Novel Approaches to Systematically Evaluating and Constructing Call Graphs for Java Software

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Michael Reif, M. Sc.

geboren in Sömmerda (Thüringen).

Referent:	Prof. Dr.-Ing. Mira Mezini
Korreferent:	Prof. Dr. Eric Bodden
Datum der Einreichung:	6. Januar 2021
Datum der mündlichen Prüfung:	23. Februar 2021

Erscheinungsjahr 2021

Darmstädter Dissertationen
D17

.

Reif, Michael : Novel Approaches to Systematically Evaluating and Constructing Call Graphs for Java Software
Darmstadt, Technische Universität Darmstadt

Jahr der Veröffentlichung der Dissertation auf TUprints: 2021
URN: urn:nbn:de:tuda-tuprints-19286
URL: https://tuprints.ulb.tu-darmstadt.de/id/eprint/19286

Tag der mündlichen Prüfung: 23.02.2021

Abstract

Whether applications or libraries, today's software heavily reuses existing code to build more gigantic software faster. To ensure a smooth user experience for an application's end-user and a reliable software library for the developer, the shipped piece of software should be as bug-free as possible. Besides manual or automatic software testing, static program analysis is one possible way to find unintended behavior. While static analysis tools can detect simple problems using pattern matching, advanced problems often require complex interprocedural control- and data-flow analyses, which, in turn, presume call graphs. For example, call graphs enable static analyses to track inputs over method boundaries to find SQL-injections or null pointer dereferences. The research community proposed many different call-graph algorithms with different precision and scalability properties. However, the following three aspects are often neglected.

First, a comprehensive understanding of unsoundness sources, their relevance, and the capabilities of existing call-graph algorithms in this respect is missing. These sources of unsoundness can originate from programming language features and core APIs that impact call-graph construction, e.g., reflection, but are not (entirely) modeled by the call-graph algorithm. Without understanding the sources of unsoundness' relevance and the frequency in which they occur, it is impossible to estimate their immediate effect on either the call graph or the analysis relying on it.

Second, most call-graph research examines how to build call graphs for applications, neglecting to investigate the peculiarities of building call graphs for libraries. However, the use of libraries is ubiquitous in software development. Consequently, disregarding call-graph construction for libraries is unfortunate for both library users and developers, as it is crucial to ensure that their library behaves as intended regardless of its usage.

Third call-graph algorithms, are traditionally organized in an imperative monolithic style, i.e., one super-analysis computes the whole graph. Such a design can hardly hold up to the task, as different programs and analysis problems require the support for different subsets of language features and APIs. Moreover, configuring the algorithm to one's needs is not easy. For instance, adding, removing, and exchanging support for individual features to trade-off the call graph's precision, scalability, and soundiness.

To address the first aspect, we propose a method and a corresponding toolchain for both a) understanding sources of unsoundness and b) improving the soundness of call graphs. We use our approach to assess multiple call-graph algorithms from state-of-the-art static analysis frameworks. Furthermore, we study how these features occur in real-world applications and the effort to improve a call graph's soundness.

Regarding aspect two, we show that the current practice of using call-graph algorithms designed for applications to analyze libraries leads to call graphs that both a) lack relevant call edges and b) contain unnecessary edges. Ergo, motivating the need for call-

graph construction algorithms dedicated to libraries. Unlike algorithms for applications, call-graph construction algorithms for libraries must consider the goals of subsequent analyses. Concretely, we show that it is essential to distinguish between the analysis's usage scenario. Whereas an analysis searching for potentially exploitable vulnerabilities must be conservative, an analysis for general software quality attributes, e.g., dead methods or unused fields, can safely apply optimizations. Since building one call graph that fits all needs is nonsensical, we propose two concrete algorithms, each addressing one use case.

Concerning the third aspect, we devise a generic approach for collaborative static analysis featuring modular analysis that are independently compilable, exchangeable, and extensible. In particular, we decouple mutually dependent analyses, enabling their isolated development. This approach facilitates highly configurable call-graph algorithms, allowing pluggable precision, scalability, and soundiness by either switching analysis modules for features and APIs on/off, or exchanging their implementations.

By addressing these three aspects, we advance the state-of-the-art in call-graph construction in multiple dimensions. First, our systematic assessment of unsoundness sources and call-graph algorithms reveals import limitations with state-of-the-art. All frameworks lack support for many features frequently found in-the-wild and produce vastly different CGs, rendering comparisons of call-graph-based static analyses infeasible. Furthermore, we leave both developers and users of call graphs with suggestions that improve the entire situation. Second, our discussion concerning library call graphs raises the awareness of considering the analysis scenario and opens up a new facet in call-graph research. Third, by featuring modular call-graph algorithms we ease to design, implement, and test them. Additionally, it allows project-based configurations, enabling puggable precision, scalability , and sound(i)ness.

Zusammenfassung

Software ist fehlerhaft. Dieser Tatsache müssen sich Entwickler:innen stellen. Schon im Prozess der Softwareentwicklung kommen zur Fehlervermeidung deshalb vielfältige Verfahren zur Anwendung, wie bspw. der Einsatz von Tests, Code Reviews und statischer Programmanalyse. Damit können Fehler frühzeitig erkannt und beseitigt werden. Statische Programmanalysen haben im Vergleich zu den anderen genannten Qualitätssicherungsmaßnahmen den Vorteil, dass sie bekannte, häufig auftretende Fehler auffinden und programmübergreifend eingesetzt werden können. Für einfache Probleme eignen sich Mustererkennungen, komplexere interprozedurale Probleme bedürfen hingegen auch Kontroll- und Datenflussanalysen–letztere setzen den Einsatz von Call Graphen voraus. Die bisherige Arbeit mit Call-Graph-Algorithmen weist in der Forschung jedoch noch Lücken auf, obwohl sie ein zentraler Grundbaustein von interprozeduralen, statischen Programmanalysen sind.

In dieser Arbeit adressieren wir drei Probleme der bisherigen Verwendung von Call-Graph-Algorithmen, die einen negativen Einfluss auf die Ergebnisse von interprozeduralen, statischen Programmanalysen haben können. Dies sind im Einzelnen: 1) die Vernachlässigung der Korrektheit[1] von Call Graphen, 2) der übermäßige Fokus auf der Entwicklung von Call-Graph-Algorithmen für Applikationen, bei gleichzeitiger Unterrepräsentation dieser für Software-Bibliotheken, sowie 3) die mangelnde Modularität und Konfigurierbarkeit von Call-Graph-Implementierungen. Alle drei Aspekte tragen dazu bei, dass das Potenzial interprozeduraler, statischer Programmanalysen nicht ausgeschöpft werden kann. Dem begegnen wir mit verschiedenen, problemspezifischen Maßnahmen.

Dem Problem der mangelnden Korrektheit (1) setzen wir eine eigens entwickelte Methodik und ein dazugehöriges Werkzeug entgegen, mithilfe derer die Quellen fehlender Korrektheit identifiziert werden können. Damit bietet sich die Möglichkeit, diese zu verstehen und zu beseitigen.

Um die problematische Unterrepräsentation von Call-Graph-Algorithmen für Software-Bibliotheken (2) aufzuzeigen, gehen wir auf deren spezifische Eigenarten und notwendige Anpassungen vorhandener Call-Graph-Algorithmen ein, um bspw. unnötige Kanten zu vermeiden oder fehlende zu ergänzen. Mehrere konkrete Algorithmen werden von uns entwickelt und gewährleisten das Erstellen eines korrekten Call Graphen für Software-Bibliotheken.

Um Call-Graph-Implementierungen modularer und konfigurierbarer zu machen (3), entwickeln wir einen generischen Ansatz für die Implementierung von kollaborativen, modularen statischen Programmanalysen, die unabhängig voneinander kompilierbar,

[1] engl. soundness—die Eigenschaft einer Analyse, alle tatsächlich möglichen Programmzustände zu erfassen.

austauschbar und erweiterbar sind. So kann für jedes Programm die bestmögliche Einstellung für Geschwindigkeit, Skalierbarkeit und Korrektheit gefunden werden.

Mit dem Lösen dieser drei Probleme fördern wir den Stand der Technik von Call Graphen in mehreren Dimensionen. Zum einen offenbart unsere systematische Bewertung der Quellen fehlender Korrektheit signifikante Schwachstellen bestehender Algorithmen. Damit diese behoben werden können, geben wir konkrete Empfehlungen für die Entwickler:innen und Nutzer:innen von Call-Graph-Algorithmen. Zum anderen schärft unsere Diskussion über Call Graphen für Software-Bibliotheken das Bewusstsein für die Berücksichtigung des jeweiligen Analyseszenarios. Damit eröffnen wir eine neue Richtung für die Call-Graph-Forschung. Nicht zuletzt erleichtert unser Ansatz für modulare Call Graphen außerdem deren Entwicklung und ermöglicht projektspezifische Konfigurationen, wodurch u. a. eine flexible Anpassung an bestehende Herausforderungen möglich ist.

Preface

I get very fast hooked on puzzles. It is not very important what the problem is about; I somewhat want to solve it anyway. Someday, a dear friend of mine suggested a website[2] to me which was full of all kinds of computer-science-related puzzles. Starting with topics regarding logic and math, over reverse engineering, to various easy to hard programming ones. Those taught me a lot, brought tons of fun, and also flared up my interest in computer science.

First of all, I would like to thank Prof. Mira Mezini for supervising me over the course of my dissertation, for her assistance, and for giving me leeway to dive into my own directions and ideas during this time. I am grateful that she gave me the opportunity to find my own work and supported me when I needed it the most.

I also thank Prof. Eric Bodden for being the second examiner of my thesis. I am grateful for the time you spent on carefully reviewing my thesis as well as for your honest feedback.

Next, I want to thank Michael Eichberg, without whom I would not be where I am today. Your advice and guidance strongly influenced my work. With me not coming from a static analysis background, I am overly glad that you have taught me a lot and accompanied me over a large course of my PhD.

Over the years, I was happy to work with and supervise a number of excellent students, namely Florian Breitfelder, Roberts Kolosovs, Mario Trageser, Dominik Helm, Florian Kübler, Javor Bence Nikolov, Bekir-Melih Bayrak, and Malte Limmeroth, Andreas Bauer, and Tobias Peter Roth. Your hard work enabled me to explore many different ideas as well as maintenance of the many research prototypes that we published and contributed to over the years. I am very grateful for your contributions to ongoing research.

Furthermore, I want to thank all the people that were proofreading this thesis. Namely, I want to thank Sven Amann, Tobias-Peter Roth, Krishna Narasimhan, Lars Baumgärtner, and Dominik Helm. Even if I was not always agreeing with their opinions, I also want to shout out a big thank you to all the anonymous reviewers of my paper submission. I highly appreciated your feedback and most of the times it helped me to communicate more clearly and to improve my work.

Next, I want to gratefully thank Christina Cifuentes and her team at Oracle Labs Australia for having me as intern. This gave me the opportunity to grow both scientifically and personally. On top of that, it enabled me to put my research to practical use. Especially, I am thankful to my internship supervisors Yi Lu and Daniel Wainwright. In addition, I want to thank the entire team and interns, namely Sora Bae, Ben Barham,

[2] `http://www.happy-security.de/` (checked on Nov 12, 2020)

Andrew Browne, Christina Cifuentes, François Gauthier, Behnaz Hassanshahi, Alexander Jordan, Paddy Krishnan, Benjamin Barslev Nielsen, Brad Moody, Joonyoung Park, and Jörn Guy Süß for all the guidance, discussions, and great amount of fun. Also a big thanks to the crew that introduced me to climbing which by now became important factor in my work-life-balance and helped me to stay sane during the course of my PhD.

The past six years would not have been as exciting without my brilliant colleagues Matthias Bahr, Andi Bejleri, Oliver Bracevac, Ervina Cergani, Joscha Drechsler, Michael Eichberg, Matthis Eichholz, Sebastian Erdweg, Nafise Eskandani, Leonid Glanz, Sylvia Grewe, Dominik Helm, Ben Hermann, Sven Keidel, Mirko Köhler, Florian Kübler, Edlira Kuci, Johannes Lerch, Ingo Maier, Ragner Mogk, Patrick Müller, Sarah Nadi, Sebastian Proksch, Guido Salvaneschi, Jan Sinschek, Jurgen van Ham, Manuel Weiel, Pascal Weisenburger, and Anna-Katharina Wickert.

Without the invaluable support of Gudrun Harris, my PhD would have been significantly harder. Your assistance surely was one of the most significant factors on my PhD journey at the STG. You always had our back and made sure that we can concentrate on achieving our goal. Also your honest and clear way of communicating, your cakes and muffins, and also your humor made the time at STG even more enjoyable. Thank you very much for being there. That said, I also want to thank Claudia Roßmann for taking over administrative tasks from Gudrun. I am sure, that you will be as helpful to following generations of PhD students as Gudrun was to me.

Last but not least, I want to thank my girlfriend, my friends, and my family. Without your support I may not have been able to achieve this goal. I am filled with gratitude, that every single one of you motivated and pushed me further, when I needed it the most and accompanied me through out the roller-coaster-like journey of mine.

Editorial notice: Throughout this thesis I use the term “we” and “us” to describe my work. This is meant to underline that research is always a cooperative effort and that I would have much less (if something at all) to present here, if other people had not took the time off of their own work to review, discuss, and contribute to mine. I am deeply grateful for their effort.

Erklärung zur Dissertation

Hiermit versichere ich, die vorliegende Dissertation ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 6. Januar 2021

Michael Reif

Contents

1. Introduction

Globally involved, the enormous and steadily growing IT industry is expected to consume $3.46 trillion alone in 2020[1]. This very industry increasingly influences our quality of life by creating software and services penetrating all aspects of our lives. Considering only the software on our smartphones that wakes us up in the morning, connects us with family and friends, helps us organize ourselves, or is used to keep us fit, all of it tightly integrates into our daily routines. Not speaking of smart homes or the tools and services we increasingly depend on to work and collaborate remotely during the COVID-19 pandemic. While tremendously easing our daily lives, the reliance on these systems and services also establishes an enormous dependency.

This dependence increases the need for high-quality software to achieve both: a) minimizing the risk of security breaches, software crashes, safety issues, or unintentional behavior and b) maximizing the software's maintainability, usability, and extensibility. A report on the cost of poor software quality[2], published by the Consortium for IT Software Quality (CISQ), estimates the financial damage in 2018 within the US originating from bad quality software to $2.84 trillion.

One recommendation of CISQ is finding and fixing problems and deficiencies as close to the source as possible or preventing them altogether, meaning to best detect and fix them during the development process. Besides manual code reviews, using static analysis tools is one option to address this recommendation. To prevent bugs and deficiencies from reaching production code, the static analysis community offers a broad range of tools at the developers' disposal. These tools span from optimizing compilers [DGC95, PVC01] for producing efficient code, over automatic bug detection tools [LL05, HP18], to find misbehaving code or security vulnerabilities, to other tools helping to navigate through [KKD+11, SBMH17] or comprehend programs [Sie16]. Adopting these, developers can take advantage of various analysis tools at all stages of the software development process [SAE+18, DWA20, VPP+20].

As research has shown, those tools are most beneficial when the implemented analyses provide comprehensible reports and are practical, i.e., produce few false warnings [JSMHB13]. Despite these expectations, developers also wish these analyses to be fast and sound [LSS+15]. Remaining practical while expected to stay sound, modeling all possible program executions, is a fierce balancing act between precision, scalability, and soundness.

Trying to satisfy the expectations, almost all static analysis tools deliberately under-

[1] `https://www.gartner.com/en/newsroom/press-releases/2020-05-13-gartner-says-global-it-spending-to-decline-8-percent-in-2020-due-to-impact-of-covid19` (checked on Sept 28, 2020)

[2] `https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf` (checked on Apr 26, 2020)

approximate problematic program behavior and willingly accept unsoundness to boost its precision and scalability [Bod18], resulting in so-called *soundy* analyses [LSS+15].

Whenever a static analysis needs to process interprocedural information, it must know about the relationship between methods. Contributing this information, call graphs [Ryd79] are a) a fundamental static analysis data structure capturing possible execution-time relationships between program methods and b) a major source of soundiness [LSS+15].

Hence, a *soundy* call graph directly influences a static analysis' results to an unknown degree. This soundiness originates from programming language features or application programming interfaces (APIs) that are not or only partly modeled within the call-graph algorithm [LSS+15, BSS+11]. A potential reason might be the trade-off between the development costs for supporting problematic language features or specific APIs and the perceived value of coping with them. In the end, supporting those features is only relevant when they occur in the analyzed programs [LSS+15]. Yet, the impact of sacrificing soundness for precision or scalability concerning single language features or APIs on the call-graph construction is poorly understood and recently seeks research attention [SDTF20, SDTF20, TLR20]. Even less known are these effects for call graphs of libraries. So far, algorithms dedicated to construction call graphs for libraries are missing. As we show in this thesis, call-graph algorithms for applications only inadequately cover the specific needs of libraries.

This thesis is dedicated to call-graph algorithms and their implementations. It provides a systematic study of the deficiencies of the state-of-the-art and makes several constructive contributions to advance it. As the major portion of call-graph research, this thesis focuses on Java.

1.1. Problem Statement

Call graphs (CG) are directed graphs that capture possible execution-time relationships between methods in a program. Whereas the graph's nodes represent the program's methods, each edge represents at least one invocation of a method m_i by a method m_j. For an object-oriented language like Java, the CG is a directed, potentially cyclic graph.

