 drills down the reuse ratio for individually declared third-party libraries instead of looking at the entire reuse of all third-party libraries. While we find similar patterns observed in Figure 5.7, we can more clearly identify the contrasts between full stitch mode and no megamorphic call sites; the full stitch mode identifies most declared third-party libraries to be reused between 30% to 60%, whereas, the no megamorphic call sites mode finds the reuse ratio to mainly vary between 0 to 20%. When comparing the median, we see the results competing: the median for no megamorphic call sites is 5.4% (functions: 4.27%), and 42% (functions: 38%) for full stitching. For example, the `commons-fileupload:commons-fileupload:1.5` reuse LOC ratio of `commons-io:commons-io:2.11.0` is 18% (2126/11653), and in the no megamorphic case, it is 4.3% (503/11653).

Considering the low reuse rate observed in the no megamorphic call sites scenario, developers should evaluate whether the operational benefits of third-party libraries justify the associated maintenance costs. As an illustrative example, eliminating `commons-io:commons-io:2.11.0` could remove two out of the five of its imported library dependencies, nearly half of the imported third-party libraries in `commons-fileupload:commons-fileupload:1.5`, potentially simplifying dependency management and reducing maintenance overhead.

Findings from RQ2: *Code reuse consistently lies within the 12-38% range. The handling of interface calls significantly affects the results: the upper-bound suggests an increase in reuse, while the lower-bound indicates minimal change. Between consecutive versions, reuse remains mostly unchanged. Overall, we observe that third-party libraries are predominantly underutilized.*

5.4 Discussion

Third-party libraries, specifically resolved through package managers, have helped developers and organizations effectively scale code reuse and reduce development time and costs. A side-effect is that projects own less first-party code and primarily depend on a large amount of third-party code; on average, our results suggest that projects consist of 87% third-party code. In light of recent software supply chain risks and stricter regulations around third-party libraries, there is an increased awareness to evaluate current reuse practices and form more responsible policies. We estimate that packages reuse between 12.2%-38% of their imported third-party code, leaving a significant amount—more than 50%—unused.

Low reuse of third-party libraries can be problematic as it increases the attack surface for vulnerabilities and malicious intents from a security perspective. Similarly, it raises questions about the cost of importing and using third-party libraries, including factors such as deviating standards on testing, breaking changes in version updates, and social factors such as abandonment of projects. Dealing with incidents related to third-party libraries could be costly in the long term and potentially outweigh the cost of having first-party code that serves the same functionality. For example, we identified one project release, `com.amazonaws:aws-java-sdk-s3` that inherits from its parent and depends on `org.apache.httpcomponents:httpclient` that only uses 43 LOC. Here, the low amount of reuse could motivate substituting third-party code with first-party code to avoid exposure to external risks, as this library also depends on another set of third-party libraries. While information such as the number of invoked external functions and LOC estimates the size of reuse, we argue that developers and organizations should evaluate how they reuse third-party libraries. Instances with very few lines or trivial dependence on third-party libraries may warrant using first-party code. Incidents such as the `left-pad` [5] incident is a strong reminder on how the impact of trivial third-party code could turn into a significant risk.

5.4.1 Reducing Unused Code In Third-Party Libraries

The unit of a package retrieved from package managers is a reusable library. As a significant amount of library code remains unused in projects, there could be reasons to explore mechanisms and best practices to organize and structure code in a way that optimizes reuse and minimizes the import of unused third-party code. One such example is introducing the project Jigsaw[4] in Java 9, a standardized module system intending to break down the Java Development Kit (JDK) and other large legacy libraries into reusable modules. As a result of modularizing the JDK, developers can create and distribute custom Java Runtime Environments (JREs) with a small print, which was impractical previously. GraalVM[5] follows another approach to reduce the amount of unused third-party code in production applications, including potential deserialization gadgets. It compiles ahead-of-time to create native binaries that only contain used functionality and thus have a smaller footprint. The use of reflection and other dynamic features, however, may require manual configuration to guide the compilation process.[6] Although a modularized system provides flexibility, creating a set of reusable modules from a library could be challenging. Thus,

[4] `https://openjdk.org/projects/jigsaw/`
[5] `https://www.graalvm.org/`
[6] `https://www.graalvm.org/22.0/reference-manual/native-image/Reflection/`

data-driven approaches, such as understanding how project reuse library functions, can aid in two ways: a) understanding usage patterns can drive how to organize the code and creation of module boundaries, and b) removing and deprecating APIs. In the latter case, providing data on code reuse to library maintainers could aid in removing largely unused or under-utilized APIs to reduce the attack surface and software supply chain risks.

5.4.2 Using Static Analysis Data in Studies

The imprecise treatment of Java's dynamic language features during call graph construction can impact the results of empirical studies. Since we rely on call graph algorithms known to construct at the scope of a project and its third-party libraries, the inexact handling of dynamically-dispatched calls (*i.e.,* interface calls) can be a source of influence on our results. By introducing a range with an upper-bound that includes all call targets and a lower-bound that eliminates megamorphic call sites, we observe not only a generally broad range between the two bounds, but also a notable difference in their appearance. Specifically, in Figure 5.8, we find that the median reuse grows over time in the upper-bound case, while in the lower-bound case, it remains essentially the same. As we observe this increase in reuse over time, we also notice an extra drive in reuse driven by the availability of new call targets, stemming from either the introduction of a new third-party library implementing a used interface or new implementations in existing third-party libraries. Therefore, when conducting studies involving call graph algorithms with inexact treatment of interface calls, our results emphasize the importance of considering such features in empirical studies and their potential influence on the outcomes. Our work provides a guideline on how to compute a lower-bound that eliminates megamorphic call sites.