The CG is a fundamental data structure for static analysis enabling more advanced interprocedural static analyses. A direct use case of CGs is to identify dead methods, i.e., methods that are unreachable. Another frequent use of CGs is, e.g., to combine them with control-flow graphs (CFG) to form interprocedural control-flow graphs (ICFG) [LR91, LR92, SHR01]. These in turn build the foundation for complex algorithms, such as solvers for data-flow problems [RHS95, SRH96], flow-sensitive points-to algorithms [DHS15], or security-related analyses [ARF+14, HREM15]. Hence, researching CGs also benefits interprocedural analyses, relying on them.

Over the last decades, the area of CG construction has received much attention [Ryd79, Shi88, DGC95, BS96, TP00, SHR+00, GC01]. Most of the existing approaches address virtual call resolution, but there is also work covering other aspects such as reflection [LWL05b], dynamic proxies [FKS18], dynamic invocations [FS19], or analyzing

incomplete programs [AL13]. Yet, several problems in the domain of CG construction remain, which we will elaborate on below.

Understanding the Effects of Accepting Unsoundness Existing research on CG construction algorithm mostly focuses on precision and scalability [LLA+15, GC01, TP00], thereby often covering only standard (non-)virtual method calls. Other more problematic language features, e.g., Java's serialization or reflection APIs, are often ignored; the developers deliberately accept so-called *soundy* [LSS+15] CGs. One reason for intentionally accepting unsoundness is the trade-off between soundness and precision/scalability. Another potential reason is the trade-off between the development cost for supporting such language features or specific APIs and the perceived value of doing so. We will use the term sound(i)ness to describe that an analysis is sound modulo a set of features.

As of now, the effects of supporting or not supporting a specific API or language feature are not well understood. The taken soundness trade-offs, along with their implications, often remain unbeknownst to the approach's users. Undisclosed trade-offs make it impossible to compare the results of different CG algorithms and, consequently, the relevance of research on CGs or interprocedural analyses using those. The trade-off's impact depends on the locations of uncovered language features in the project and, hence, is best assessed in a project-specific way. For instance, assume that a target project's main method uses reflection in combination with system properties to load the executing code. An CG algorithm, e.g., class-hiearchy analysis (CHA) [BS96], that does not cover these features would only contain calls to the reflection API; missing all application methods. As a result, the CG reaches only a fraction of the methods it should actually reach.

Library Call Graphs Another problem with state-of-the-art is that it does not cover well the specific needs of libraries. However, the use of libraries is ubiquitous in software development. Currently, the gold standard for constructing CGs for libraries is to use a standard algorithm, such as class-hierarchy analysis (CHA) [DGC95], rapid-type analysis (RTA) [BS96], or variable-type analysis (VTA) [SHR+00] and to consider all non-private methods as entry points. However, this ignores the nature of libraries that distinguish them from applications. Libraries are open worlds that can be extended by their users via inheritance, but they usually also provide public APIs as an interface. Ignoring these unique properties might lead to CGs that both miss call edges and also contain spurious ones. Hence, we lack a systematic discussion of library CGs and can consequently not estimate the need for CG algorithms targeting libraries specifically.

Modular Call-graph Algorithms Traditionally, CG algorithms are implemented in an imperative monolithic style, i.e., one super-analysis computes the entire CG. These monolithic designs become complex fast [BS09b], when one single analysis must support all the different features and APIs that are relevant to CG construction. More importantly, support for individual features or APIs cannot be developed in isolation, cannot be reused for other analyses, and cannot easily be added, removed, and exchanged to

trade-off between precision, sound(i)ness, and performance in a fine-tuned way. A modular approach to CG construction would be desirable to adapt the algorithms to the needs of the analyzed projects. In the end, the support of a feature is only relevant if it is present in applications. For example, support for Java 7's `invokedynamic` only became relevant after Java 8, i.e., to compile lambda expressions.

Summary To advance the state-of-the-art in CG construction and, consequently, also interprocedural analyses, we:

a) need a systematic evaluation of the state-of-the-art in CG construction,

b) require automatic documentation of a CG's capabilities,

c) must study the domain of library CGs,

d) must investigate the implications of supporting language features, APIs, and library-specifics and their relevance in real-world programs, and

e) require a modular approach to CG construction to easily enable trade-offs between precision, sound(i)ness, and performance.

This thesis contributes to all the above points.

1.2. Contributions of this Thesis

To start with, we contribute a systematic approach to assess and create Java corpora which provides the field's researchers with a tool to comprehend existing and to build efficient corpora for their needs. In particular, we use it in this thesis as a bases for our systematic study of Java language features and APIs impeding CG construction and to investigate their relevance in real-world applications. Our automated test suite covering these language features and APIs enables researchers and developers to improve CGs systematically and to compare new algorithms to existing ones with minimal effort. We use this test suite for our extensive assessment of well-known CG algorithms and, thus, provide an excellent overview of the field's state-of-the-art. Our automated pipeline for project-specific CG assessment allows us to evaluate how well-suited a specific algorithm implementation is for a particular project, inspect its weaknesses, and facilitates the project's comprehension. Our research on new CG algorithms targeting software libraries improves over the state-of-the-art, creates awareness for problems when analyzing incomplete codebases, and makes a significant step towards practically usable library CG algorithms.

Finally, our system for collaborative modular CG construction eases the development of CG algorithms not only allowing pluggable precision, sound(i)ness, and scalability but also rapid prototyping of new feature abstractions; bringing us one step closer to practical and more sound CGs. In the following, we elaborate a bit more on individual contributions.

A Query Engine for Code Features Our first contribution one is HERMES, a novel, generic open-source framework for the systematic assessment of Java projects. The proposed approach provides a code query engine with a flexible API that allows integrating arbitrary queries, e.g., to find API calls, language features, or to compute metrics. These queries can be run on a project to collect information about the project's structure and the language features that it uses. One can use the collected project information to either a) comprehend a project or b) find test or evaluation projects with specific properties. HERMES can automatically query a set of programs and compute a minimal set covering the queried features, which can then directly act as an evaluation, benchmark, or test set.

Assessment of the State-of-the-art in Call-graph Algorithms Our second contribution, is the design of a comprehensive and extensible framework, CATS, to test the recall of CG algorithms. First, we identify and present the Java language features and APIs that impair a CG's recall if not explicitly supported. Using these, we design a test suite and specify one unique test case for each identified feature and manually annotate the expected call edges as ground truth. Based on CATS, we assess the capabilities of CG algorithms of several state-of-the-art static analysis frameworks, uncovering their immense difference in the number of call edges and their poor performance concerning recent language features and APIs. As outcome of this assessment, we document each algorithms capabilities and capture it within a profile. Furthermore, we identify fundamental differences in how CGs are constructed, rendering actual implementations of the same algorithm incomparable. We published our test suite and our experimental results to allow other researchers to reproduce them and to evaluate further CG algorithms using our benchmark and compare them.

A Systematic Evaluation of Sources of Unsoundness in Call Graphs Our third contribution is JUDGE, a methodology and toolchain for analyzing CG algorithms in a project-specific manner. JUDGE builds on HERMES and CATS to thoroughly identify, understand, and evaluate sources of unsoundness in individual CG algorithms. Given a CATS' CG algorithm profile and an application, JUDGE finds and documents all sources of unsoundness in the applications, equipping users with the tool to decide whether the given CG is suitable for that specific program. Applying JUDGE, we examine the prevalence of Java language features and core APIs within different open source programs. Based on these insights, we give directives on what both static analysis researchers and framework developers can do to build and use CGs transparently. We also present an approach to reduce a CG's unsoundness manually. We published JUDGE along with our experiment data to allow other researchers to reproduce our results and facilitate further research on CGs and enable static analysis writers to implement, test, or debug their CG algorithms.

Design Space for Library Call-graph Algorithms In our fourth contribution, we address CG construction for Java libraries and investigate the implications of analyzing software

libraries in isolation, i.e., not integrated into an application. We motivate the need for CG construction algorithms dedicated to libraries and thoroughly discuss the design space for such algorithms. Furthermore, we design two different algorithms for two practical use cases; one of them is suitable to identify security issues, while the other one is better-suited for analyses that target general software quality issues. By evaluating both algorithms, we reveal unused code within the codebase of the Java Development Kit. While demonstrating the need and spanning up library CG algorithms' design space, we create awareness to consider the analysis' use case when choosing the CG and, thereby, broaden the CG construction area.

Modular Collaborative Call-graph Construction Our fifth contribution is a framework for composable CG construction. Instead of monolithic algorithms that address several language features and APIs, we propose an approach where various orthogonal analyses for individual language features and APIs collaboratively compute a single CG. As this framework is part of a collaboratively developed, more generic approach for modular and collaborative static analyses, we will present the more generalized approach. Our approach allows exchangeability and pluggable extension of analyses to improve sound(i)ness, precision, and scalability. Besides demonstrating the modular design of analyses, we use our approach to develop TACAI, our novel refinable intermediate representation.

Advanced Call-graph Algorithms for Libraries Contribution six is an investigation into advanced CG algorithms for Java libraries. We reimplemented four propagation-based application CG algorithms within our modular CG framework and then extended them to enable the construction of library CGs. Furthermore, we compare these four algorithms and determine their viability for library analysis.

1.3. Structure of this Thesis

This thesis is organized as follows.

In Part I, we a) introduce required terminology, b) give an overview over the state-of-the-art of CG construction for Java, static analysis frameworks, as well as their testing, debugging, and comparison approaches, and c) identify requirements on CG construction to cope with real-world software. We discuss CG algorithms and points-to analyses that can be used to construct CGs for either the whole program or program fragments only (Chapter 2). Furthermore, we present research about dynamic language features that impede the construction of CGs in practice. Then, we introduce the analysis frameworks offering implementations of the proposed algorithms and investigate how those implementations have been tested, benchmarked, and compared in the past (Chapter 3).

In Part II, we present our methods and tools for systematically assessing the quality of algorithms for CG construction. First, we introduce Hermes, our generic framework for the systematic assessment and the creation of corpora consisting of Java projects. We discuss how we can use Hermes to build an effective test suite or evaluation corpora

and conduct two case studies to show its usefulness (Chapter 4). Second, we identify multiple Java language features and Java core APIs that are relevant for the sound construction of a CG and propose our CG test framework, Cats, that covers them. Based on Cats, we conduct a study to evaluate state-of-the-art static analysis frameworks concerning their CG construction capabilities (Chapter 5). To estimate the potential effect of unsupported language features and APIs on the computed CG, we use Hermes—along with all the identified features and APIs—to further study their prevalence in numerous open source programs. Combining Hermes with Cats, we finally present and evaluate Judge, our tool to perform a project-specific evaluation of a CG's sources of unsoundness (Chapter 6).

In Part III, we discuss various aspects of how to design library CGs and implement them in a modular, collaborative fashion. First, we motivate the necessity for specific CG construction algorithms for incomplete programs, such as libraries. We discuss why an analyses' use case must be considered when constructing a library CG, introduce the open- and closed-package assumption, and span up a new design space for CG algorithms (Chapter 7). Depending on the analyses' scenario, we propose two library CG construction algorithms that differ in entry points and the so-called call-by-signature resolution. To demonstrate the approach's usefulness, we evaluate and compare the CGs in different use cases and conduct a case study to methods that are not used within the Java Development Kit (JDK).

Second, we present our novel approach to modular collaborative static analysis (Chapter 8). To figure out the needs for such a system, we perform three case studies. The first case study comprises a modular implementation computing three-address code, the second modular CG algorithm, and the third one comprises purity, escape, and immutability analyses. Then, we use our case studies to distill a list of requirements on such a framework. We propose our approach that leverages the modularity of blackboard systems and combines declarative and imperative static analysis techniques, satisfying all of these requirements. Showcasing its benefits, we provide evaluation results and compare the approach to a state-of-the-art static analysis tool.

Third, We present the intermediate representation that we use as input for our modular CG algorithms (Chapter 9). Further, we use our knowledge about CG construction for libraries and our framework for modular static analysis to develop more advanced library CG algorithms (Chapter 10). That is, we investigate how advanced CG algorithms from the set-based framework behave when we apply the open- and closed-package assumptions. Furthermore, we determine which of these algorithms is viable to use as a library CG.

In Part IV, we summarize this thesis and present an outlook to further future work on various topics in focus of this thesis.

1.4. Publications

Several publications have been co-authored during the doctoral project that resulted to this theses. We briefly overview them in following, thereby distinguishing between those

that are directly related to the contributions of this thesis and others.

1.4.1. Publications Directly Related to this Thesis

Many of the contributions presented in this thesis have previously been published to software engineering conferences or workshops. The rest of this section gives an overview over these publications and the respective parts of this thesis. The thesis parts may contain verbatim content of the publications.

Call-graph Construction for Java Libraries [REH+16] In this paper, we show that the current practice of using CG algorithms designed for applications to analyze libraries leads to CGs that, at the same time, lack relevant call edges and contain unnecessary edges. The former motivates the need for CG construction algorithms dedicated to libraries. Unlike algorithms for applications, CG algorithms for libraries must consider the goals of subsequent analyses. Specifically, we show that it is essential to distinguish between the scenario of analyzing for potentially exploitable vulnerabilities from the scenario of analyzing for general software quality attributes, e.g., dead methods or unused fields. This distinction affects the decision about what constitutes the library-private implementation, which needs special treatment. Thus, building one CG that satisfies all needs is nonsensical and gives plenty of different design options which are described in Chapter 7.

[REH+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for Java libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, 2016

Hermes: Assessment and Creation of Effective Test Corpora [REHM17] The paper presents Hermes, a framework for assessing a given corpus of Java projects and for the computation of a minimal corpus regarding the evaluated features. Hermes builds the foundation for our systematic evaluation of sources of unsoundness presented in Chapter 4. It presents an early version of the query infrastructure, an initial set of feature queries that partially address CG features and Java language APIs.

[REHM17] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 43–48, 2017

Systematic Evaluation of the Unsoundness of Call-graph Construction Algorithms for Java [RKEM18] The paper presents an early version our CG test framework, Cats, described in Chapter 5. This suite covers 64 language features and APIs relevant during CG construction. We use it to compare multiple CG algorithms of two major static analysis frameworks for Java Bytecode.

[RKEM18] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Proceeding ISSTA '18 Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, SOAP 2018, pages 107–112, 2018

Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs [RKE+19] This work presents JUDGE, our toolchain for the evaluation of language features and APIs that are relevant when building CG algorithms, comparing CG algorithms, evaluating how well-suited a specific algorithm is for a particular kind of project, and to facilitate the creation of project-specific sound CGs. JUDGE is presented in Chapter 6. This article further comprises extensive studies concerning the capabilities of four state-of-the-art Java static analysis frameworks and the prevalence of unsoundly handled features. Unfortunately, the results show that any CG algorithms lacks support for many features frequently found in the wild. Moreover, we find that comparing the results of static analyses that rely on CGs can be skewed.

[RKE+19] 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, 2019

Modular Collaborative Program Analysis in OPAL [HKR+20] This paper presents a novel approach to modular collaborative static analysis presented in Chapter 8. At first, we distill a list of requirements on frameworks for collaborative static analysis from three case studies. We then propose an approach that leverages the modularity of blackboard systems and combines declarative and imperative static analysis techniques, satisfying all of these requirements. Unlike monolithic analyses, our system allows exchangeability and pluggable extension of analyses to fine-tune an analysis's sound(i)ness, precision, and scalability. A thorough evaluation of the approach shows that it supports implementing various static analyses showing its generality. It further showcases its modularity features, good performance and provides promising results for its parallelization.

[HKR+20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 184–196, 2020

TACAI: An Intermediate Representation Based on Abstract Interpretation [RKH+20] In this paper, we present an intermediate presentation (IR) of Java Bytecode to facilitate the development of static analyses (Chapter 9). Concretely, we propose TACAI, a refinable, abstract IR that is based on abstract interpretation results of a method's bytecode. Exchanging the underlying abstract interpretation domains enables the creation

of various IRs of different precision levels. Our evaluation shows that TACAI can be efficiently computed and provides slightly more precise receiver-type information than Soot 's Shimple representation. Furthermore, we show how exchanging the underlying abstract domains impacts the generated IR.

[RKH+20] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. Tacai: An intermediate representation based on abstract interpretation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2020, pages 2–7, New York, NY, USA, 2020. Association for Computing Machinery

1.4.2. Other Publications

In addition, the author contributed to the following publications on static analysis during the work on this PhD thesis.

[HREM15] Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini. Getting to know you: Towards a capability model for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 758–769, New York, NY, USA, 2015. ACM

[GAE+17] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: Obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 638–648, New York, NY, USA, 2017. Association for Computing Machinery

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936, 2017

[EKH+18] Michael Eichberg, F Kübler, D Helm, M Reif, G Salvaneschi, and M Mezini. Lattice based modularization of static analyses. In *ISSTA Companion/ECOOP Companion*. ACM, 2018

[HKE+18] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. A unified lattice model and framework for purity analyses. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 340–350, 2018

[LWR20] Yi Lu, Daniel Wainwright, and Michael Reif. Probabilistic call-graph construction, May 2020. US Patent App. 16/200,045

[GMB+20] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, pages 694–707, New York, NY, USA, 2020. Association for Computing Machinery

1.4.3. My Contributions

As I stated in my preface, I am thankful to my collaborators and all who somehow influenced my research. Software engineering and static analysis research is often a collaborative effort. Thus, pinpointing contributions to single persons is not always possible. Contributions often arise from ideas that are then iteratively discussed and refined with colleagues, involving input from various participants. Moreover, most of my work is related to the static analysis framework Opal, whose maintenance was a joint venture with other researchers from our Software Technology group. All of us worked together on the entire framework, sometimes at overlapping parts—benefiting each other's work. Additionally, students play an indispensable role when they implement and evaluate ideas. This thesis also contains verbatim content from individual publications, including joint and sometimes indistinguishable contributions from colleagues. Next, I try to highlight my specific contributions as much as possible.

Chapter 2 and Chapter 3 present a literature review for relevant topics addressed in this thesis, solely done by me.

Chapter 4 discusses joint work with fellow researchers. I developed the idea and the concept of the publication on Hermes [REHM17]. However, software design discussions and parts of the implementations involved Michael Eichberg and Ben Hermann. Collaboratively, Michael Eichberg and I implemented Hermes' core and designed its APIs (cf. Section 4.1). Then, together with Ben Hermann, we implemented all the feature queries described in Section 4.1.2. I designed and evaluated Hermes on my own.

Chapter 5 includes several works that are joint efforts of different members in our Software Technology group. Most ideas and concepts are genuinely my work. Nevertheless, I supervised multiple students who supported me in the experimental evaluations and parts of the implementation [RKEM18, RKE+19]. This holds specifically for Sections 5.1, 5.2, and 5.3 where students assisted me in realizing the setup and running the evaluation. The Cats' (cf. Section 5.2) design and structure was my idea. However, it did undergo several iterations involving colleagues, a bachelor thesis, and student assistants until we ended up with the presented design. I invested much manual labor in studying the Java Language Specification [GJS+18a] and the Java Virtual Machine Specification [GJS+18b] as well as in validating the ground truth of our test cases. Adhering to good scientific practice, another colleague double checked the ground truth correctness.

Chapter 6 includes work from two related publications [RKEM18, RKE+19] with fellow researchers. Like Hermes and Cats, also Judge's pipeline was primarily my idea and concept. Although I developed the concept and the overall framework idea, its

realization involved several colleagues and students. This holds particularly for Sections 6.1 and 6.2 where student assistants contributed by implementing and evaluating my concepts. I developed the general test framework comprising the serialization of the call graphs, OPAL's framework adapter, the call-graph matcher, as well as the reporting. Furthermore, a student familiar with SOOT and WALA implemented their framework adapters and a colleague developed DOOP's framework adapter as well as the test-case extractor. I designed all for experiments presented in Section 6.2.2, 6.2.3, 6.2.4, and 6.2.5. To conduct our study of the prevalence of language features and APIs in-the-wild, we implemented a set of HERMES queries to find all the features we identified for our test suite in real applications. While I implemented most of these queries, other colleagues implement some too. Moreover, I ran the experiment and also analyzed the results. Then I discussed them along with my observations with fellow researchers. Analogously, I proceeded with experiment two (cf. 6.2.3) and experiment three (cf. 6.2.4). I personally performed the project-specific evaluation of *Xalan* using JUDGE, which is present in Section 6.2.5.

Chapter 7 presents joint work pertaining to library call graphs [REH+16], performed with members of the ST research group at TU Darmstadt. While Johannes Lerch brought up awareness for the trusted-method-chaining attack described in Section 7.1.2, I contributed to the work's general concepts. I brought up the idea of distinguishing the analysis setting, came up with the open- and closed-package assumption, studied much code to find and validate these assumptions, implemented LIBCHA$_{\text{OPA}}$ and LIBCHA$_{\text{CPA}}$, as well as designed the evaluation. However, discussions and joint design decisions with fellow researchers accompanied the entire process.

Chapter 8 includes collaborative work of several group members done in the context of the *CRISP* project. Although the general ideas and concepts are not my work, my involvements include supporting the implementation, discussing concepts, and performing the evaluation. I mainly contributed two case studies, one on modular call-graph construction (cf. Section 8.3.2) and another one on three-address-code generation (cf. Section 8.3.1), which we used to distill important requirements on a modular collaborative static analysis system.

Chapter 9 presents joint work with several researchers from the Software Technology Group. The publication on TACAI was my idea. Its foundation, OPAL's abstract interpretation engine, was implemented by Michael Eichberg. The initial implementation of TACAI and its maintenance was a collaborative effort. I evaluated the approach.

Chapter 10's content also presents partially joint work with several researchers as well as one master thesis. The initial implementations and evaluation of the different call-graph algorithms from Tip et al. [TP00], which we adapted to my concepts of library call graphs [REH+16], was the result of a master thesis. According to my ideas, the student implemented multiple call-graph algorithms initially, adapted them to enable library analysis, and evaluated them. I improved the implementation. Throughout the process, I guided the student's works.

Throughout many publications [REH+16, REHM17, RKEM18, RKE+19, RKH+20], I wrote most initial texts and was heavily involved in revising, restructuring, and polishing them.

Part I.

Background and State-of-the-art in Call-graph Construction and Comparison

2. Call-graph Algorithms

Call graphs (CG) are a significant building block of most interprocedural analyses used in static program analysis applications. Existing analysis tools support a wide range of tasks, such as compiler optimization, refactoring, bug detection, code comprehension, or code verification. Consequently, the domain of CG construction received a considerable amount of attention. Analogously, that applies to the problem of points-to analysis, i.e., it determines what heap abstractions a pointer can point to and, thus, can be used to generate a CG.

Most of this CG research concerns whole-program analysis, i.e., it assumes that the entire program is under analysis. However, in many practical applications, e.g., refactorings, that is not the case. Therefore, program-fragment analysis came up. It refers to analysis techniques capable of analyzing partial programs. So far, existing work in this research field only addresses scenarios in which the analyzed program is the application.

Furthermore, CG construction has to overcome many obstacles when applied to real-world applications. Programming languages as Java provide multiple APIs and language features that must be approximated when constructing CGs. These impediments lead to research on how individual APIs, frameworks, or language features can be approximated.

We discuss topics related to standard virtual method call resolution, i.e., general CG algorithms (cf. Section 2.1) before elaborating on points-to analyses (cf. Section 2.2). After that, we discuss general approaches that analyze program fragments (cf. Section 2.3). Finally, we discuss related work concerning Java's dynamic language features and specific APIs (cf. Section 2.4).

2.1. Application Call Graphs

Existing CG algorithms make a closed-world assumption, i.e., they assume the whole program being analyzed is available. Furthermore, their design mainly targets virtual invocations[1], i.e., callsites where the call targets are dependent on the runtime type of the receiver object. Hence, the following research disregards a) other program types, e.g., frameworks or libraries, and b) dynamic Java language features, e.g., the *invokedynamic* instruction or the reflection API.

However, we will find that the presented algorithms may be adaptable to other program types and relevant Java language features or APIs.

The easiest and least precise way to resolve polymorphic method calls in Java is to apply call-by-name semantic. When using call-by-name to resolve a method call,

[1] Virtual invocations are also known as dynamic dispatch or polymorphic callsites.

the called method's name determines all call targets. Consequently, the method call's potential receivers are all types that declare a method with the same name.

Analogously to call-by-name, call-by-signature algorithms resolve calls based on the method's signature, i.e., its return type, name, and parameter types. Besides call-by-name or call-by-signature CG algorithms, class-hierarchy analysis [DGC95] (CHA) is one of the simplest algorithms. CHA reduces the number of potential targets by only considering the methods declared within a subtype of the receiver, i.e., it takes the program's class hierarchy into account.

Bacon and Sweeney [BS96] show that rapid type analysis (RTA) improves over CHA by only considering subtypes that are instantiated by the considered application. In particular, RTA used for applications benefits from the fact that libraries usually define many types that are not used by an application, but nevertheless will be considered for call edges in CHA.

Tip and Palsberg [TP00] attribute different precisions of CGs to the number of sets used to approximate runtime values of expressions. They introduce the algorithm family CTA, FTA, MTA, and XTA. While CHA does not use a set at all, RTA uses one set to capture which types have been instantiated for the whole program. CTA uses distinct sets for classes, MTA uses distinct sets for classes and fields, FTA uses distinct sets for classes and methods, and XTA uses distinct sets for classes, fields, and methods. The rationale behind using multiple sets is to provide somewhat local type information and, thus, resolve callsites more accurately.

Sundaresan et al. [SHR+00] introduce declared-type analysis (DTA) and variable-type analysis (VTA). The more popular VTA uses a type-propagation graph, a directed graph where nodes represent variables and edges assignments between them. Sets of possible types are then assigned to each node, representing the runtime type a variable could potentially point to. Starting with allocation sites, these type sets are propagated along the directed edges of the graph. To determine possible call targets at call sites, the type set of the receiver's node is intersected with its statically possible subtypes. DTA is a less precise simplification of VTA that uses a single node per declared type (possibly merging multiple variables) instead of per variable.

Grove et al. present a framework that allows uniform modeling of multiple context-sensitive and context-insensitive CG algorithms [GC01]. They distinguish three contour selection functions that allow varying levels of context-sensitivity. Thereby, a contour denotes each context-sensitive version of a procedure. These functions enabled them to extend Shivers k-CFA [Shi88] to the more precise k-l-CFA algorithm.

To summarize, the design of these CG algorithms targets application CGs. However, researchers use these CGs for libraries and frameworks and consider all non-private methods as entry points, ignoring that libraries are a) open worlds and b) usually provide a public API.

Challenge 1: These CG algorithms target applications and lack a separate discussion of whether and how they can be applied to libraries.

2.2. Points-to Analysis

Points-to analysis, or pointer analysis, is a static analysis that determines to which heap references a variable or field can point to. Hence, points-to analyses are a precursor for more complex analysis such as escape analysis [CGS+99]. Given a points-to analysis, it is easy to generate a CG from its results because a points-to analysis knows the runtime types a variable—used as the receiver of a call—can point to. Even if points-to analysis is an extensively investigated field of research [RC00, SGSB05, SB06, Ste96, XR08, XRS09, ZXZ+14], we are not aware of works comprehensively discussing points-to analysis for analyzing libraries only. In particular, in such a scenario, not all allocation sites can be known. The correct result to what a parameter of a method callable by an application may point to is therefore not well defined, i.e., if the user creates new subtypes that extend a library class and implement a library interface. Moreover, which result is useful may depend on the use case the points-to analysis is applied to. For example, one could use a single allocation site to represent unknown allocation sites outside the library or multiple unique allocation sites for distinct entry points into the library. While the former is cheaper to compute and useful to answer *may alias* problems, the latter yields wrong results. Contrary, for *must alias* problems, the former is wrong.

Lately, Dietrich et al. [DHS15] presented a points-to analysis via transitive closure structure. They evaluate their approach on the library shipped with OpenJDK and can compute precise results in less than a minute. While they evaluate their approach on an extensive library only, they do not discuss whether the results remain correct in cases where a variable may point to an unknown allocation site outside the library.

Rountev and Ryder [RR01] presented an approach to construct summary information for libraries, which assumes all possible client applications. The summaries can be applied when constructing points-to information for a client application. They show that the results of their approach are equal to those computed by a whole-program analysis. They assume clients to be able to use all *exported variables*. While these include function references, a discussion is missing as what to include in exported variables for soundness and what can be excluded to increase precision. For example, some function references must be included to avoid trusted method chaining attacks, as discussed in Section 7.1.2, and others may be excluded under the closed-package assumption to increase precision.

Allen et al. [AKS15] discuss how to compute points-to information when analyzing a Java library in isolation. The core idea is to determine the so-called *most general application* (MGA) that subsumes all possible applications by using a single abstract allocation site per statically declared type of an entry point. Still, a discussion about the correct result of the points-to analysis is missing. Their solution uses a single abstract allocation site per statically declared type of an entry point. From their description, it seems that the approach misses call edges due to possible library extensions.

To recap, at first glance, many of the works published in the field seem to address computing CGs (or the larger points-to problem) for libraries. However, they address only parts of the problem. For instance, all works lack systematic considerations related to the inheritance of library classes. Hence, they disregard that the external world can potentially extend libraries.

Challenge 2: A systematic discussion related to the inheritance of library classes is missing, i.e., how does the extension of library classes from the outside world affects a CG.

2.3. Program-fragment analysis

Program-fragment analysis refers to analysis techniques capable of analyzing incomplete parts of a program. So far, existing work in this research field only addresses scenarios in which the analyzed program is the application. We address in this work the opposite case: analyzing a library while not knowing the client application.

Ali and Lhoták present the tool *Cgc*, capable of creating sound CGs without analyzing library code [AL12]. It makes use of the *separate compilation assumption*, i.e., that the library has been compiled without access to the code of the application, limiting the ways library code can interact with application code. Hence, the library cannot instantiate application classes (except via reflection, serialization, or cloning objects for which the results are unsound). Building upon this work, the authors introduce the tool *Averroes*, which generates placeholder code behaving as an over-approximation of the original library code [AL13]. *Averroes* allows using any whole-program CG algorithm on the application and the generated placeholder while still benefiting from analyzing a much smaller codebase than it would have to consider when including the original library code.

Rountev and Ryder [RMR04] present a fragment class analysis for testing. They generate a particular main method that over-approximates a test suite's behavior, enabling to apply existing whole-program analyses on a program's fragment. Their approach addresses test coverage computation only, using case-specific assumptions that do not hold in general.

Whereas works exist to analyze libraries or partial programs, they mainly concentrate on making them analyzable by bringing them to a state where one can use the already known techniques.

Challenge 3: We miss a disscussion of what it takes to analyze a library in isolation.

2.4. Dynamic Language Features

Livshits et al. [LWL05b] introduced the first static reflection analysis for Java. Their reflection analysis uses a points-to analysis to determine all possible sources of strings that flow into reflective calls as class names (e.g. `Class.forName()`)). Furthermore, if they detect that a class name's origin is outside the program, e.g., it is read from a file, they allow manual specification points where the approach's user can provide the required external input. However, when the analysis witnesses a reflective object creation (e.g., via `clz.newInstance`) without inferring the class' name, it tries to exploit intraprocedural cast operations to approximate the object's type. The latter was not only adopted by different static analysis frameworks such as DOOP [BS09b], WALA [IBM],

Soot [VRCG+10, LBLH11], Chord [NAW06], and OPAL [EH14] but also inspired many reflection analyses [LTSX14, LTX15, SBKB15, ZTLX17, ZLTX18, LTX19].

Java's Dynamic Proxy API creates type-safe proxy classes via runtime bytecode engineering, which will then forward the calls—using Java reflection—to a previously specified handler class. Fourtounis et al. [FKS18] recently introduced the first approach to deal with dynamic proxies. Using the Doop framework, they found that resolving dynamic proxies also requires support for other object flows (e.g., calls or reflective operations) as well as other program semantics (e.g., string tracking).

Since Java 7, the new *invokedynamic* bytecode instruction provides a call instruction with user-defined semantics [Ros09]. All current compilers starting from Java 8 use it to compile the newly introduced lambda expression construct, and compilers from Java 10 also use it to concatenate Java's strings. Only resolved at runtime, this instruction complicated the constructions of CGs [ARL+14]. Recently, Fourtounis et al. [FS19] published a deep static model for the invokedynamic instruction. Their approach fully models method handles and approaches the invokedynamic support at the language feature's fundamental level.

The research focus of the community is centering towards reflection. Other features, such as dynamic proxies or dynamic invocations via *invokedynamic* receive only a little attention. However, other Java features, APIs, and runtime-specific callbacks also impede CG construction, such as serialization, type casts, or static initializers, these features require more attention.

Challenge 4: Not all features that are relevant to CG construction are researched equally well.

Furthermore, it is unclear how relevant single programming language features and APIs are in practice. Still, understanding their practical impact is highly relevant, since the occurrence of the ignored features in real software can have a devastating impact on the constructed CGs.

Challenge 5: A systematic investigation of the practical relevance of single language features within real-world applications is missing.

3. Implementing and Comparing Static Analysis and Call-graph Algorithms

This chapter first introduces the state-of-the-art frameworks for static analysis that are used by the community to construct call graphs (CG) (cf. Section 3.1) as foundation for other static analyses. Then, we elaborate on how these frameworks and their static analyses tested and compared in terms of performance or precision (cf. Section 3.2). Afterward, we discuss studies that compare CG algorithms or their implementations with respect to precision and recall or their sound(i)ness (cf. Section 3.3).

3.1. Static Analysis Frameworks

The static analysis community working on the Java ecosystem often integrates their research in one of the state-of-the-art static analysis frameworks, such as Soot [VRCG+10], Wala [IBM], Doop [BS09b], or Opal [Pro18b, HKR+20]. All frameworks provide several different algorithms to construct CGs, but the set of implemented algorithms differs. Furthermore, it is not clear how the implemented CG algorithms compare concerning performance, precision, recall, and their support for different language features and APIs.

Soot Soot is a general-purpose static analysis framework that enables program analysis, manipulation, and optimization. It provides algorithms to construct CGs directly (e.g., RTA or VTA) or via points-to analysis (SPARK [LH03]) on-the-fly. To facilitate the development of static analysis, Soot provides several intermediate representations, such as *Baf*, *Jimple*, *Shimple*, or *Grimp*. However, most analyses rely either on *Jimple* or *Shimple*. Whereas the former is a typed three-address intermediate representation, the latter is additionally transformed into a static single assignment form.