5.4.3 Threats to Validity

Internal validity

We use the Lines of Code (LOC) metric to gauge the size of third-party libraries and reuse. However, this may not present a comprehensive view of libraries, as complexity changes might not necessarily result in added lines of code. To mitigate this limitation, we also count the number of public methods as a complementary metric for the size of import and reuse.

The OPAL call graph generator can resolve function invocations that involve static and dynamic dispatch and limited handling of calls that invoke the Reflection API. While recent studies [178, 179] suggest that this limitation does not significantly contribute to unsoundness, we must recognize that we cannot make definitive claims about the completeness of the generated call graphs due to the lack of ground truth for the analyzed projects together with their third-party libraries. Using additional analyses, such as dynamic analysis, to eliminate unlikely function targets as an alternative to imprecise handling of dynamic dispatch can be error-prone. It may lead to unsound inferences, primarily due to the need for more test coverage of projects and their library dependencies. As a result, we choose to mitigate this issue by specifying a range representing a lower and upper bound of our analysis.

Maven Central does not follow a strict versioning schema, unlike other package repositories such as JavaScript's `npm` or Rust's `Cargo`. As a result, there are user-defined versions for which we cannot determine any order, which is essential for identifying the differences

in changes between consecutive versions of releases. To address this issue, we adopt the approach proposed by Raemaekers *et. al.,* [11], which focuses exclusively on selecting packages and releases that adhere to the Semantic Versioning (SemVer) specification.

External validity
We recognize that the code reuse is not generalizable to other package repositories and only explains the properties of the Census-II dataset. Instead of a comprehensive repository scale analysis, we have a more focused selection of third-party libraries known to be widely used across enterprise applications. However, we expect that the patterns of import and reuse would yield similar results across other ecosystems, as the number of third-party libraries has generally grown for most language-based ecosystems [27]. Our approach can be replicated for programming languages with a resolver for package dependencies and a call graph generator.

5.5 Conclusion & Future Work

Our study examines how Java projects import and reuse third-party libraries over time. We use projects from the Census-II dataset representing widely used libraries in production applications. We construct call and dependency graphs for available project releases and their library dependencies and measure usage in reused lines of code and the number of externally invoked functions using reachability analysis. Dynamic dispatch calls complicate how we approximate the reuse of third-party code; to mitigate the risk associated with this, we compute an upper bound representing all interface calls and a lower bound that excludes megamorphic call sites. The difference we observe in our study between the approximation between the bounds highlights the importance of quantifying and treating imprecision in static analysis and quantifying the effect in results.

We observe that over a 17 year span that the proportion of external code in projects has historically remained high at around 75-80%, steadily growing to 87% in recent years, confirming that majority of code that ships with projects originates from libraries. At the same time, we find that projects reuse in the median range from 12.2% (lower-bound) to 38% (upper-bound) of available external code in recent years, with marginal growth in reuse in the lower-bound case and significant growth in the upper-bound case. Drilling down to the reuse of individual third-party libraries, we observe that the median project reuse is 6% for the lower-bound and 42% for the upper-bound, suggesting that third-party code remains largely unused. In line with previous work, we observe that a library's used features remain constant over time (a slight variation of ±1%).

In light of increasing software supply chain risks, our results demonstrate that projects have low reuse of third-party libraries, raising questions on the operational cost of using third-party libraries versus maintaining those lines internally.

6

Conclusion

This chapter summarizes the contributions of this thesis, revisits our original research questions, and concluding remarks on applying static analysis to software supply chain problems.

6.1 Contributions

The main contributions of the thesis can be summarized as follows:

- A new technique—Präzi—to construct code-based networks of package repositories by augmenting build manifest data with call graphs. We provide a reference implementation for Rust's Crates.io to evaluate the practicality of the technique, demonstrating that our approach can compile and infer call graphs for most releases made in Crates.io and new applications in the form of API-based reachability analysis such as tracking deprecation and bloat. Präzi also addresses the temporal properties of package repositories (i.e., new releases trigger build systems to update the dependency tree of packages automatically) for conducting evolutionary studies by introducing time as a variable, enabling on-the-fly network generation and stitching of call graphs for a timestamp *t*.
- A benchmark-inspired review protocol for evaluating how accurately dependency networks infer package relationships of a repository. This protocol zeroes in on package edges in the repository where network discrepancies occur, enabling a comparison of dependency networks. By manually reviewing these edges, we can establish a partial ground truth, which will help researchers and practitioners understand the trade-offs and limitations of different networks, specifically, how the absence or presence of language features in call graph generators affects the inference of package edges.
- Novel findings providing researchers and practitioners insights into practical trade-offs when using code-based and metadata-based networks of package repositories for reachability analysis. Package-based networks tend to overestimate package relationships, including edges where no code reuse is evident. Conversely, call-based

dependency networks effectively exclude such edges without code reuse. These call-based networks can underestimate relationships, as call graph generators may only accommodate some language features. Reachability analysis performed on code-based or metadata-based networks yields similar results for tasks related to direct package relationships. However, the same does not hold for transitive package relationships, where the reused functionality of a directly imported third-party library influences the portion of reused functionality in transitively imported third-party libraries.

- An automated infrastructure that assesses how effectively project test suites can identify semantically breaking changes during updates of third-party libraries. This system creates artificial updates by mutating functionality in direct and transitively imported third-party libraries, subsequently calculating a mutation score for these libraries. In parallel with project test coverage, the infrastructure also determines the test coverage of reused third-party functions. These calculated metrics provide developers with crucial insight into the sufficiency of their test suites for automated dependency updating tools and highlight any coverage gaps that may necessitate additional tests.