Soot requires the entire program on the classpath to generate a CG. If Soot detects unresolved classes during the analysis, e.g., classes not on the classpath, it stops immediately. However, it is possible to configure Soot such that it ignores unknown classes. Whereas it is then possible to perform an analysis, using this option still leads to the loss of essential information, such as class-, method-, and field signatures of the ignored classes. Soot's CGs not only handle virtual calls but also model some implicit invocations triggered by the Java Virtual Machine. For example, they support static initializers, finalizers, and calls of `Thread.run()`. Soot's CGs support different configuration options to enable static reflection support, such as *safe-forname* and *safe-newinstance*. For instance, when Soot's RTA is used, activating these options forces the RTA to consider all types as instantiated when `Class.forName(...)` or `Class.newInstance(...)` is called. Hence, the resulting CG will degenerate to a CHA CG.

Wala WALA is another static analysis framework for Java and JavaScript, developed by IBM Research. Besides general analysis utilities, data structures, interprocedural dataflow solver, and intermediate representation form to facilitate static analysis, WALA can generate various CGs. These are either constructed by points-to analysis or by specific CG algorithms (e.g., RTA). Additionally, WALA provides multiple configuration options to resolve reflective calls, e.g., `Class.forName(...)`. WALA also enables the user to specify packages to be excluded from the analysis. This option is enabled by default. If not configured otherwise, WALA always excludes several Java-related packages (e.g., java.awt). Due to this feature, WALA might construct imprecise or unsound CGs.

Doop DOOP is the state-of-the-art framework for declarative static points-to analysis for Java. It implements a wide range of points-to analysis, including context insensitive, call-site sensitive, and object-sensitive analysis. While performing points-to analysis, DOOP also constructs CGs for the entire input program on-the-fly. DOOP relies on Datalog to implement static analyses in a strictly declarative manner using a rule-based approach. Rules can easily be exchanged or added (e.g., for new APIs or language features), enabling high configurability. For instance, DOOP has many analysis options to enable/disable support for various forms of reflection, dynamic proxies, etc. Furthermore, DOOP emulates the behavior of some common native methods, e.g., `Thread.run()`. Finally, when a DOOP analysis is configured, it uses SOOT to parse the target program, transforms it to SOOT's *Shimple* representation, and then generates the required Datalog facts from *Shimple*.

Opal OPAL is an extensible framework, written in Scala, to analyze, process, engineer, and manipulate Java bytecode. To support these different analysis tasks, OPAL provides several data structures and algorithms specific to static analysis. Its unique selling point is its highly-customizable framework for the lightweight abstract interpretation of bytecode. OPAL's abstract interpretation can treat each method as a potential entry point, making no assumptions over the program's state or a method's parameters. However, it supports interprocedural, flow-, path-, object-, and context-sensitive analyses. If the analysis is interprocedural, one can configure the length of the call chain. OPAL provides several preconfigured abstract domains to make it easier for developers to use its abstract interpretation feature. Based on these, OPAL provides a configurable three-address code intermediate representation, encoding different information depending on the employed abstract domains. Additional information derived from that abstract interpretation can comprise whether a variable is *null*, a branch is unreachable, or that two variables are aliases. Furthermore, OPAL also provides several CG (e.g., CHA and RTA) and points-to algorithms.

The research presented in this thesis was done and incorporated in OPAL.

To recap, we do not know how existing static analysis frameworks (SOOT [VRCG+10], WALA [IBM], DOOP [Sma18], and OPAL [EH14, HKR+20]) compare in terms of performance and capabilities of their CG algorithms. This is also relevant when deciding

which framework and algorithms to use, especially, since preliminary studies [MNGL98, SDTF20] have shown that CG implementations vary more widely than expected. Unfortunately, we lack methods and tools to investigate this problem in a systematic way.

Challenge 6: We require methods and tools for systematically analyzing and understanding the capabilities of CG algorithms with respect to their supported language features and core APIs.

3.2. State-of-the-art in Comparing Static Analyses

The defacto standard of proposing new algorithms is to publish a prototype implementation and an evaluation of the new approach, showing its feasibility. Fully establishing these experimental results requires independent reproduction. Finding an uncomfortable large number of results failing this test, the Association for Computing Machinery (ACM) lays the foundation for a formal artifact evaluation process which is already implemented by many ACM conferences and journals [1]. As a result of this artifact evaluation process, published works can earn badges, verifying the research's functionality, reusability, availability, or integrity.

To obtain these badges, static analysis researchers open-source their implementations and evaluate their approach using well-established corpora [Liv05, BGH+06, TAD+10, DSST17] for easier comparison to the state-of-the-art, or present and publish their self-made data sets [ARF+14, REH+16, GMB+20]. As every approach or analysis targets specific needs and goals, naively falling back to existing corpora as well as defining new data sets poses several challenges.

However, the creation of an unbiased, representative, and long-lived corpus is difficult. The lack of such corpora in various research areas has led authors to build their corpora, which differ in particular in two dimensions: a) criteria for project selection and b) evaluation goals.

Blackburn et al. [BGH+06] created the DaCapo benchmark suite, which primarily targets Java performance evaluation. They also discussed how to develop and test such corpora. They determined that their benchmark should consist of diverse and easy to use real-world applications. Besides these criteria, they identified a set of dynamic and static software metrics to assess a project's performance behavior.

Tempero et al. [TAD+10] first identified size, content, representativeness, and permanence as critical aspects for project selection. Based on these criteria, they created a curated code collection of 100 Java projects. These projects range from libraries over application frameworks to different kinds of applications. The focus of the Qualitas Corpus is on aiding researchers to carry out empirical studies of code.

In SecuriBench, Livshits et al. [Liv05] selected large web applications, which have known security vulnerabilities. Consequently, SecuriBench can be used to evaluate static and dynamic security analyses. Other corpora like DroidBench [ARF+14], Point-

[1] `https://www.acm.org/publications/policies/artifact-review-and-badging-current` (checked on Sept 18, 2020).

erBench [SNAB16], or the Darmstadt Library Corpus (DLC) [REH+16] provide data sets with yet different goals as well as different criteria to assemble the corpus.

The design of the previously introduced corpora has one specific goal in mind, but their suitability with respect to their original goals often remains unknown. Especially the inclusion of (yet) another real-world project into a corpus is repeatedly justified based on its perceived difference, rather than based on qualitative measures. Additional measures like goal-relevant metrics, e.g., the degree to which a project uses Java reflection or the occurrence of specific properties—already used by some corpora—could be used to assess the suitability regarding a given goal. However, the static analysis area pursues many goals ranging from entire static analysis frameworks [VRCG+10, EH14] over data-flow analyses [ARF+14, LSBM15] to lower-level CG algorithms [AL12, REH+16]. Furthermore, as programs and programming languages improve steadily, old corpora often become out-dated quickly. Conversely, hand-picked data sets can be recent, but it is harder to compare them to state-of-the-art analyses, not supporting the current software version.

Furthermore, most of the previously presented corpora are no longer maintained, indicating the difficulty keeping them up-to-date. Efforts have been made to partially or completely automate corpora creation to address this shortcoming.

Dujmović et al. [Duj10] presented a parameterized approach to automatically generate fully synthetic programs that already allow benchmarking and testing but is unusable to evaluate a system on real-world applications.

Nguyen et al. [DEB16] present Automatic Benchmark Management (ABM), a methodology for mining software repositories to semi-automatically extract an up-to-date, updatable, and representative corpus that includes applications from various domains. However, no assessment is done when including the projects, and it may be the case that many projects do not have relevant differences.

These corpora are good starting points to build up-to-date, comprehensive evaluation- or test corpora. Nevertheless, using these blindly does not ensure a meaningful evaluation or testing setup since the corpora might over- or under-represent relevant features.

Challenge 7: It is unclear to which degree the constructed corpora support the evaluation goals and to which degree all relevant properties, like occurrences of programming language features, the usage of certain APIs, and problematic design patterns, can be found in the projects. This lack of knowledge of the properties of the used projects generally leads to questionable evaluation results.

3.3. State-of-the-art in Call-graph Comparison

Call graphs are a central data-structure required by many static analyses, ranging from the detection of unused methods [EHMG15] to advanced control- and dataflow analyses [ARF+14]. The algorithm used for constructing a CG is directly impacting a client analysis' results; often even to a considerable extent. Hence, the algorithm's choice and the quality of its implementation, therefore, predetermines an analysis' precision and

recall. The research efforts we present below compare CGs, which is the first step to understand differences in CGs and their effect on subsequent analyses.

Murphy et al. [MNGL98] conducted a study where they compared CG algorithms for C. They found that CGs emitted by different tools vary for identical input programs and deemed the barely understood practical effects of approximations as the problem's origin. Furthermore, they discussed how one should choose a CG algorithm and recommended to check its input constraints, its documented or implicit design decisions, and its correctness for one's needs. However, such information is generally not available.

Lai et al. [LLA+15] discussed CG construction for different kinds of Java codebases with respect to potential sources of unsoundness and imprecision. However, they solely focused on programs compiled from JVM-hosted languages such as OCAML, Jython, Scheme, Scala, or JRuby. They aim to describe the challenges that arise when constructing CGs for such programs and only used Wala for their analysis. For their study, they focused on minimal, artificial code examples, and the identified sources of unsoundness were reflective calls and `invokedynamic` usage.

Lhoták [Lho07] presented a tool that enables a manual, qualitative comparison between two CGs by first finding differences and then inspecting them. The work targets the debugging of CG algorithms, requiring manual inspection of the generated CGs. Also, systematic identification of sources of unsoundness is not possible if the compared CGs both miss some edges; in that case, the graphs would be incomplete.

Sui et al. [SDE+18] compared Soot, Wala, and Doop's CG implementations using a micro-benchmark suite, taking the test program's execution environment into account. They measure the recall and also the precision of the tested algorithms. In follow up work [SDTF20] they used a hybrid approach to extract patterns that cause statically generated CGs to be unsound. Utilizing a subset of the XCorpus [DSST17], they compute a diff between dynamically recorded context call trees and static CGs generated with Doop. Comparing the results, they perform a root cause analysis investigating features that are not supported by Doop's CG. They found that not only Java reflection but also serialization and native methods of the Java Virtual Machine are significant reasons for unsoundness within their data set.

Other works presented CG algorithms or algorithm families [Shi88, TP00, GC01, ARL+14, ARL+15], evaluated and compared them concerning their size, the number of reachable methods, poly- and monomorphic callsites, and runtime. Their CG comparison focuses on the CGs' size and their capabilities to resolve polymorphic calls and, therefore, their precision.

Another CG algorithm family concerns the Scala language. Ali et al. [ARL+14, ARL+15] show that Java CGs are too imprecise when used on Scala—due to lost type information and Scala's custom features such as traits—and, therefore, propose several low-cost CG algorithms targeting Scala. However, they generate their CGs from source code, not investigating if the Java compiler emits code that must explicitly be handled by bytecode-based static analysis frameworks, e.g., Scala's custom *invokedynamic*s.

Challenge 8: Mostly, CG algorithms are compared concerning their capabilities to resolve virtual calls. Although some research recently investigated the soundness of CGs, more systematic and automatically repeatable studies are needed.

Despite the research efforts, the community accepts *soundy* CG algorithms and uses them for their interprocedural analyses without questioning their suitability. Moreover, these analysis are then evaluated on corpora or hand-picked programs and compared to the state-of-the-art analysis without investigating the effect of the underlying CGs.

Challenge 9: Call-graph implementations often come with undocumented implicit design decision and input constraints. The community still lacks methods to automatically assess and document a CG algorithm's capabilities.

Moreover, no tools exist that help to explore the origins and effects of under-approximated features and, therefore, the introduced unsoundness. Still, understanding their practical impact is highly relevant, since the occurrence of the ignored features in real software can have a devastating impact on the constructed CGs. This impact depends on the locations of uncovered language features in the project and, hence, is best assessed in a project-specific way.

Challenge 10: We need an systematic investigation on the implications of (not) supporting individual language features and APIs and their relevance in real-world programs.

Part II.

Methods and Tools for Systematically Assessing the Quality of Algorithms for Call-graph Construction

Comparing and Evaluating Static Analyses and Call Graphs

To compare and evaluate static analyses, researchers have proposed several corpora, benchmarks, and test suites. Chapter 3 of this thesis presented state-of-the-art approaches, used to evaluate newly suggested static analyses and compare them to prior work. In this part, we suggest new techniques, to address limitations of state-of-the-art

Our discussion in Chapter 3 shows that the design of an existing corpus always targets a specific goal. Thus, their suitability for evaluating arbitrary static analyses is unclear. In many cases, we lack the understanding to which degree the used corpus a) supports the evaluation goals and b) contains all evaluation-relevant properties, such as programming language features, APIs, or design constructs. As most call-graph (CG) evaluations primarily focus on the CG's precision, relevant language features and their influence remain an afterthought. Precision is an essential metric since high numbers of false positives are the foremost reason why developers are hesitant to use static analysis tools [JSMHB13]. However, we are also interested in a CG's sound(i)ness [LSS+15], since our assessment of the state-of-the-art suggests that CGs do not support relevant programming language features and APIs. We argue that it is vital to understand CG algorithms' capabilities and trade-offs to decide their suitability for a given project or subsequent analysis. Furthermore, we are interested in how these CG algorithms compare across different state-of-the-art static analysis frameworks. Therefore, in this part of the thesis, we focus on a) finding suitable evaluation and test programs and b) assessing the capabilities of call-graph algorithms.

To address the identified challenges, we present three related building blocks to support developers and users of static analyses in performing a systematic comparative assessment of algorithms for call-graph construction. Figure 3.1 gives an overview of the tools and shortly depicts their relations.

To facilitate comprehension of existing evaluation corpora and ease the construction of new ones, we propose Hermes (cf. Chapter 4). Hermes is an extensible and generic query engine for code that enables the systematic assessment of given corpora and creation new corpora. Based on an extensible set of queries, Hermes provides a comprehensive overview of a project's features and the features' locations, whose understanding is critical for many analysis projects. Depending on the evaluated queries' results, Hermes can then compute the minimal set of those projects that are necessary to cover all relevant features. Using this minimal set, one can test and evaluate analyses efficiently.

In Chapter 5, we discuss the design of Cats, our comprehensive Call-graph Assessment and Test Suite. Based on a set of hand-crafted tests that cover call-graph-construction-relevant features, Cats enables us to compute a fingerprint of the unsoundness of call-

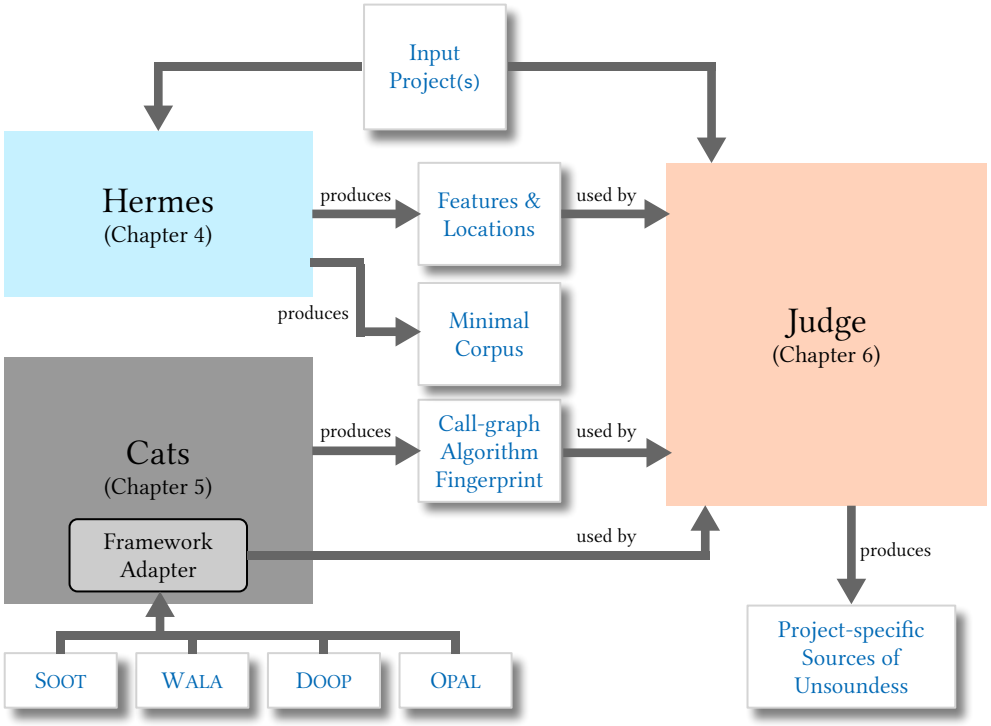

Figure 3.1.: Overview of all tools and their relation that are presented in this chapter.

graph algorithms. Developers can use Cats as a regression test suite or as a reference for new implementations of call-graph algorithms. Furthermore, it empowers us to compare static analysis frameworks. We employ Cats' fingerprinting for several call-graph algorithms from Soot, Wala, Doop, and Opal and, hence, document their capabilities concerning their supported programming language features and APIs.

In Chapter 6, we propose Judge, our overarching toolchain for analyzing call-graph algorithms concerning the language features they cover in a project-specific manner. Judge builds on top of Hermes and Cats' fingerprints to find and document a CG's sources of unsoundness within an assessed program. In several experiments, we use Judge to:

a) determine the prevalence of language features and APIs that affect soundness in modern Java Bytecode,

b) compare the CGs of Soot, Wala, Doop, and Opal, highlighting essential differences in their implementations, and

c) evaluate the necessary effort to achieve *reasonable* soundness of CGs in a project-specific way.

4. Hermes: Assessment and Creation of Effective Test Corpora

This chapter presents our work to assess a set of programs, with the following specific contributions:

- Hermes, an open-source framework[1] for the assessment of a given corpus of Java projects and for the computation of a minimal corpus regarding the evaluated features
- An initial set of feature queries to collect, provide, and comprehend information about a project
- An evaluation that shows the usefulness of the the assessment and creation of test corpora
- A case study that shows that the Qualitas Corpus [TAD+10] is not suitable for any kind of evaluation and should, e.g., not be used to evaluate the level of support for Java 7 or newer features

We will discuss the proposed approach and its realization in Section 4.1. Next, we present in Section 4.2 our evaluation and case study. The chapter ends with a conclusion in Section 4.3.

4.1. Design and Implementation

Hermes is an extensible, configurable framework for the comprehensive assessment of a given set of projects concerning a wide range of different features. We implement Hermes upon the Java bytecode analysis framework OPAL [EH14, HKR+20] and, therefore, require compiled Java programs as input. The latter can be either standalone applications or libraries; all projects form the *base corpus*.

Figure 4.1 gives an overview of Hermes' design. Taken the base corpus, Hermes assesses these programs via querying them for the occurrence of features. Whereas a *feature* is an abstract concept that can express a call to an API, a bytecode instruction, or a complexity metric, a *feature extension* is a concrete value of a feature, such as a call to `Class.newInstance()`, an occurrence of the *invokedynamic* instruction, or the computed value of a complexity metric. A respective *feature query* then determines the extensions of a feature for a given project: It is a static analysis that given a project as

[1] `https://www.opal-project.de/Hermes.html` (checked on Sept 01, 2018).

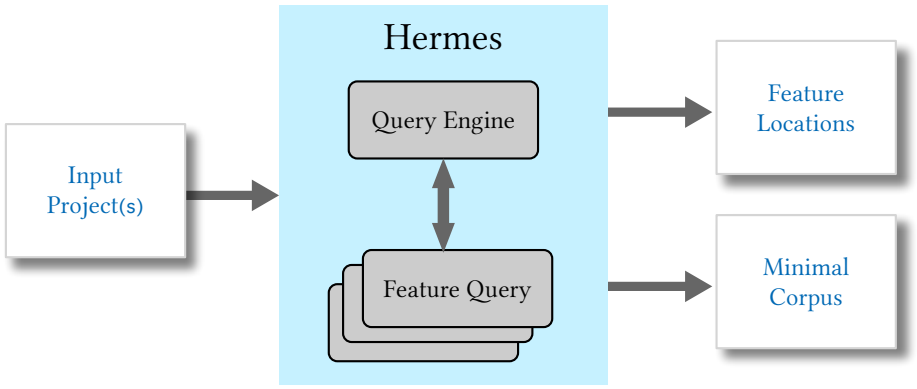

Figure 4.1.: General design overview of HERMES, showing its inputs and outputs.

input collects all *feature extensions*. After collecting the desired information, HERMES produces a report containing the found features and their locations within the program. Depending on the feature, the location information is expressed at package, class, method, or instruction level. The set of feature queries is configurable and customizable.

Using the results of the evaluation of all queries, HERMES can then automatically compute a *minimal corpus* which ensures that all features are found in at least one project. This subset can then be used for effective and efficient testing and evaluation purposes.

4.1.1. Approach

HERMES is written in Scala and uses multiple representations of Java bytecode from OPAL which enables lowest level queries but also feature queries at a high abstraction level. Additionally, OPAL provides useful abstractions such as a `Project` and also provides a wide range of standard functionalities like computing control-flow graphs and call graphs. This facilitates the implementation of feature queries which range from metrics to data- and control-flow dependent analyses. The computation of the optimal corpus is done using the constraint programming library *Choco* [PFL16].

In the following, we describe the main components of HERMES along with the steps a user must take to assess and optimize a *base corpus*.

Corpus configuration Before running HERMES, all projects of the *base corpus* have to be specified. Listing 4.1 shows an example configuration for a small corpus consisting of two projects. Each project specification consists of a unique *id* (line 6 and 12) and a specification of its classpath (*cp* line 7 and 13). Additionally, the two optional attributes *libcp* (line 8) and *libcp_default* (line 9) can be used to specify the project's libraries. The first one specifies the paths to the libraries' jar files and *libcp_default* is used to add a dependency to a predefined library to the project, e.g., a language runtime. The available default libraries are the *current* Java Runtime Environment (JRE) as a whole

```
1  {
2      "org": {
3          "opalj": {
4              "hermes": {
5                  "projects": [{
6                         "id": "Apache ANT 1.7.1 - Javac 6",
7                         "cp": "../../projects/Apache ANT 1.7.1.jar",
8                         "libcp": "../../dependencies/Apache ANT
                             1.7.1.jar",
9                         "libcp_default": "JRE"
10                        },
11                        {
12                            "id": "argouml-excerpt",
13                            "cp": "../../projects/argouml-excerpt.jar"
14                        }]
15             }
16         }
17     }
18 }
```

Listing 4.1.: Example JSON configuration file (.json) specifying the corpus projects.

or just the `rt.jar`[2]. Library class paths need to be specified whenever some feature query requires information that cannot be extracted from the project alone, e.g., feature queries related to the inheritance hierarchy generally require a complete type hierarchy.

Query configuration HERMES provides a customizable configuration where the queries which should be evaluated are configured. By default, all available queries will be evaluated. The set of queries should—in general—be selected with a concrete analysis and test/evaluation goal in mind. For example, if a test corpus for integration testing of a static analysis should be created, it might be important to ensure that *all* language-specific features are found at least once in the given projects. If the evaluation goal is the scalability of the analysis, it may be more important to ensure that specific features occur with a certain frequency. However, it is possible to always run all available queries, but as every query is also a potentially complex static analysis, this might be too expensive. Listing 4.2 shows a configuration that enables the queries in Line 3, 5, and 7 and which disables the query in Line 4. Each entry specifies the fully qualified name of the class that implements the query (*query*) and whether the given query should be executed or not (*activate*). New queries can simply be added to the configuration analogously.

[2]Here, *current* refers to the one used for running HERMES.

```
1 {
2     "org.opalj.hermes.queries" = [
3     { "query" : "queries.Metrics", "activate" : true }
4     { "query" : "queries.MethodsWithoutReturns", "activate" : true}
5     { "query" : "queries.JDBCAPIUsage", "activate" : false }
6     ...
7     { "query" : "queries.MethodTypes", "activate" : true } ]
8
9 }
```

Listing 4.2.: Hermes' configuration of enabled and disabled feature queries.

Corpus evaluation and visualization Given a complete configuration, we can then start Hermes. Hermes' UI provides an overview of the current state of the evaluation, provides descriptions of the activated queries, and shows basic size metrics related to the projects.

Additionally, the evaluation of each activated feature query for each project belonging to the specified base corpus is directly started. As soon as a feature query was evaluated, Hermes shows the resulting number of feature occurrences and makes it possible to jump to concrete occurrences of the feature in the respective project's code base—if supported by the query. In general, a query can report feature occurrences at the package, class, method, or instruction level. Being able to navigate to concrete feature occurrences is helpful when developing new feature queries, but also if a more detailed understanding of the feature in the context of a specific project is required. The amount of location information that is kept is configurable and managed by Hermes to ensure that very large test corpora such as the Qualitas Corpus can successfully be evaluated.

4.1.2. Feature Queries

A feature query is a static analysis that is given a project as input and then collects all feature extensions of one or multiple closely related features. For example, it is possible to write a query which collects all Java 7 class files found in a specific project and another one for Java 8 class files. Alternatively—and also more efficiently—it is possible to write a single feature query that analyzes every class file once and adds every class file to its respective feature category. To ensure that all features are uniquely identifiable across all feature queries, each query assigns a unique id to each derived feature.

All feature queries have to implement the `FeatureQuery` interface, which defines the two functions shown in Listing 4.3. The first function `featureIDs` (Line 3) defines a list of unique feature ids where each id represents the name of a derived feature. The second function (`apply` - Line 5) defines the query itself. The input for the static analysis is the project configuration (Line 6), OPAL's representation of a `Project` (Line 7), and a raw one-to-one representation of the project's Java class files (Line 8). The raw representation supports queries which need to work on unprocessed class files; e.g., those

```scala
trait FeatureQuery {

  def featureIDs: Seq[String]

  def apply[S](
    projectConfiguration: ProjectConfiguration,
    project: Project[S],
    rawClassFiles: Traversable[(da.ClassFile, S)]
  ) : TraversableOnce[Feature[S]]
}
```

Listing 4.3.: Scala trait that must be implemented by all feature queries.

that want to analyze the constant pool in detail. The representation provided by the project enables higher level code analyses, such as control- and data-flow analyses or abstract code interpretation[3].

A selection of the currently available feature queries is listed in Table 4.1 together with the number of derived feature extensions and a short description of each feature. The available queries demonstrate the variety of possible analyses: they reach from basic *API usage* queries, which can be used to select projects for API misuse detection, specification mining, or injection analyses, over *JVM* and *language features* based queries—e.g., to find suitable integration test corpora—up to *control-* and *data-flow* analyses. The latter can, e.g., be used to get some understanding of how Java reflection is used.

4.1.3. Computing an Optimal Corpus

After all queries have been evaluated for all projects, it is possible to let Hermes compute the subset of projects which has the overall minimal number of methods (*optimization goal*) and which ensures that every feature occurs at least once in some project (*constraint*). I.e., Hermes would prefer two small projects with, e.g., two methods each over one project with ten methods. Minimization of the overall number of methods is done because in most cases it better reflects the overall effort that is necessary when the corpus is eventually used for evaluation or test purposes.

For more elaborated use cases, it is possible to export the computed results using a CSV file and to perform some external post processing, e.g., to study a particular feature in-the-wild.

4.2. Evaluation

In the following, we describe the evaluation of Hermes for the two use cases: "*Comprehending a test corpus*" and "*Generation of an effective integration test suite*".

[3] An example for an API query can be found in Appendix A, an example for a query computing metrics can be found in Appendix B, and an example for a custom query is available in Appendix C.

Table 4.1.: Available feature queries including their category, number of unique features and a short description.

feature query	**category**	**# features**	**description**
BasicMetrics	metrics, control flow	15	Extracts the following basic metrics: methods per class, fields per class, the number of children (NOC), and McCabe and groups them per complexity category (e.g., in case of McCabe: linear methods, simple methods (2 to 3 paths), complex methods (more than 3 paths).
BytecodeInstructions	JVM features	201	List of all Java bytecode instructions as defined in the Java Virtual Machine Specification (Java 1.1 up to Java 8).
ClassFileVersion	JVM features	6	Extracts the class file version (Java 1.1 up to Java 9) of each class file belonging to the project where each version is a single feature.
ClassTypes	language features	10	Extracts the information about the type of the specified class; e.g., how may concrete classes, annotations, interfaces, interfaces with default methods, or Java 9 modules are defined.
JavaCryptoArchitecture-Usage	API usage	8	Extracts information about the usage of core classes and interfaces, for instance ciphers, keys, or signatures, from the Java Crypto Architecture (JCA) according to the official reference guide.
JDBCAPIUsage	API usage	5	Extracts information about the usage of Java's JDBC API and SQL statement kinds.
MethodsWithoutReturns	control flow	2	Extracts whether a method either never returns normally, e.g., by throwing an exception, or has a real infinite loop without any possibility to return.
MethodTypes	language features	9	Extracts the information about the type of the specified methods; e.g., whether a method is native, synchronized, or is a varargs method.
ReflectionAPIUsage	API usage	12	Derives which methods/functionality of Java's classical Reflection API is used within a project.
SystemAPIUsage	API usage, capabilities	8	Extracts the usage of API methods that are related to the state of the JVM, capabilities [HREM15], or used to access the underlying operating system; e.g., spawning an external process, playing sound, or working with the `java.lang.SecurityManager`.
TrivialReflectionUsage	API usage, data flow	1	Counts the number of cases where `Class.forName` calls can be trivially resolved, because the respective String(s) are directly available.
UnsafeAPIUsage	API usage	19	Derives usage information about `sun.misc.Unsafe` according to the classification of Mastrangelo et al. [MPM+15].

All measurements were done on a Mac Pro with a Xeon E5 CPU with 8 cores@3GHz. The Java Virtual Machine (Java 8 Update 121) was given 24GB of heap memory.

4.2.1. Comprehending Test Corpora

To understand the nature of the projects contained in the latest release of the Qualitas Corpus [TAD+10] (QC) from September 2013, we ran Hermes using all queries against all projects and inspected the result. As expected—given the release date of QC—none of the projects used Java 8. More surprisingly, none of the projects used the JavaFX framework already introduced in 2008. This indicates that even though the corpus already contains over 100 projects, some domains are not well represented. Furthermore, only one (Hibernate) of the included projects used Java 7 features[4]. Overall, this preliminary analysis suggests that using the Qualitas Corpus to evaluate or test analyses that support Java features released after 2011 is not meaningful.

4.2.2. Generating Integration Test Suites

For the second evaluation, we used Hermes to compute an optimal test corpus based on the Qualitas Corpus [TAD+10] (QC) for generic integration testing purposes; i.e., we used Hermes to compute the subset of all QC projects that should enable us to perform effective and efficient integration testing of general static and dynamic analyses. The concrete goal for the evaluation was to use the minimal set of projects for testing the analysis described in the paper "Hidden Truths in Dead Software Paths" [EHMG15]. The core part of that analysis is a very generic data- and control-flow analysis and it should be able to handle all valid Java bytecode. Using this minimal set of projects should give us basically the same level of confidence in our developed analysis as using all QC projects.

The first step of this evaluation was to run Hermes against all projects using every available query. After each query was evaluated for all projects, we let Hermes compute the minimal set of projects which a) has the minimal overall number of methods and b) ensures that every feature occurs at least once in some project[5]. The set of projects computed by Hermes consists of the following five projects: `joggplayer`, `jchempaint`, `hibernate`, `quilt`, and `nakedobjects`. It took Hermes less than a second to find the minimal solution.

The second step was to determine the overall code coverage of the paper's core control- and data-flow analysis. We measured the coverage using *scoverage*[6] twice: Once, running the analysis against all 100 projects of the Qualitas Corpus and once running it only against the automatically determined set of five projects. The time to run the analysis against all projects was $1006s(\approx 16.77min)$ while it took $169s(\approx 2.82min)$ for the selected projects, i.e., just using the selected projects is nearly 6 times faster.

[4] Java 7 was released in 2011 and already two years old when the updated corpus was created.

[5] Recall that features which are not found at all across all given projects, such as those related to Java 8 features in case of the QC, are simply ignored.

[6] `https://github.com/scoverage` (checked on Dec 15, 2018).

However, the code coverage is slightly better (1.06%) when we all QC projects. An investigation of the code coverage data revealed that some projects, which did not belong to the selected projects, contained advanced exception handling and more elaborate array-based accesses.

4.2.3. Discussion

To recap, the evaluation shows that one can use HERMES to get a better understanding of available corpora and also to compute minimal test corpora that enable effective integration testing. Furthermore, given the very primitive nature of the available queries and the achieved quality of the results, it is evident that we do not need complex queries to compute effective test corpora.

Additionally, by adding further queries related to exceptions/exception handling and to array accesses, one can compute a test corpus that is most likely still much smaller than the complete QC, while being as effective for testing data- and control-flow analyses.

4.3. Conclusion

Testing and evaluation are essential and generally very time consuming tasks that are part of the development of every new analysis. Both tasks are typically done using test corpora consisting of large(r) collections of projects. But as discussed in Chapter 3, without explicit tool support, it is hard to judge whether the selected projects have the desired/necessary features. Additionally, it is impossible to know which projects are useful and which just test/evaluate the same functionality over and over again.

To address these issues, we proposed HERMES, a generic framework that facilitates the assessment of a given set of Java bytecode projects with respect to their properties. Thereby, we gave researchers a tool to a) assess whether a given corpus suits their needs and b) enable meaningful evaluations.

Furthermore, HERMES contains several built-in feature queries, allowing users to explore various properties of projects. These are a starting point for selecting projects for different static analyses; e.g., for SQL injections, cryptographic security flaws, or call graph construction. We demonstrated HERMES' usefulness by using it to better understand the Qualitas Corpus and by computing a minimal test corpus useful for integration testing of generic data- and control-flow analyses. Thus, we successfully addressed *Chlg. 7*, identified in Chapter 3.

5. CATS: A Framework for Systematically Testing the Unsoundness of Call-graph Construction Algorithms

Following the idea of test suites for static analyses, such as SecuriBench [Liv18] and DroidBench [ARF+14], we facilitate the task of assessing call-graph (CG) algorithms with an automated test framework. We call this framework Cats. Its goals are:

a) to provide a comprehensive overview of programming language features and APIs relevant to CG construction,

b) to automate the assessment of a CG algorithm's capabilities, and

c) to enable reproducible and extensible experiments.

It is possible to add new CG algorithms to compare and extend the test suite in the future. We publish Cats for future studies[1].

5.1. Design and Implementation

The core idea is to have a wide range of small, focused test cases. Each test case—as far as possible—tests a single relevant soundness aspect related to CG construction. These test cases provide the *ground truth* and, when compiled, are used as input for the different CG algorithms.