- A static technique that detects semantically breaking changes during updates of third-party libraries. As project test suites may not fully cover or effectively test certain reused third-party functions, static call graphs can bridge these gaps, as they estimate reuse based on the source code. By pinpointing the functions that have changed between the old and the new version, we can utilize reachability analysis to identify if a path from the altered function leading to the project exists. Practitioners, particularly tool makers, can use our technique to build hybrid approaches incorporating dynamic and static analysis. Such strategies could enhance the robustness of updating third-party libraries and diminish the likelihood of introducing regression issues. The technique is implemented in an open-source tool, Uppdatera, which works for Java-based projects.

- A baseline on code reuse of third-party libraries enabling practitioners and researchers to evaluate and comprehend whether projects overuse or underuse third-party code. Project maintainers can gain preliminary insights into the code reuse of specific third-party libraries by indicating the amount of imported and reused lines of code (LOC). Cases with low reuse could motivate its removal and insourcing might be better to cut operational costs and risks.

6.2 Research Questions Revisited

In this section, we revisit and answer the research questions from Chapter 1.

RQ 1 How feasible is code-based reachability analysis in practice?

In Chapter 2, we demonstrate the capability to build call-based networks that are representative of package repositories, proving that we can compile and generate a call graph for 70% of all indexed releases—a figure that escalates to 90% in practical circumstances

due to some indexed releases having compilation errors resulting from coding mistakes. Compiling and creating call graphs for Crates.io spans ten days, a relatively insignificant one-time expenditure considering the low incremental cost and scalability associated with adding new releases. This approach's effectiveness and practicality are further exemplified in Chapter 4 with Uppdatera, where compilation and call graph construction are comparable with the execution times of test suites in DevOps settings.

By establishing a partial ground truth to compare metadata-based networks with call-based networks, we ascertain that call-based networks can eliminate approximately 35% of the false edges present in conventional metadata-based networks, primarily resulting from unused or absent import statements. We also discover that the call graph generator's level of support for diverse language features significantly impacts the inferred inter-package relationships. For instance, in use cases demanding uninstantiated constructs like generic or conditionally compiled functions, the accuracy of package relationship inference may suffer.

In our initial comparative study of reachability analysis involving five security vulnerabilities presented in Chapter 2, we find that code-based reachability analysis is three times more precise than metadata-based analysis, albeit at the cost of lower recall. The recall is based on all Rust language features and would noticeably improve when we disregard language features such as generic or conditionally compiled functions. We extend our initial reachability analysis evaluation to an empirical study in Chapter 3. This more extensive study finds empirical evidence that code-based and metadata-based networks are interchangeable for reachability analysis involving direct package relationships and can serve as adequate approximations of each other. However, the networks display substantial differences when it comes to transitive package relationships. In February 2020, the average number of transitive dependencies reported was 17 for metadata-based networks and only 6 for code-based networks. Upon manually examining this discrepancy, we determine that metadata-based networks tend to overestimate transitive package relationships primarily because they miss to account for the context of how a package reuses its third-party libraries.

Implications: Our findings shed light on several shortcomings of metadata-based networks, commonly used by researchers and practitioners for studying software supply chain problems ranging from package distributions to applications. They also highlight scenarios where these networks can feasibly substitute code-based networks. For inferring metadata-based networks of package distributions, we ascertain that simply validating dependency descriptors is not sufficient; it is also essential to validate the source or binary of packages. Package repositories like Crates.io or npm do not validate the content of packages as part of their publication process, which can lead to issues. We find that 20% of Crates.io releases either fail to compile or include a failed release in their dependency tree, posing a significant threat to the validity of existing empirical studies.

Although adopting static analysis for software supply chain-based analysis may not always be feasible, there are instances where precision is not a critical factor. In such cases, call-based and metadata-based networks yield similar results when analyzing direct package relationships, suggesting that projects and packages are likely to reuse at least one API of an imported third-party library in practice. However, this does not hold for indirect package relationships; here, metadata-based networks predict poorly by adding

edges to all transitive package relationships. Package repositories and applications are not homogeneous collections of software—each application and project reuses a third-party library differently, and what they reuse from a library determines whether that extends to using a transitively imported library. For instance, if a third-party library uses another library for JSON parsing functionality, but the application using this third-party library does not use any JSON functionality, then the application would not use the transitively imported library.

We also underline the influence of the soundness of a call graph generator on the inferred call-based dependency network. Language features that call graph generators typically overlook, such as uninstantiated generic and conditionally-compiled functions, can be crucial in certain use cases, such as security. In such scenarios, metadata-based networks offer a more sound alternative. However, given that metadata-based networks may produce many false positives for transitive package relationships, combining both networks can be beneficial in cases where soundness is critical. This approach can reduce workload by offering specific source code paths simplifying the overall analysis.

RQ 2 How does code-based reachability analysis complement tests in third-party libraries?

Our prior research question evaluated code-based reachability analysis as a potential replacement for metadata-based reachability analysis. In this research question, we shift our focus to examining the application of dynamic analysis in software supply chain problems, specifically in upgrading a third-party library from one version to another without introducing semantically breaking changes.

Our results in Chapter 4 reveal that Open Source Software (OSS) projects in Java typically exhibit a test coverage of 58% for the functionality they reuse from directly imported third-party libraries and 20% for functionality from indirectly imported libraries. These figures underscore the substantial gaps in test coverage of third-party libraries. Furthermore, by applying mutation analysis to simulate artificial third-party updates, we determined that project test suites detect, on average, 47% of artificially introduced regressions in directly imported libraries and 35% in indirectly imported libraries. This indicates the limited effectiveness of conventional test suites in mitigating regressions introduced by third-party library updates.