Figure 5.1 provides an overview of the proposed approach. For each set of closely related test cases we use a single markdown file which contains all related tests (<*Test Fixtures Category*>*.md*). For example, we create one markdown file for each of the following categories: usages of Java *Reflection*, Java 8 language features, usages of *sun.misc.Unsafe*, or *Serialization*. Using markdowns enables us to generate a concise, human-readable description of the test cases that can be enriched with additional background information. Each test case consists of a small runnable Java program which uses a specific language feature and/or API along with a brief description of the unique features of the test case. Additionally, each test case contains one or more annotations to describe the expected call targets; i.e., to specify the *ground truth*.

The *Test-case Extractor* parses the markdown files and retrieves the test cases, compiles them, and bundles each one into a respective *jar* file. In addition to the test cases that are compiled by the test-case extractor, we can provide a number of precompiled

[1] `https://bitbucket.org/delors/cats/src/master/` (checked on Dec 2, 2020).

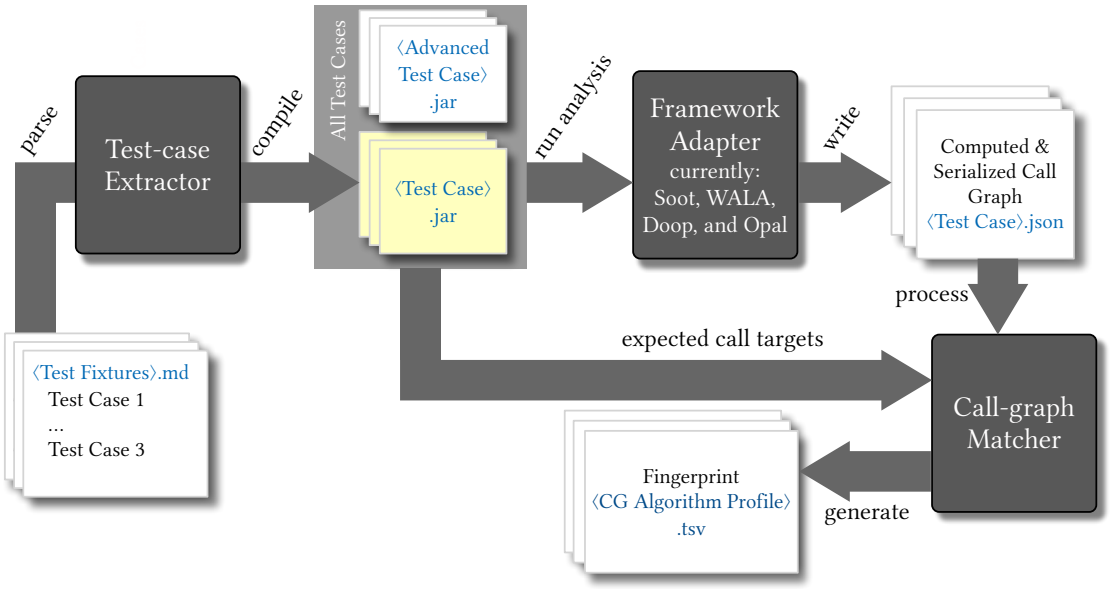

Figure 5.1.: Overview of the CG test framework: From test-case specification and compilation, over evaluation to fingerprint generation.

test cases. We use this feature for more advanced test cases that cannot be created within a single compilation process, e.g., when imitating a software evolution scenario.

After that, we use a *Framework-specific Test Adapter* to construct a call graph (CG) for each individual CG algorithm. After construction, the graph is serialized to a common JSON-based representation. The last step is performed by the *Call-graph Matcher*. It loads the CG and compares the found call targets with those explicitly specified in the test cases. Based on its findings, the CG matcher then generates a *fingerprint* of the algorithm's capabilities, i.e., a report that summarizes the results. In the following, we provide more details regarding the individual steps.

Test-case Specification There are two classes of test cases. The first class consists of basic test cases that can be created using Java code. These are defined in markdown files (.md) that contain a high-level description of the test case along with the source code. Listing 5.1 shows an example markdown file. Each file is structured in the same way: The first level header (e.g., *Trivial Reflection* in Line 1) identifies the test category. A second level header (e.g., `TR1` in Line 3) identifies a concrete test case. After the second level header comes the specification of the main class (Line 4) and a short description (Line 5) that is followed by multiple code snippets which—taken together—form an executable Java program. The first line of each test case is a Java comment that identifies the target file in which the code will be stored (Line 7). As a test case can consist of multiple *public* Java classes, it is not always possible to declare one test case in a single Java file. Listing 5.1's *TR1* test case will be stored in the file *tr1/Foo.java*.

The second class of test cases consists of test cases that cannot be generated by the

````
1  #TrivialReflection
2  The strings are directly available...
3  ##TR1
4  [//]: # (MAIN: tr1.Foo)
5  Test reflection with respect to static methods.
6  ```java
7  // tr1/Foo.java
8  package tr1;
9  import lib.annotations.callgraph.IndirectCall;
10 class Foo {
11     static String m() { return "Foo"; }
12
13     @IndirectCall(
14         name = "m", returnType = String.class,
15         line = 17, resolvedTargets = "Ltr1/Foo;" )
16     public static void main(String[] args) throws
           Exception {
17         Foo.class.getDeclaredMethod("m").invoke(null);
18 }}
19 ```
20 [//]: # (END)
21 ##TR2 ...
````

Listing 5.1.: An example test-case definition from the *Trivial Reflection* category.

Java 8 compiler. These *Advanced Test Cases* (cf. Figure 5.1) are manually compiled using another compiler (e.g. Java 10 or Scala), created via bytecode engineering, or by replaying code evolution scenarios. The study of the Java Virtual Machine Specification (JVMSpec) led us to test cases that represent valid bytecode but cannot be generated by the Java compiler. For example, the JVM supports so-called `MethodHandle Constants` which are primarily intended to be used by other JVM-hosted languages. Furthermore, due to code evolution, it may happen that an interface `SuperI` defines a default instance method `m` and its subinterface `SubI` a corresponding static method `m`. That is, both methods have the exact same signature and only differ in the access modifier (e.g., public or protected). Such bytecode is legal and works reliably, but cannot be created using Java source code.

Annotating the Ground Truth In order to detect missing call edges, a specification of the ground truth is required. We decided to use Java's annotations (cf. Line 13 in Listing 5.1) to specify the crucial CG edges that should be part of the CG. Due to the decision that all code snippets have to be executable programs, it is sometimes necessary to perform multiple calls to achieve the required state. Hence, each method may contain multiple call sites. Therefore, we identify the relevant call sites using line numbers, the callee's name, as well as its return and parameter types. To avoid ambiguous call sites, the test are restricted to have only one method call with the same name per line of code.

```
{ "callSites": [
    { "method": {
           "name": "main",
           "parameterTypes": ["[Ljava/lang/String;"],
           "returnType": "V",
           "declaringClass": "Ltr1/Foo;" },
       "line": 12,
       "declaredTarget": {
           "name": "getDeclaredMethod",
           "parameterTypes": ["Ljava/lang/String;",
                              "[Ljava/lang/Class;"],
           "returnType": "Ljava/lang/reflect/Method;",
           "declaringClass": "Ljava/lang/Class;" },
      "targets": [ {
          "name": "getDeclaredMethod",
          "parameterTypes": ["Ljava/lang/String;",
                             "[Ljava/lang/Class;"],
          "returnType": "Ljava/lang/reflect/Method;",
          "declaringClass": "Ljava/lang/Class;" } ]
    },
    ...
  ]
}
```

Listing 5.2.: A serialized CG from the *TR1* test case shown in Listing 5.1.

We provide two annotations: First `CallSite` to specify direct call edges between two methods. This one is used for standard virtual method calls, constructor calls, static method invocations, and default method invocations (Java 8). The second one, `IndirectCall`, is used to specify indirect calls. Consider the reflective call *m.invoke(null)* in Line 17 (Listing 5.1). In this case the CG may (also) contain call edges to the *Reflection* API and/or a call to the target method (`m` in the example); however, the representation of such calls is framework specific and to abstract over differences, e.g., how reflective calls, method references, etc. are actually handled by the frameworks, we specify that we expect some path leading to the expected targets as shown in Lines 13-15.

Serialization of the Call Graphs In the JSON representation (cf. Listing 5.2) of the CGs, each method is represented using its name, the parameter types, the return type, and the fully qualified name of its declaring class. A call site is represented by the caller method, the line number, the declared target method, and the set of computed target methods.

Validating the Call Graph Identifying missing call edges is done by iterating over all methods of a project and comparing the found call targets against the specified ones. The presence of call edges related to indirect calls is validated by performing a breadth-first search on the computed CG; starting with the main method. The final report then lists failed test cases along with the missed calls.

Fingerprint Generation To construct the fingerprint of a CG algorithm, CATS uses the final report. Please note that each test case is testing one specific feature that is relevant when constructing CGs. Hence, each test case a CG algorithm passes represents a single feature the algorithm supports. Given the sum of all test cases, we can derive a CG algorithm's fingerprint, representing its capabilities of supported language features, APIs, and bytecode features.

CATS supports the analysis of various CG algorithms offered by the frameworks. Four algorithms from WALA, four algorithms from SOOT, 1 algorithm from DOOP, and one algorithm from OPAL.

5.2. Test Suite

In the following, we discuss our extended test suite by first giving a high-level overview of the test categories (cf. Table 5.1) before we discuss individual test cases. Overall, we define 122 test cases which we grouped in 23 categories.

5.2.1. Test Categories

Classloading Using a *java.lang.ClassLoader*, it is possible to load and use a specific class in multiple (incompatible) versions.

Class.forName Exceptions Loading a class at runtime, using *Class.forName(...)*, can cause various exceptions. If the classloading fails and an exception is thrown, the exception's handler becomes a valid program path. Since ignoring these exception handlers can lead to unsoundness, we cover multiple test cases that *always* lead to an exception.

Dynamic Proxies Java's Dynamic Proxy API creates (via runtime bytecode engineering) type-safe proxy classes which will then forward the calls—using Java reflection—to a previously specified handler class.

Interface Default Methods Java 8 introduced default methods which are defined in interfaces and have to be taken into account when resolving virtual method calls. These default methods act as fallback case when a Java class implements that interface but does not override the respective interface method. We included test cases for virtual method invocations concerning interface default methods and maximally-specific interface methods. The latter must be computed when a subclass/interface inherits multiple interfaces that define a method with the same signature.

Category	Abbreviation	# Test Cases
Classloading	CL	4
Dynamic Proxies	DP	1
Interface Default Methods	J8DIM	6
Static Interface Methods	J8SIM	1
Java 8 invokedynamics	MR/Lambda	11
JVM Calls	JVMC	5
Library Analysis	LIB	5
Trivial Reflection	TR	9
Locally Resolveable Reflection	LRR	3
Context-sensitive Reflection	CSR	4
Method Handles	MH	9
Class.forname Exceptions	CFNE	4
Non-virtual Calls	NVC	6
Serialization	Ser	9
Externalizable	ExtSer	3
Lambda Serialization	LamSer	2
Signature Polymorphic Methods	SPM	7
Static Initializers	SI	8
Types	-	6
Unsafe	-	7
Virtual Calls	VC	4
Java 9/10 Features	J9+	2
Non-Java Bytecode	NJB	6
Total		122

Table 5.1.: Overview of the test suite showing the different categories, their abbreviations, and their number of test cases.

JVM Calls Calls of those methods that are (only) done by the JVM due to some event, such as calling *start* on a Thread. In that case the JVM will eventually call the *Thread*'s *run* method. Other examples comprise object finalizers or access control functionality.

Lambdas and Method References The Java 8 compiler started to use *invokedynamic* instruction to compile lambda expressions (e.g. `() ⟹ doSomething();`) as well as calls that are based on method references (e.g., `String::length`). We included test cases for various different cases of lambdas and method references that result in different bytecode.

Library Analysis As will be discussed in Chapter 7, the target of a method call in a library may require call-by-signature resolution when computing call graphs *just* for the library. Therefore, this category comprises test cases that assume an open-world

scenario.

Virtual Calls Such method calls are at the core of Java. When a virtual method is called, the target is resolved depending on the runtime type of its receiver object. When the runtime type cannot be determined precisely, a sound call-graph algorithm will over-approximate the receiver type and then determine the set of possible call targets. The provided test suite contains various test cases containing polymorphic method calls. Those cases cover class and interface receiver types in the presence of different type hierarchies.

Non-virtual Calls Non-virtual method calls, i.e., constructor calls, *super* calls, *private* method calls and *static* method calls. The call target at a non-virtual callsite is always unambiguous.

Trivial Reflection Usage of the classical reflection API—*java.lang.reflect.** in combination with *java.lang.Class*'s methods—where the call target is immediately available (e.g., `Class.forName("XYZ")`). Hence, test cases that belong to this category are rather trivially resolvable, as all API inputs are directly known and neither data-flow nor control-flow analyses are required.

Locally-resolvable Reflection Usage of the classical reflection API where an intraprocedural control-/data-flow analysis is required to resolve the call targets. Hence, the information passed to the reflection API is defined within the reflection-using method.

Context-sensitive Reflection Usage of the classical reflection API where an interprocedural control-/data-flow analysis is required to resolve the call targets. Hence, the information passed to the reflection API is defined outside of the reflection-using method.

Method Handles Usage of the modern reflection API—*java.lang.invoke.** and Java 7's *MethodHandle* API. This category tests only methods from the *MethodHandle* API that are not signature polymorphic, i.e., the method descriptor used at the call site must always match the signature of the called method. In contrast to the different levels of classical reflection test cases, tests defined in this category have all inputs immediately available. They do not require any control- or data-flow analysis.

Signature Polymorphic Methods Calls from this category comprise signature polymorphic method[2] calls concerning from Java's *java.lang.MethodHandle*'s *invoke* and *invoke-exact* methods, or *java.lang.invoke.VarHandle* respectively. In general, a method is considered signature polymorphic if it has the following three properties: the method a) is either declared within *MethodHandle* or *VarHandle*, b) has a single formal parameter of type *Object*[], and c) has the *ACC_VARARGS* and *ACC_NATIVE* flags set.

[2] More information pertaining to signature polymorphic methods can be found within the Specification of the Java Virtual Machine [GJS+18b] in §2.9.3

Method calls of this category are unique because the invoked method's signature can differ from the invoked method when the method handle invocation happens through *MethodHandle*'s *invoke* method. Therefore, special semantic applies to those method calls. For instance, passed parameters are (un)boxed, cast, or widened automatically. Please note, those automated operations are not performed when *invokeExcact* is called.

Serialization Java's serialization mechanism allows one to persistently store and retrieve objects using object serialization. To use this mechanism, classes must implement the *java.io.Serializable* or *java.io.Externalizable* interface. Additionally, classes can define several callback methods, e.g., *writeObject* or *readResolve*, that are called by the JVM during (de-)serialization. Hence, those method calls are implicitly performed by the JVM and not visible to the programmer. For example, when deserialization of a *Serializable* object occurs, the default constructor[3] of the first non-serializable superclass is invoked. This constructor must exist and must be accessible from the class. Likewise, *Externalizable* classes also must define a default constructor which is then invoked during deserialization.

Static Initializers In Java, every time a class is loaded by the JVM, a call to its static initializer[4] is performed by the language runtime. Those calls are implicit and, therefore, must be explicitly modeled by the CG algorithm.

Types Type casts and *instanceof* checks can be performed using language features, but also using core Java APIs. We added several test cases that test both: API-based and language-feature-based type casts and *instanceof* checks. Type narrowing itself is not directly related to soundness but still a language feature that needs coverage. Considering a static analysis on Java bytecode, e.g., with Soot, type casts and *instanceof* checks require handling for additional bytecode instructions.

Unsafe With *sun.misc.Unsafe*, Java provides an internal API that allows direct memory manipulations via Java code. Using the methods *compareAndSwapObject*, *putObject*, or *getObject*, objects can be put into or retrieved from fields. The test cases therefore test whether the call graphs contain call edges to those virtual methods that are due to an unsafe field update. E.g., if a method m invocation occurs on a field of type T which is updated to an object of type $TSub$ (with $TSub extends T$) via *Unsafe*, the call graph must contain an edge to $TSub.m$. Despite that the API is designed for internal use only, security checks to retrieve an instance of the class can be circumvented, e.g., by using Java's Reflection API. In fact, this API is used by several wide spread libraries [MPM+15].

[3] A default constructor is a constructor without any formal parameters.

[4] More information pertaining to static initializers can be found within the Specification of the Java Virtual Machine [GJS+18b] in §12.4.1.

Java 9/10 Features added with Java 9 and 10, such as *private* interface methods and modules.

Non-Java Bytecode Legal Java Virtual Machine (JVM) bytecode that cannot be created using Java. This bytecode can either be created through software evaluation scenarios that require multiple compilation steps or by compilers for other JVM-hosted languages such as Clojure, Groovy, Kotlin, or Scala. These languages can theoretically be analyzed with a static bytecode analysis tool. However, the emitted bytecode and the usage of JVM features can differ among languages [SMSB11, SMS+12], e.g., Groovy and Scala use *invokedynamic* instructions differently than Java and may require special handling. Please note that test cases in this category require a manual compilation process.