However, we observed a noteworthy improvement when we implemented code-based reachability analysis as an alternative to conventional test suites for third-party library updates. We achieved a detection rate of 74% for regressions in directly imported third-party libraries and 64% in transitively imported libraries. This detection rate is nearly double that of traditional test suites, thus highlighting the potential of code-based reachability analysis to provide more comprehensive coverage of reused third-party library functionalities.

We applied code-based reachability analysis to 22 real-world dependency updates on GitHub in a practical test. This experiment revealed that it generated false positives in six instances due to dynamically dispatched calls and misclassifying refactorings. Despite this, code-based reachability analysis proved its value by complementing areas where test suites fell short. Specifically, it identified three updates containing semantic breaking changes that went undetected by the tests.

Implications: Similar to avoiding writing code using third-party libraries, we find empirical evidence that this also extends to not writing tests for reused third-party functionality. This lack of emphasis on third-party testing is not entirely surprising; most literature [107, 128, 129] on testing primarily targets first-party code, often neglecting the significance of third-party code. With the increasing use of third-party libraries in software projects, there is a growing need to establish best practices and discuss the necessity of writing tests for such libraries. The log4shell [181] incident is a stark reminder of this necessity: developers could not validate whether using the `log4j` logging library exposed them to the risk of remote Java executable injection. Although this area warrants further research, we recommend that practitioners write tests for critical functionality reused from third-party libraries to defend against hidden malicious intent within the software supply chain. Our evaluation technique, which employs mutation analysis, can offer first-hand insights into the effectiveness of existing test suites in dealing with third-party libraries and help determine where to focus testing efforts.

Mirhosseini *et. al.,* [84] qualitatively discovered that developers regard automated third-party library updates with suspicion, perceiving tests as unreliable. Our empirical findings corroborate this, given the significant gaps we identified in test coverage. To enhance reliability for users of automated dependency updating tools, offering insights into the test coverage of third-party library features could indicate whether functions changed in a third-party library already have coverage. Our dynamic call graph generator and the diffing component of Uppdatera can generate such information. Increased transparency and additional insights into pull requests could foster greater trust in the process.

When evaluating third-party libraries, our findings suggest that dynamic analysis by running test suites is insufficient, similar to static analysis through change impact analysis. Hence, software supply chain toolmakers should consider hybrid approaches, as static analysis can complement test suites in areas they cannot reach within third-party libraries. Static analysis can fill the gaps left by traditional testing approaches by approximating how projects reuse third-party code.

RQ 3 How do software projects reuse imported third-party libraries?

In Chapter 5, we use reachability analysis to gauge how projects reuse third-party libraries by calculating the ratio of reused lines of code to all external lines of code and the count of externally invoked functions compared to all available external functions. When we calculate the ratio of first-party to third-party code available to developers in OSS Java projects, we find that an average of 87% of reusable code stems from third-party libraries. The proportion of available third-party code has grown from 81% to 87% in ten years, likely attributable to the increasing number of resolved transitive third-party libraries or the expansion of library size. However, in our analysis of consecutive versions, we note that projects typically remain stable, with minimal inclusion or exclusion of third-party library code.

Upon identifying the percentage of third-party code available in Java projects, we estimate that projects reuse between 12.2% and 38% of their imported third-party code, leaving a considerable amount—more than 50%—unused. In our examination of trends and patterns of reuse over ten years, we notice that reuse largely stays consistent over time. This pattern holds even when comparing consecutive versions. Hence, our findings align

with prior research [42, 171] that examined API usage, indicating that reuse maintains a steady trend over time.

Implications: Our findings provide a benchmark for organizations and developers to evaluate their level of third-party library reuse. We observe an overall trend of increasing third-party library functions, likely due to increasing transitively imported and resolved third-party libraries in projects. However, projects tend to maintain a stable level of third-party library reuse over time, meaning they do not necessarily utilize more existing libraries as they become available. As a result, over 50% of available third-party library functionality is unused in projects, leading to questions about the efficacy of promoting third-party library reuse through package managers as a form of effective software reuse. Moreover, the abstraction of a library as a collection of classes and interfaces could, by design, contribute to code bloat and added maintenance overhead. As practitioners strive to reuse code with maximal development benefits and minimal maintenance and operational costs, researchers should investigate ways to minimize these costs, especially considering that libraries remain largely unused.

Practitioners often need more awareness of how much they reuse from third-party libraries. Therefore, our estimation technique and benchmark can assist them in evaluating the operational costs and risks associated with the placement of third-party code over first-party code. The infamous left-pad incident [3] is a notable example of the risks of using third-party libraries in projects. Importing a single library brings along a suite of other libraries, each with its development standards and practices. Consequently, each imported library requires monitoring to stay informed of security incidents and breaking changes. If an organization utilizes only a tiny fraction of a library (e.g., 50 lines of code) that could be replaced with first-party code, transitioning to first-party code would eliminate all associated operational costs and risks. Consequently, practitioners must consistently and proactively conduct evaluations to ascertain the long-term justification for each use of third-party libraries and consider actively swapping out existing third-party library usage with first-party codes. This practice helps balance immediate development cost savings and long-term operational costs.

6.3 Concluding Remarks

Revisiting our original objective:

> This thesis applies code-based reachability analysis to address software supply chain problems, focusing on enhancing the safety of third-party library updates and understanding code reuse of third-party libraries.

In this thesis, we have extensively investigated the application of code-based reachability analysis to software supply chain problems. Our focus has been on inferring dependency networks from source code, enhancing the safety of third-party library updates, and understanding code reuse patterns, particularly in light of transitive dependencies.

Our research establishes the practicality and viability of code-based reachability analysis. We have demonstrated that constructing call-based networks to represent package repositories is achievable and advantageous. Compared to their metadata-based counter-

parts, call-based networks significantly enhance the precision of reachability analysis by reducing false positives. However, we also found that the effectiveness of these networks is highly dependent on the call graph generator's characteristics and the specific requirements of each use case.

We further explored the role of code-based reachability analysis in testing third-party libraries. Conventional test suites, we discovered, exhibit significant coverage gaps when dealing with third-party library updates. Our research underlines the potential of a hybrid approach, which combines dynamic and static analysis. This hybrid methodology considerably increases fault detection rates, enhancing third-party library updates' safety.

In conclusion, this thesis reveals the substantial potential of code-based reachability analysis in addressing software supply chain problems. It demonstrates the practical implications of applying static analysis from third-party library updates to empirical studies of package distributions.

Bibliography

URLs in this thesis have been archived on Archive.org. Their link target in digital editions refers to this timestamped version.

References

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

[2] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 351–361. IEEE, 2016.

[3] Isaac Schlueter. The npm blog — kik, left-pad, and npm. `http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`, 2017.

[4] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, Feb 2018.

[5] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395. ACM, 2017.

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

[7] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, volume 17, pages 995–1010, 2019.

[8] Joseph Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis, Delft University of technology, 2015.

[9] Stephen P Borgatti, Ajay Mehra, Daniel J Brass, and Giuseppe Labianca. Network analysis in the social sciences. *science*, 323(5916):892–895, 2009.

[10] Tom Preston-Werner. Semantic versioning. `https://semver.org`, 2013.

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

[12] Adam Baldwin. Details about the event-stream incident. `https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident`, 2018.

[13] John Dunn. Pypi python repository hit by typosquatting sneak attack. `https://nakedsecurity.sophos.com/2017/09/19/pypi-hit-by-typosquatting-attack`, 2017.

[14] Ashley Mannix. Rust ecosystem working group. `https://github.com/rust-lang-nursery/ecosystem-wg`, 2018.

[15] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L Glassman. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

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

[17] Anderson Brian, Tolnay David, and Turon Aaron. The rust libz blitz. `https://blog.rust-lang.org/2017/05/05/libz-blitz.html`, 2020.

[18] Herbert A Simon. The science of design: Creating the artificial. *Design Issues*, pages 67–82, 1988.

[19] Roel J Wieringa. *Design science methodology for information systems and software engineering*. Springer, 2014.

[20] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14:131–164, 2009.

[21] Claes Wohlin, Martin Höst, and Kennet Henningsson. Empirical research methods in software engineering. *Empirical methods and studies in software engineering: Experiences from ESERNET*, pages 7–23, 2003.

[22] Joseph Hejderup, Moritz Beller, and Georgios Gousios. Rustpräzi replication package. February 2019, Zenodo. DOI: 10.5281/zenodo.8152060, 2019.

[23] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. Präzi: From Package-based to Call-based Dependency Networks. January 2021, Zenodo. DOI: 10.5281/zenodo.4478981, 2021.

[24] Joseph Hejderup and Georgios Gousios. Can We Trust Tests To Automate Dependency Updates? A Case Study of Java Projects. January 2021, Zenodo. DOI: 10.5281/zenodo.4479015, 2021.

[25] Joseph Hejderup, Henrik Plate, Anand Ashok Sawant, and Georgios Gousios. Evaluating the Impact of Third-Party Library Reuse in Java Projects: An Empirical Study. April 2023, Zenodo. DOI: 10.5281/zenodo.7874912, 2023.

[26] Raula Gaikovina Kula, Coen De Roover, Daniel M German, Takashi Ishio, and Katsuro Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 288–299. IEEE, 2018.

[27] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, 2019.

[28] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*, pages 95–110. Springer, 2018.

[29] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[30] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories*, pages 181–191, 2018.

[31] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 644–655. ACM, 2018.

[32] Filipe Roseiro Cogo, Gustavo Ansaldi Oliva, and Ahmed E Hassan. An empirical study of dependency downgrades in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2019.

[33] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. On the impact of using trivial packages: an empirical case study on npm and pypi. *Empirical Software Engineering*, pages 1–37, 2019.

[34] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node. js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[35] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. Using others' tests to identify breaking updates. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 466–476, 2020.

[36] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software.

In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 449–460. IEEE, 2018.

[37] 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. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 559–563. IEEE, 2018.

[38] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node. js applications: How far are we? In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1199–1203. IEEE, 2020.

[39] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 101–104. ACM, 2018.

[40] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to api deprecation? the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[41] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[42] Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 561–571. IEEE, 2018.

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

[44] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 195–204, 2010.

[45] Hoan Anh Nguyen, Tien N Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830. IEEE, 2019.

[46] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L Glassman. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2020.

[47] Yehuda Katz. Cargo: predictable dependency management. `https://blog.rust-lang.org/2016/05/05/cargo-pillars.html`, 2016.

[48] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[49] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *European Conference on Object-Oriented Programming*, pages 688–712. Springer, 2012.

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

[51] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, 2000.

[52] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. *Practical virtual method call resolution for Java*, volume 35. ACM, 2000.

[53] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[54] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, 1997.

[55] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994.