5.2.2. Test Case Design

For systematically designing the test suite, we studied the Java Virtual Machine Specification (JVMSpec) [GJS+18b] and Java's core APIs (*java.**). When constructing the test cases, we tried to ensure that a test case will only succeed if the algorithm explicitly supports the respective feature. This is, however, not possible in all cases; some test cases are simply supported due to an algorithm's inherent imprecision. For example, some of the test cases related to *Types* or *Unsafe* manipulate references and can therefore negatively affect soundness in those algorithms that are points-to information based. If those algorithms do not model the effects of, e.g., manipulating references using the Unsafe API, the points-to information will be incorrect—potentially leading to unsound results. CG algorithms, such as class-hierarchy analysis (CHA), that just rely on the type information found in the bytecode handle related scenarios in a sound manner; they just assume all subclasses. Furthermore, we did not add explicit test cases related to custom native methods because none of the frameworks support cross-language analyses.

5.3. Using Cats to Study the Call-graph Algorithms of State-of-the-art Static Analysis Frameworks

In the following, we describe how we evaluate Soot, Wala, Doop, and Opal's call-graph (CG) algorithms by applying the proposed test suite. The study is driven by the following two research questions.

RQ1 How do the CGs of Soot, Wala, Doop, and Opal compare to each other in terms of feature support?

RQ2 What are the main sources of unsoundness in built-in CG algorithms of Soot, Wala, Doop, and Opal?

Table 5.2.: Support of language features and APIs of Soot, Wala, Opal, and Doop's call graphs.

Category	**Soot**				**WALA**				**OPAL**	**DOOP**
	CHA	RTA	VTA	SPARK	RTA	0-CFA	N-CFA	0-1-CFA	RTA	Points-to
CL	◐ 4/6	◐ 4/6	◐ 4/6	◐ 3/6	◐ 4/6	◐ 4/6	◐ 2/6	◐ 4/6	◐ 4/6	◐ 4/6
DP	● 1/1	● 1/1	⊖ 0/1	⊖ 0/1	● 1/1	⊖ 0/1	⊖ 0/1	⊖ 0/1	● 1/1	⊖ 0/1
J8DIM/J8SIM	◐ 3/7	◐ 3/7	◐ 3/7	◐ 3/7	● 7/7	● 7/7	● 7/7	● 7/7	● 7/7	◐ 3/7
MR/Lambdas	◐ 1/11	◐ 1/11	⊖ 0/11	⊖ 0/11	● 11/11	◐ 10/11	◐ 10/11	◐ 10/11	● 11/11	◐ 1/11
JVMC	◐ 4/5	◐ 4/5	◐ 3/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5
LIB	◐ 2/5	◐ 2/5	◐ 2/5	◐ 2/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 1/5	◐ 2/5	⊖ 0/5
TR	◐ 4/9	◐ 4/9	◐ 4/9	◐ 4/9	◐ 3/9	◐ 6/9	⊖ 0/9	◐ 6/9	● 9/9	◐ 3/9
LRR	● 3/3	● 3/3	● 3/3	● 3/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	◐ 1/3	◐ 2/3
CSR	● 4/4	● 4/4	● 4/4	● 4/4	⊖ 0/4	⊖ 0/4	⊖ 0/4	⊖ 0/4	◐ 1/4	◐ 0/4
MH	◐ 3/9	◐ 3/9	◐ 1/9	⊖ 0/9	◐ 2/9	⊖ 0/9	⊖ 0/9	⊖ 0/9	● 9/9	◐ 1/9
CFNE	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	◐ 3/4	● 4/4	● 4/4	● 4/4
NVM	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
Ser	◐ 1/9	◐ 1/9	⊖ 0/9	⊖ 0/9	⊖ 0/9	⊖ 0/9	⊖ 0/9	⊖ 0/9	◐ 5/9	⊖ 0/9
ExtSer	● 3/3	● 3/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	◐ 1/3	● 3/3	◐ 1/3
LamSer	◐ 1/2	◐ 1/2	⊖ 0/2	⊖ 0/2	⊖ 0/2	⊖ 0/2	⊖ 0/2	⊖ 0/2	◐ 1/2	⊖ 0/2
SPM	⊖ 0/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	● 7/7	⊖ 0/7
SI	● 8/8	● 8/8	● 8/8	◐ 7/8	◐ 7/8	◐ 6/8	◐ 6/8	◐ 6/8	● 8/8	◐ 7/8
Types	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	● 6/6	◐ 2/6	● 6/6	● 6/6	● 6/6
Unsafe	● 7/7	● 7/7	⊖ 0/7	⊖ 0/7	● 7/7	⊖ 0/7	⊖ 0/7	⊖ 0/7	● 7/7	◐ 0/7
VC	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4	● 4/4
J9+	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	⊖ 0/3	◐ 2/3	⊖ 0/3
NJB	⊖ 0/6	⊖ 0/6	⊖ 0/6	⊖ 0/6	◐ 3/6	◐ 3/6	◐ 3/6	◐ 3/6	◐ 4/6	⊖ 0/6
sum	67/122	67/122	51/122	47/122	67/122	65/122	55/122	58/122	102/122	42/122

● indicates fully supported feature categories, ◐ indicates partially supported feature categories, and ⊖ indicates not supported feature categories.

5.3.1. Setup

All measurements are done using WALA 1.5.0, SOOT 3.1.0, OPAL's develop branch[5], and DOOP's master branch[6]. From WALA, we use the following algorithms: WALA_{RTA}, $\text{WALA}_{\text{0-CFA}}$, $\text{WALA}_{\text{N-CFA}}$[7], $\text{WALA}_{\text{0-1-CFA}}$—all configured with the `FULL` reflection option. WALA requires to specify packages to be excluded from the analysis. However, for this experiment we choose to exclude no package. For all SOOT CGs (SOOT_{CHA}, SOOT_{RTA}, SOOT_{VTA}, and $\text{SOOT}_{\text{SPARK}}$ [LH03]) we use the options *safe-forname* and *safe-newinstance*. These options make SOOT consider all types as instantiated when `Class.forName` or `Class.newInstance` is used. We could not use *types-for-invoke* due to exceptions being thrown [SDE+18]. Furthermore, we use *include-all* to ensure that no packages are filtered. Our library test cases are additionally started with *library:signature-resolution* and *all-reachable* to make use of SOOT's capabilities to analyze library code. DOOP's CG is set to be *context-insensitive* with *classical-reflection* turned on. For OPAL_{RTA}, we use the standard configuration. Please note that we described the used algorithms in Section 2.1.

All test cases with respect to libraries are started with the respective library entry points.

5.3.2. Comparing Call-graph Algorithm Fingerprints

Next, we will compute and compare each algorithm's fingerprint. Table 5.2 summarizes the computed algorithm profiles. The first column shows the test categories. Columns two to ten show for each test category the individual test results per CG algorithm. A cell's symbol indicates whether all (●), some (◐), or none (⊖) of the tests succeeded; the numbers represent the number of succeeded vs. all tests.

The results shows that basic language features like static initializers (*SI*), (non-)virtual calls (*(N)VC*), and type casts (*Types*) are well supported. An exception are two static initializer cases: the first one is not supported by $\text{SOOT}_{\text{SPARK}}$, DOOP, and WALA and the second one is not supported in WALA. *SI4* models a case where a Java 8's interface's static initializer must be called, i.e., the JVM initializes a subclass of an interface that defines a default method. An unexpected behavior is shown by $\text{WALA}_{\text{N-CFA}}$. It can only handle type casts that are performed using Java's explicit `cast` and `instanceof` APIs, but does not support built-in operators, i.e., `instanceof` or type casts of the form `(String) o;`.

Serialization-related methods (*Ser*) are not well supported by WALA and DOOP, they are slightly better supported by SOOT and are best supported by OPAL ($\approx 50\%$). The methods (in particular `readObject` and `writeObject`—which will be called by the JVM) must be considered when object (de-)serialization occurs in reachable methods.

Java 8 language features, such as default methods (*J8DIM*), lambdas, and method references (*MR*) are mostly handled correctly by WALA and OPAL but not supported

[5] OPAL's commit id: 3107c45c8a00de0e132691a6275d39b5a4aa415b.
[6] DOOP's commit id: cdc59ce71d6510198da396cf6a7d20d73c6466d9.
[7] We use N=1 throughout the whole evaluation.

by Soot and Doop. Furthermore, Opal is the only framework that supports the new method handle API (*MH*) and signature polymorphic methods (*SPM*).

Furthermore, support for Java's reflection API varies, but all—except of $\text{Wala}_{\text{N-CFA}}$—provide at least some support. Moreover, Soot's reflection options enable it to resolve all advanced reflection test cases (*LRR* and *CSR*); calls to `Class.newInstance` are resolved to all initializers in the project.

Table 5.2 shows that only the basic algorithms: Soot_{CHA}, Soot_{RTA}, Wala_{RTA}, and Opal_{RTA} can deal with Java's Unsafe API as well as the Dynamic Proxy API. However, more advanced CG algorithms are not able to detect those cases. Here, the imprecision of CHA/RTA benefits the support of those two APIs.

Only Opal supports non-Java bytecode (*NJB*) and Java 9/10 features (*J9+*).

Please note that the $\text{Wala}_{\text{N-CFA}}$ implementation performs consistently worse than Wala's other implementations. After corresponding with the maintainers of Wala they confirmed that his behavior was caused by a bug.

RQ1 – How do call-graph algorithms from Soot, Wala, Doop, and Opal compare? Generally, all CG algorithms provide similar support for basic language features. Differences in the supported features show when we look at specific APIs. Whereas Serialization is best supported by Opal, Soot provides the best support for Java's classical reflection API. Furthermore, Wala und Opal provide the best language support for Java 8, which is still missing in Soot and Doop. While reflection support can also be mixed in by using the dynamic reflection analysis tool *Tamiflex* [BSS+11], there is no alternative for the other features.

In summary, all frameworks support different features. Opal is the most recent framework and supports more of the recently added Java features and APIs than the other frameworks. Advanced features for which solutions were proposed in literature [BSS+11, SBKB15, FKS18]—such as Doop's dynamic proxy support—are not enabled by default. However, Soot, Wala, and Doop do instead support a large variety of different configuration options that all effect the CGs but are not completely covered within this work.

> *Obs.1:* In terms of feature support, Opal outperforms Wala, Soot, and Doop's CGs. Wala and Opal can be used to analyze Java 8 or newer. However, using Soot and Doop's CGs is still viable when analyzing older Java versions. Moreover, when using the dynamic reflection analysis tool *Tamiflex* [BSS+11], no explicit static reflection support is required.

RQ2 – What is the main source of unsoundness in built-in call-graph algorithms? All built-in CGs struggle with the resolution of reflective method calls. Another unsoundness source pertaining to Doop and Soot's CGs is the introduction of new language features. Several features introduced in Java 8 are not yet supported[8]. All frameworks struggle to support even newer features from Java 9 upwards. Furthermore, we observe that

[8] Java 8 first release was in March 2014.

APIs pertaining to dynamic language features, such as serialization, reflection, method handles, or classloading remain unsupported.

> *Obs.2:* While all frameworks are actively maintained, implementing support for up-to-date language features from Java 8 or higher seems challenging.

Additionally, corner cases, e.g., object creation via method references or static initializers of interfaces, hinder the sound and correct construction of CGs. WALA's defect $\text{WALA}_{\text{N-CFA}}$ algorithm as well as the lack of support for corner cases confirm the need for a comprehensive CG test suite. Moreover, it is important to be capable to comprehend a CG algorithm's strengths and weaknesses and provide a test suite for developers.

> *Obs.3:* This test framework can act as an integration test suite or a reference for a new CG algorithm and can be used to increase comparability across different implementations.

5.3.3. Threats to Validity

The performed evaluation demonstrates the design and usefulness of a comprehensive test suite to assess sources of *unsoundness* in CG construction algorithms. The test suite is, however, not complete with respect to all Java features, core APIs, or runtime (JVM) callbacks. For instance, test cases for JNI calls, general exception handling, Java 11 or higher, and others are missing. Additionally, it is possible that we were not able to identify all relevant test cases within the covered categories. However, the test cases were developed by researchers with many years of experience in doing Java-based static analyses and were cross-checked by the author of this thesis as well as another researcher. Furthermore, the thesis' author was responsible for constructing Java CG algorithms as part of his professional career, and a fellow researcher has developed Java bytecode analyses for more than 15 years. Hence, the likelihood that we missed relevant features is low. The test suite already covers 122 language features and APIs that are used in practice and, therefore, allows us to draw valid conclusions regarding the tested features and core APIs. Moreover, CATS is publicly available and, therefore, can be extended by independent researchers.

Since all test cases are manually annotated, there is a chance of annotation mistakes. To mitigate this threat, we thoroughly reviewed all our test cases and added a built-in verifier that checks if a test case is correctly annotated, e.g., it checks whether the annotated call is present within the bytecode. Furthermore, the programs were executed, and all annotations and unexpected results were independently verified by the author of this thesis and by another researcher independently.

5.4. Conclusion

In this chapter, we discussed the design of CATS, a comprehensive and extensible CG test framework. The proposed approach consists of a test suite and a pipeline that can a) parse the test suite, b) automatically run the tests against a CG algorithm, and c)

compare the computed CGs with the ground truth. Furthermore, we provide an interface that allows any bytecode analysis framework for Java to provide an adapter to test their CGs. The test suite consists of 122 manually forged and annotated test cases in 23 different categories. Each test case tests a unique feature relevant to CG construction and contains the ground truth as an annotation. We implemented four adapters that integrate twelve CG algorithms from four well-known static analysis frameworks: Soot, Wala, Doop, and Opal. By using these adapters, Cats enabled us to study and document the unsoundness of their CGs, thereby addressing *Chlg. 6*, *Chlg. 8*, and *Chlg. 9*. The evaluation revealed the weaknesses and strengths of Soot, Wala, Doop, and Opal's built-in CGs and that they vary significantly. Moreover, we were able to find a bug within Wala's N-CFA algorithm.

6. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs

Building on Hermes (cf. Chapter 4) and Cats (cf. Chapter 5), we design Judge, to analyze call-graph (CG) algorithms with respect to the language features they cover in a project-specific manner. Given a CG fingerprint—generated by Cats—and the features of an application for which we want to construct the CG—identified by Hermes—, Judge finds and documents sources of unsoundness in the application.

We use Judge to answer the following research questions:

RQ1 Which language and API features are used how frequently by which kind of code?

RQ2 How do Soot, Wala, Doop, and Opal compare to each in terms of runtime?

RQ3 Which CG algorithms are suitable for a specific application kind?

RQ4 Given support for manually tuning the entry-points considered by an algorithm, how much effort is necessary to increase the soundness of a CG to an acceptable level.

Subsequently, we first introduce Judge and then perform four experiments, one per research question.

6.1. Design

Figure 6.1 depicts the building blocks and the workflow of Judge for analyzing call-graph (CG) algorithms. Judge's input are (a) capability fingerprints from the CG algorithms under investigation and (b) a project for which we want to investigate the project-specific unsoundness of CG algorithms.

The upper part uses Cats to run all CG algorithms on the test suite and to compute fingerprints reporting whether the algorithms support a specific language feature or not. This part must only be performed once, unless new test cases or algorithms are added to Cats. The lower part of the workflow computes the CG for the input project with different algorithms and in parallel evaluates the prevalence of the CG-relevant features under investigation in the project code. Given the CG of a project P constructed by algorithm AL, the occurrence of the features $FSET$ under investigation in P's code, $FSET$, and the AL's profile, Judge reports potential sources of unsoundness of AL in P's CG. The latter step is called project-specific analysis of a CG.

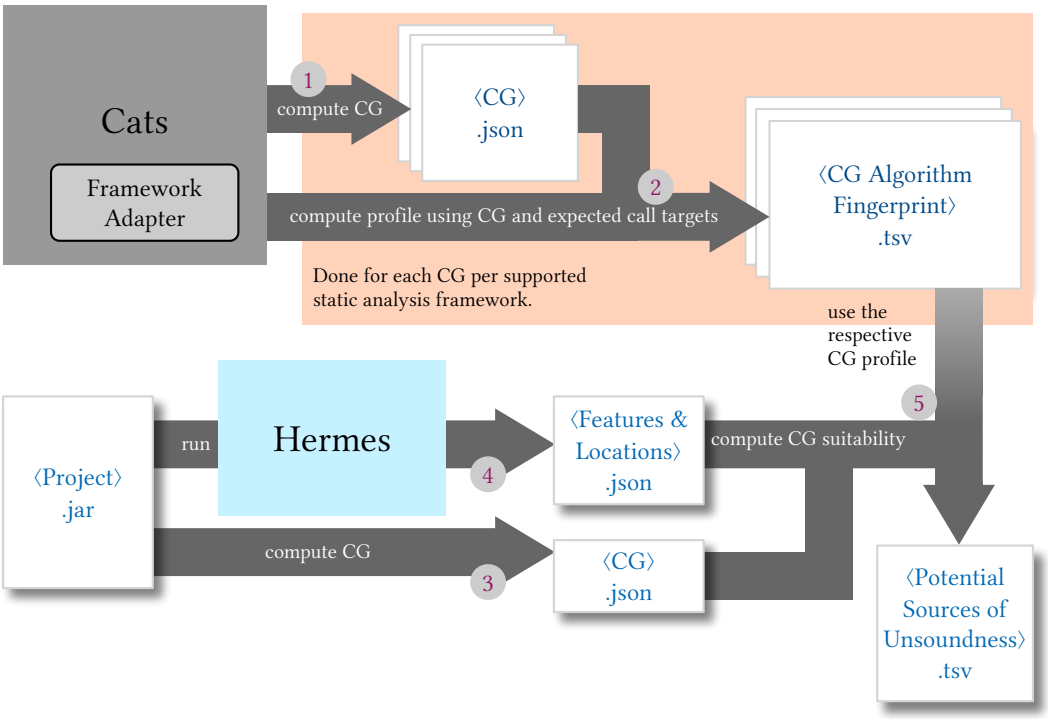

Figure 6.1.: Judge: An overview of our CG analysis toolchain.