[56] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[57] Saba Alimadadi, Ali Mesbah, and Karthik Pattabiraman. Hybrid dom-sensitive change impact analysis for javascript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[58] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. Pycg: Practical call graph generation in python. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1646–1657. IEEE, 2021.

[59] Matúš Sulír and Jaroslav Porubän. A quantitative study of Java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 17–25. ACM, 2016.

[60] 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*, 29(4):e1838, 2017.

[61] Pedro Martins, Rohan Achar, and Cristina V Lopes. 50k-c: A dataset of compilable, and compiled, java projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 1–5. IEEE, 2018.

[62] Niko Matsakis. Introducing mir. `https://blog.rust-lang.org/2016/04/19/MIR.html`, 2016.

[63] Konstantinos Triantafyllou. A benchmark for rust call-graph generators. `https://users.rust-lang.org/t/a-benchmark-for-rust-call-graph-generators/34494`, 2019.

[64] Functions. `https://doc.rust-lang.org/stable/reference/items/functions.html`, 2018.

[65] Call expressions. `https://doc.rust-lang.org/stable/reference/expressions/call-expr.html`, 2018.

[66] Method-call expressions. `https://doc.rust-lang.org/stable/reference/expressions/method-call-expr.html`, 2018.

[67] Types: Trait objects. `https://doc.rust-lang.org/stable/reference/types/trait-object.html#trait-objects`, 2018.

[68] Macros. `https://doc.rust-lang.org/stable/reference/macros.html`, 2018.

[69] Jorge Aparicio. cargo-call-stack: Static, whole program stack analysis. `https://github.com/japaric/cargo-call-stack`, 2019.

[70] Rust Language Documentation. The Rust programming language. `https://doc.rust-lang.org/book/first-edition/trait-objects.html`, 2018.

[71] William G Cochran. *Sampling techniques*. John Wiley & Sons, 2007.

[72] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[73] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[74] Open source license compliance. `https://www.blackducksoftware.com/solutions/open-source-license-compliance`, 2018.

[75] Tidelift. https://tidelift.com/, 2018.

[76] Rustsec advisory database. https://github.com/RustSec/advisory-db, 2018.

[77] Anand Ashok Sawant, Maurício Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 561–571, New York, NY, USA, 2018. ACM.

[78] Rust performance warning. https://github.com/servo/servo/issues/17399, 2018.

[79] Rust semantical bug. https://github.com/crossbeam-rs/crossbeam-epoch/pull/53, 2018.

[80] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in Java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18, 2005.

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

[82] Will G Hopkins. *A new view of statistics*. Will G. Hopkins, 1997.

[83] Enys Mones, Lilla Vicsek, and Tamás Vicsek. Hierarchy measure for complex networks. *PloS one*, 7(3):e33799, 2012.

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

[85] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pages 470–481. IEEE, 2016.

[86] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. An empirical study on the impact of refactoring activities on evolving client-used apis. *Information and Software Technology*, 93:186–199, 2018.

[87] Anand Ashok Sawant and Alberto Bacchelli. fine-grape: fine-grained api usage extractor–an approach and dataset to investigate api usage. *Empirical Software Engineering*, 22(3):1348–1371, 2017.

[88] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[89] How to use semantic versioning. https://docs.npmjs.com/getting-started/semantic-versioning, 2018.

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

[91] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 791–796. ACM, 2018.

[92] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151. IEEE, 1995.

[93] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[94] Greenkeeper: Automated dependency management. `https://greenkeeper.io`, 2019.

[95] Dependabot: Automated dependency updates. `https://dependabot.com`, 2019.

[96] Whitesource renovate. `https://renovatebot.com`, 2019.

[97] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[98] Michael Hilton, Jonathan Bell, and Darko Marinov. A large-scale study of test coverage evolution. In *ASE*, pages 53–63, 2018.

[99] Pavneet Singh Kochhar, David Lo, Julia Lawall, and Nachiappan Nagappan. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, 66(4):1213–1228, 2017.

[100] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12. IEEE, 2017.

[101] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, volume 2, pages 109–118. IEEE Press, 2015.

[102] Brian Anderson. Test the downstream impact of rust crate changes before publishing. `https://github.com/brson/cargo-crusader`, 2018.

[103] Andrey Ponomarenko. Java api compliance checker. `https://lvc.github.io/japi-compliance-checker`, 2011.

[104] Gleb Bahmutov. Do not break dependant modules. `https://glebbahmutov.com/blog/do-not-break-dependant-modules`, 2014.

[105] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 64–73. IEEE, 2014.

[106] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[107] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[108] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 470–481. IEEE, 2016.

[109] Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[110] Alex Gyori, Shuvendu K Lahiri, and Nimrod Partush. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328. ACM, 2017.

[111] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic howpublisheds in theoretical computer science*, 217:5–21, 2008.

[112] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[113] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[114] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[115] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[116] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436, 2014.

[117] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 12–21. IEEE, 2012.

[118] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*, pages 511–522. ACM, 2018.

[119] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, 2018.

[120] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[121] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

[122] Eric Bruneton. Asm 4.0 a java bytecode engineering library. `https://asm.ow2.io`, 2011.

[123] Joanna Cecilia Da Silva Santos and Julian Dolby. Program analysis using wala. In *ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[124] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–261. ACM, 2019.

[125] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452. ACM, 2016.

[126] Mike Strobel. Procyon - a suite of java metaprogramming tools. `https://github.com/mstrobel/procyon`, 2016.

[127] Russ Cox. Surviving software dependencies. *Communications of the ACM*, 62(9):36–43, 2019.

[128] James A Whittaker. *How to Break Software: A Practical Guide to Testing with Cdrom*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[129] William C Hetzel and Bill Hetzel. *The complete guide to software testing*. QED Information Sciences Wellesley, MA, 1988.

[130] Nathan P Kropp, Philip J Koopman, and Daniel P Siewiorek. Automated robustness testing of off-the-shelf software components. In *Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224)*, pages 230–239. IEEE, 1998.

[131] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezze. Compatibility and regression testing of cots-component-based software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 85–95. IEEE, 2007.

[132] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[133] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[134] Anders Møller and Martin Toldam Torp. Model-based testing of breaking changes in node. js libraries. *changes*, 4:15, 2019.

[135] Darius Foo, Ming Yi Ang, Jason Yeo, and Asankhaya Sharma. Sgl: A domain-specific language for large-scale analysis of open-source code. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 61–68. IEEE, 2018.

[136] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[137] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM, 2011.

[138] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[139] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 9–pp. IEEE, 2005.

[140] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[141] Daniel M German, Ahmed E Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[142] Bixin Li, Xiaobing Sun, and Hareton Leung. Combining concept lattice with call graph for impact analysis. *Advances in Engineering Software*, 53:1–13, 2012.

[143] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

[144] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[145] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering howpublisheds*, volume 28, pages 128–137. ACM, 2003.

[146] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500. IEEE Computer Society, 2004.

[147] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[148] August Shi, Milica Hadzi-Tanovic, Lingming Zhang, Darko Marinov, and Owolabi Legunsen. Reflection-aware static regression test selection. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[149] Owolabi Legunsen, August Shi, and Darko Marinov. Starts: Static regression test selection. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 949–954. IEEE, 2017.

[150] Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 91–100. IEEE Press, 2019.

[151] Benjamin Danglot, Martin Monperrus, Walter Rudametkin, and Benoit Baudry. An approach and benchmark to detect behavioral changes of commits in continuous integration. *Empirical Software Engineering*, 25(4):2379–2415, 2020.

[152] Leuson Da Silva, Paulo Borba, Wardah Mahmood, Thorsten Berger, and Joao Moisakis. Detecting semantic conflicts via automated behavior change detection. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 174–184. IEEE, 2020.

[153] Prem Devanbu, Sakke Karstu, Walcelio Melo, and William Thomas. Analytical and empirical evaluation of software reuse metrics. In *Proceedings of IEEE 18th International Conference on Software Engineering*, pages 189–199. IEEE, 1996.

[154] James M Bieman. Deriving measures of software reuse in object oriented systems. In *Formal Aspects of Measurement: Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, 5 May 1991*, pages 63–83. Springer, 1992.

[155] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, 27(5):102, 2022.

[156] Joseph Hejderup and Georgios Gousios. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software*, 183:111097, 2022.

[157] Andreas Dann, Ben Hermann, and Eric Bodden. Upcy: Safely updating outdated dependencies. International Conference on Software Engineering (ICSE), 2023.

[158] Cyber resilience act. `https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act`, Sep 2022.

[159] Executive order on improving the nation's cybersecurity: Cisa. `https://www.cisa.gov/topics/cybersecurity-best-practices/executive-order-improving-nations-cybersecurity`, May 2021.

[160] Paul Sawers. Google open-sources clusterfuzzlite to secure the software supply chain. `https://venturebeat.com/ai/the-power-of-infrastructure-purpose-built-for-ai`, Nov 2021.

[161] Reproducible builds. `https://reproducible-builds.org`, 2019.

[162] Hilary Carter. Census ii of free and open source software - lish/lf. `https://data.world/thelinuxfoundation/census-ii-of-free-and-open-source-software`, Mar 2022.

[163] Frederick Brooks and H Kugler. *No silver bullet.* April, 1987.

[164] 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.

[165] Fenton E. Norman. *Software Metrics: A Rigorous Approach.* Chapman & Hall, 1991.

[166] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the api usage in java. *Information and Software Technology*, 73:81–100, 2016.

[167] 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*, 184:111134, 2022.

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

[169] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+1 popular java apis. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 400–410, 2016.

[170] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):45, 2021.

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

[172] Steven Raemaekers, Arie Van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 221–224. IEEE, 2013.

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

[174] Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

[175] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In Clémentine Maurice, Leyla Bilge, Gianluca Stringhini, and Nuno Neves, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43, Cham, 2020. Springer International Publishing.

[176] David F Bacon and Peter F Sweeney. Fast static analysis of c++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, 1996.

[177] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP'95—Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995 9*, pages 77–101. Springer, 1995.

[178] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1049–1060, 2020.

[179] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. Reflection analysis for java: Uncovering more reflective targets precisely. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 12–23. IEEE, 2017.

[180] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM*

39th International Conference on Software Engineering (ICSE), pages 507–518. IEEE, 2017.

[181] Condé Nast. A Year Later, That Brutal Log4j Vulnerability Is Still Lurking. `https://www.wired.com/story/log4j-log4shell-one-year-later`, 2023.

Curriculum Vitæ

Joseph Ibrahim Hejderup

1990/11/15 Date of birth in Tranemo, Sweden

Education

2/2017–2/2022 Ph.D. Student, Software Engineering Research Group,
Delft University of Technology, The Netherlands,
Fine-Grained Analysis of Software Supply Chains,
Supervisor: Dr. Georgios Gousios
Promotor: Prof. Dr. Arie van Deursen

8/2012–-5/2015 M.Sc. Computer Science, Delft University of Technology, The Netherlands,
Thesis: *In Dependencies We Trust: How Vulnerable Are Dependencies In Software Modules?*

8/2009–6/2012 B.Sc. Computer Engineering, Chalmers University of Technology, Sweden,
Thesis: *Plattformsoberoende Applikationsutveckling för Smarta Mobiltelefoner*

Experience

2/2022–-Present Member of Technical Staff,
Endor Labs, Palo Alto, California, USA

6/2015–-9/2017 Scientific Programmer, Software Engineering Research Group,
Delft University of Technology, The Netherlands,

4/2014–-6/2014	Visiting Researchers, Software AnaLysis and Testing (SALT), University of British Columbia, Canada.

List of Publications

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

2. Enrique Larios Vargas, *Joseph Hejderup*, Maria Kechagia, Magiel Bruntink, and Georgios Gousios: Enabling real-time feedback in software engineering. Published in Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, 2018.

3. Moritz Beller and *Joseph Hejderup*: Blockchain-based software engineering. Published in Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, 2019.

4. *Joseph Hejderup* and Georgios Gousios: Can We Trust Tests to Automate Dependency Updates? A Case Study of Java Projects. Published in Journal of Systems and Software, 2022.

5. *Joseph Hejderup*, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios: Präzi: From Package-based to Call-based dependency Networks. Published in Empirical Software Engineering, 2022.

6. *Joseph Hejderup*: On the Use of Tests for Software Supply Chain Threats. Published in Proceedings of the ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses, 2022.

Included in this thesis.

Titles in the IPA Dissertation Series since 2021

D. Frumin. *Concurrent Separation Logics for Safety, Refinement, and Security.* Faculty of Science, Mathematics and Computer Science, RU. 2021-01

A. Bentkamp. *Superposition for Higher-Order Logic.* Faculty of Sciences, Department of Computer Science, VU. 2021-02

P. Derakhshanfar. *Carving Information Sources to Drive Search-based Crash Reproduction and Test Case Generation.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2021-03

K. Aslam. *Deriving Behavioral Specifications of Industrial Software Components.* Faculty of Mathematics and Computer Science, TU/e. 2021-04

W. Silva Torres. *Supporting Multi-Domain Model Management.* Faculty of Mathematics and Computer Science, TU/e. 2021-05

A. Fedotov. *Verification Techniques for xMAS.* Faculty of Mathematics and Computer Science, TU/e. 2022-01

M.O. Mahmoud. *GPU Enabled Automated Reasoning.* Faculty of Mathematics and Computer Science, TU/e. 2022-02

M. Safari. *Correct Optimized GPU Programs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2022-03

M. Verano Merino. *Engineering Language-Parametric End-User Programming Environments for DSLs.* Faculty of Mathematics and Computer Science, TU/e. 2022-04

G.F.C. Dupont. *Network Security Monitoring in Environments where Digital and Physical Safety are Critical.* Faculty of Mathematics and Computer Science, TU/e. 2022-05

T.M. Soethout. *Banking on Domain Knowledge for Faster Transactions.* Faculty of Mathematics and Computer Science, TU/e. 2022-06

P. Vukmirović. *Implementation of Higher-Order Superposition.* Faculty of Sciences, Department of Computer Science, VU. 2022-07

J. Wagemaker. *Extensions of (Concurrent) Kleene Algebra.* Faculty of Science, Mathematics and Computer Science, RU. 2022-08

R. Janssen. *Refinement and Partiality for Model-Based Testing.* Faculty of Science, Mathematics and Computer Science, RU. 2022-09

M. Laveaux. *Accelerated Verification of Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2022-10

S. Kochanthara. *A Changing Landscape: On Safety & Open Source in Automated and Connected Driving.* Faculty of Mathematics and Computer Science, TU/e. 2023-01

L.M. Ochoa Venegas. *Break the Code? Breaking Changes and Their Impact on Software Evolution.* Faculty of Mathematics and Computer Science, TU/e. 2023-02

N. Yang. *Logs and models in engineering complex embedded production software systems.* Faculty of Mathematics and Computer Science, TU/e. 2023-03

J. Cao. *An Independent Timing Analysis for Credit-Based Shaping in Ethernet TSN.* Faculty of Mathematics and Computer Science, TU/e. 2023-04

K. Dokter. *Scheduled Protocol Programming.* Faculty of Mathematics and Natural Sciences, UL. 2023-05

J. Smits. *Strategic Language Workbench Improvements*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-06

A. Arslanagić. *Minimal Structures for Program Analysis and Verification*. Faculty of Science and Engineering, RUG. 2023-07

M.S. Bouwman. *Supporting Railway Standardisation with Formal Verification*. Faculty of Mathematics and Computer Science, TU/e. 2023-08

S.A.M. Lathouwers. *Exploring Annotations for Deductive Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2023-09

J.H. Stoel. *Solving the Bank, Lightweight Specification and Verification Techniques for Enterprise Software*. Faculty of Mathematics and Computer Science, TU/e. 2023-10

D.M. Groenewegen. *WebDSL: Linguistic Abstractions for Web Programming*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2023-11

D.R. do Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. Faculty of Science, Mathematics and Computer Science, RU. 2024-01

M.J.G. Olsthoorn. *More Effective Test Case Generation with Multiple Tribes of AI*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-02

B. van den Heuvel. *Correctly Communicating Software: Distributed, Asynchronous, and Beyond*. Faculty of Science and Engineering, RUG. 2024-03

H.A. Hiep. *New Foundations for Separation Logic*. Faculty of Mathematics and Natural Sciences, UL. 2024-04

C.E. Brandt. *Test Amplification For and With Developers*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-05

J.I. Hejderup. *Fine-Grained Analysis of Software Supply Chains*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2024-06