As mentioned above, Judge relies on Cats and, therefore, is a) restricted to analyzing CG algorithms that are accessible over a framework adapter and b) is limited to the features that are tested within the test suite. Hence, Judge supports the analysis of the various CG algorithms offered by the frameworks: 4xWala, 4xSoot, 1xDoop, and 1xOpal. Furthermore, Cats' fingerprint contains overall 122 test cases and, therefore, we can investigate 122 features which are grouped in 23 categories (cf. Table 5.1).

6.1.1. Querying for Cats' features in Code

To understand the prevalence of features affecting the soundness of CGs (cf. Step 3/blue area in Figure 6.1), Judge uses Hermes, which was presented in Chapter 4.

To recap, Hermes executes code queries against a large code base and then produces reports on the queries' findings. Each query is an analysis that checks if a specific feature is found in a given code base. The result is a report that lists the locations (in terms of the instructions' program counters) that use a feature along with the Hermes feature id.

To judge a CG's soundness concerning a particular project, we must be able to map all the features we tested with Cats to real-world programs. In order to do so, we developed Hermes queries to derive the features modeled by the test cases. Then, we mapped all test case ids to the query's feature ids. For example, the query to check for occurrences of trivially-resolvable reflective methods searches for invocations of `java.lang.reflect.Method.invoke` and then uses data-flow information to check

if, e.g., the name of the target method can be locally resolved. When this query is run, it will reveal whether a method contains no reflective calls, trivially-reflective calls, or non-locally resolvable calls. If the method contains reflective calls, the result will also identify the respective bytecode instruction. Each of the 3 categories is considered to be a feature in Hermes terminology and is assigned a unique feature id.

All queries perform a most-conservative intra-procedural analysis. Ergo, test cases that require an inter-procedural analysis, e.g., test cases related to reflective calls that test if a framework is able to track strings across method call boundaries to (soundly and precisely) resolve reflective call targets, are only partially covered. Writing queries for these test cases would be subject to false positives and false negatives; the query would require information about the flow of strings in the application and no such analysis exists that is sound and precise. Therefore, it would be impossible to use those queries to reliably identify code locations that are sources of unsoundness.

However, for these test cases we write queries that determine that the local analysis is inconclusive and then flag the method accordingly. Such queries often handle multiple test cases by reporting that a finding belongs to one of multiple test cases, i.e., the query reports an id consisting of all test cases the finding may matches. For example, test cases of context-sensitive reflection are grouped because the query cannot distinguish where the method's parameter originates from.

Hence, the queries *only* derive 107 features for 122 test cases. Altogether, we developed 15 queries for Hermes.

6.1.2. Project-specific Call-graph Analysis

For the project-specific evaluation of an algorithm, we compute its CG for the project (Step 3 in Figure 6.1). As discussed previously, we use Hermes to find the locations of all features that may affect the soundness (Step 4 in Figure 6.1). Finally, the computed CG is used to determine all reachable methods that use unsupported features (Step 5 in Figure 6.1). This enables the identification of the *initial* sources of unsoundness[1].

Figure 6.2 illustrates the project-specific assessment of a CG algorithm. The first two columns are project agnostic and represent the CG algorithm's profile: the first one lists Hermes' features ids (which map to the respective test cases); the second one identifies a feature as being supported or not. Columns three to six are project specific: Column three (Extensions Count) shows how often a feature was found by the respective Hermes query—in our case, the project contained three polymorphic calls, two reflective calls, one Java `invokedynmic` instruction, and zero Scala `invokedynamic` s. The fourth column represents the mapping between the occurrences of a feature (column 3) and its locations/methods (column 5). Finally, column six shows whether the methods where the features were found are reachable from the constructed CG—i.e., are an *immediate source of unsoundness*—or not.

With respect to the reflection usage of method m_y, we make two observations: 1) the CG algorithm does not support the resolution of reflective method calls and 2) method

[1] Sources of unsoundness are always only potential sources of unsoundness because we do not check whether the instructions themselves are reachable.

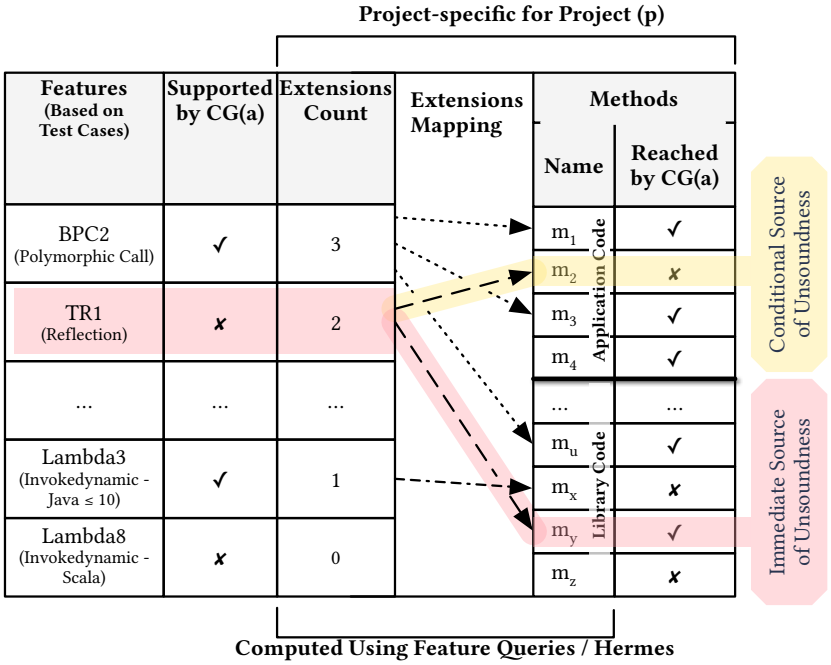

Figure 6.2.: Project-specific CG analysis.

`my` is already reachable. Hence, this reflection usage in m_y is a *source of unsoundness* because it knowingly leads to missing call edges. The reflective usage in method m_2 is—in contrast—not reachable according to the current CG and is so far a *conditional source of unsoundness*; i.e., it would be another source of unsoundness if the method would be reached. In other words, conditional sources of unsoundness are potentially relevant because the impact of known, not soundly handled features on the constructed CG remains unknown.

6.2. The Study

We perform four experiments to answer the following four research questions:

RQ1 Which language and API features are used how frequently by which kind of code?

RQ2 How do Soot, Wala, Doop, and Opal compare to each other concerning runtime costs?

RQ3 Which call-graph algorithms are suitable for a specific application kind, i.e., which kind of code base?

RQ4 Given support for manually tuning the entry-points considered by an algorithm, how much effort is necessary to increase the soundness of a call graph (CG) to an acceptable level.

6.2.1. Setup

All measurements are done using WALA 1.5.0, SOOT 3.1.0, OPAL's develop branch[2], and DOOP's master branch[3]. From WALA, we use the following algorithms: WALA_{RTA}, $\text{WALA}_{\text{0-CFA}}$, $\text{WALA}_{\text{N-CFA}}$[4], $\text{WALA}_{\text{0-1-CFA}}$—all configured with the `FULL` reflection option. WALA requires to specify packages to be excluded from the analysis. However, for this experiment we choose to exclude no package. For all SOOT CGs (SOOT_{CHA}, SOOT_{RTA}, SOOT_{VTA}, and $\text{SOOT}_{\text{SPARK}}$ [LH03]) we use the options *safe-forname* and *safe-newinstance*. These options make SOOT consider all types as instantiated when `Class.forName` or `Class.newInstance` is used. We could not use *types-for-invoke* due to exceptions being thrown [SDE+18]. Furthermore, we use *include-all* to ensure that no packages are filtered. Our library test cases are additionally started with *library:signature-resolution* and *all-reachable* to make use of SOOT's capabilities to analyze library code. DOOP's CG is set to be *context-insensitive* with *classical-reflection* turned on. For OPAL_{RTA}, we use the standard configuration. Please note that we described the used algorithms in Section 2.1.

All test cases with respect to libraries are started with the respective library entry points. We perform all experiments on a server with two Intel Xeon E5-2620 CPUs and 64 GB RAM.

6.2.2. Experiment 1: Studying the Prevalence of Language Features and APIs

Our corpus for analyzing the prevalence of language and API features (RQ1) includes the XCorpus [DSST17], the top 50 *distinct* libraries from Maven Central [Mvn18] (from July 2018), the top 15 apps from Google's Playstore (from January 2018), plus five popular Clojure [Hic18], Groovy [pro18a], Kotlin [kot], and Scala [Lau18] projects from GitHub. We run our Hermes queries on the entire corpus and evaluate and compare the results.

Table 6.1 visualizes the results using a heatmap. It shows the relative frequency of each feature (cf. Feature column) within each corpus. We include the OpenJDK column as a separate corpus because most corpus projects are built upon it and, hence, at least partially use its features. A feature's relative frequency is color coded using a logarithmic scale as shown in the legend of Table 6.1. Slightly yellow boxes (■) identify unused features and red boxes (■) those found in $\geq 5\%$ of all methods; we chose 5% because only seven of our 122 features occur in more than 5% of all methods. Features used in no corpus (e.g., Groovy *invokedynamics*, or the serialization of lambdas) and always soundly resolved features (e.g., standard poly-/monomorphic calls) are not included. For example, test caes from *NVC*, *VC*, and *Types* are omitted since all algorithms support them.

> *Obs.4:* All the API and language features supported by Java up to version 7 are used widely across all code bases.

[2] OPAL's commit id: 3107c45c8a00de0e132691a6275d39b5a4aa415b.
[3] DOOP's commit id: cdc59ce71d6510198da396cf6a7d20d73c6466d9.
[4] We use N=1 throughout the whole evaluation.

Table 6.1.: Feature prevalence across different corpora.

Feature	OpenJDK 8	XCorpus	Top50Maven	Scala	Groovy	Kotlin	Clojure	Android	Feature	OpenJDK 8	XCorpus	Top50Maven	Scala	Groovy	Kotlin	Clojure	Android
J8DIM1									ExtSer2								
J8DIM2									ExtSer3								
J8DIM3									SI1								
J8DIM4									SI2								
J8DIM5									SI3								
J8DIM6									SI4								
JVMC1									SI5								
JVMC2									SI6								
JVMC3									SI7								
JVMC4									SI8								
JVMC5									MH1.1								
Lambda1									MH1.2								
Lambda2									MH2								
Lambda3									MH3								
Lambda4									MH4								
Lambda5									MH5								
Lambda6									MH6								
Lambda7									MH7								
LIB3									MH8								
LIB4									TR1								
LIB5									TR2								
MR1									TR3								
MR2									TR4								
MR3									TR5								
MR4									TR6								
MR5									TR7								
MR6									TR8								
MR7									TR9								
Native									LRR1								
Ser1									LRR2								
Ser3									CSR1+CSR2								
Ser4									LRR3+CSR3								
Ser5									Unsafe1								
Ser6									Unsafe2								
Ser7									Unsafe3								
Ser8									Unsafe4								
Ser9									Unsafe5								
ExtSer1									Unsafe6								
									Unsafe7								

The most frequently used feature that was introduced with Java ≥ 8 is the call of static interface methods (*J8SIM6*). This case occurs in 12% of all methods of the top 50 Maven projects. However, *Scalatest* [Inc18] is responsible for ≈ 90% of all uses and, hence, the feature's heaviest user. Clojure and Android code have not yet adapted Java 8 call semantics. Other Java 8 features, e.g., `MethodHandle` constants, are rarely used; primarily by the *Nashorn* library.

Obs.5: Support for Java 8 is a must, given the frequent use of Java 8 call semantics features in modern code (*J8DIMX*), unless one analyzes only Android or Clojure code.

Serialization-related functionality (*Ser3-7,9*, *ExtSer*) and Java's Reflection API (cf. *TR, LRR, CSR*) are both used with medium frequencies; also in modern code.

Obs.6: Supporting classic reflection and serialization is strongly recommended, independent of the source code's age.

Many features (e.g., method references *MR*), Java's MethodHandle API (*MH*), native methods (cf. *native*), or Java's Unsafe API (cf. *Unsafe3-7*) occur with varying frequency and not in all corpora.

Obs.7: Support for many features is only required in specific scenarios.

Obs.8: The distribution of the feature usage is very different for the XCorpus when compared to the JDK 8 and/or the other corpora, therefore its representativeness for evaluating CG construction algorithms is limited. In particular, the usage of the Lambdas and the MethodHandle API increases when we compare its usage frequency in the XCorpus vs. the top 50 Maven libraries.

6.2.3. Experiment 2: A Detailed Assessment of State-of-the-art Call-graph Algorithms

In this experiment we compare different CG algorithms. As we found in Chapter 5, CG algorithms support different sets of features. We construct the CGs for five XCorpus projects (*jasml*, *javacc*, *jext*, *ProGuard*, and *sablecc*) to assess the CGs' size and construction time. We select these projects because they all have a) well-defined main classes, and b) can be processed by at least one CG algorithm of each framework. We run all CG algorithms once on all five projects including the Java Runtime Environment 1.6_30 from Doop's benchmark project [Sma]. The latter is chosen to attain comparability regarding the runtime; we set a 90 minutes timeout.

Table 6.2.: Performance comparison across call graphs from Soot, Wala, Doop, and Opal.

Project	**#Methods**		**Soot$_{CHA}$**		**Soot$_{RTA}$**		**Soot$_{VTA}$**		**Soot$_{SPARK}$**		**Opal$_{RTA}$**	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	12 184	18 s	12 134	75 s	8 012	17 s	10 356	22 s	3 195	13 s
javacc	162 484	2 185	13 035	22 s	12 986	97 s	8 863	22 s	9 752	17 s	4 222	12 s
jext	163 569	3 270	34 604	97 s	34 470	697 s	20 259	97 s	20 605	73 s	15 705	15 s
proguard	165 797	5 498	36 425	84 s	36 256	647 s	20 928	100 s	28 912	136 s	7 771	11 s
sablecc	162 670	2 371	14 138	18 s	14 088	104 s	9 687	24 s	12 101	24 s	4 932	11 s
average				47.8 s		324 s		52 s		54.4 s		12.4 s

Project	**#Methods**		**Wala$_{RTA}$**		**Wala$_{0\text{-}CFA}$**		**Wala$_{N\text{-}CFA}$**		**Wala$_{0\text{-}1\text{-}CFA}$**		**Doop $_{CI}$**	
	all (incl. JDK)	project	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
jasml	160 564	265	75 817	362 s	-	*timeout*	-	*timeout*	-	*timeout*	14 149	579 s
javacc	162 484	2 185	76 643	399 s	-	*timeout*	-	*timeout*	-	*timeout*	14 952	618 s
jext	163 569	3 270	79 513	411 s	-	*timeout*	-	*timeout*	-	*timeout*	27 194	1 698 s
proguard	165 797	5 498	80 240	465 s	-	*timeout*	-	*timeout*	-	*timeout*	18 205	949 s
sablecc	162 670	2 371	77 607	460 s	-	*timeout*	-	*timeout*	-	*timeout*	15 774	680 s
average				419.4 s		-		-		-		9048.8 s

```
1 Collection c1 = new LinkedList();
2 Collection c2;
3 if(cond){
4   c2 = new ArrayList();
5 } else {
6   c2 = new Vector();
7 }
8 c2.add(null); // Call site
9 Collection c3 = new HashSet();
```

(a)

```
1 public static Object copy(Object o){
2   return (String[]) o.clone();
3 }
```

(b)

Listing 6.1.: Code examples to demonstrate how WALA, SOOT, and OPAL's CGs differ in precision.

The performance results are shown in Table 6.2. Column one lists the project, column two gives the number of all methods including the JDK and column three the number of project methods. The remaining columns list the number of reached methods and the CG's construction times for each algorithm.

OPAL is the fastest framework. All of WALA's context-sensitive CGs timed out. DOOP has the slowest CG algorithm that still finished in time, followed by WALA_{RTA} and SOOT. The CGs constructed by RTA algorithms of SOOT, WALA, and OPAL differ extremely. This is partly due to the different handling of basic virtual methods calls, which all handle soundly, but with very different precision. Other reasons are the supported features as well as the different usage of receiver-type and cast information.

Listing 6.1a partly explains the difference. The three local variables `c1`, `c2`, and `c3` are assigned different subtypes of `Collection`, namely `LinkedList`, `ArrayList`, `Vector`, and `HashSet`. The call on line 8 is then resolved differently. WALA considers all instantiated subtypes of `Collection`, i.e., all types where the constructor is called in Listing 6.1a, SOOT computes an upper-type bound for `c2` and the call is thus resolved to all subtypes of `AbstractList`. OPAL computes union and intersection types and determines that `c2` can either be an `ArrayList` or a `Vector`. For this example, WALA would add four, SOOT three, and OPAL two call edges on Line 8.

Also unrelated to the supported feature set is the use of type information from casts, which are performed after a method call. In case of Listing 6.1b, WALA and OPAL use the cast (cf. Line 2), which is performed after the method call, to refine the object's upper-type bound to `String[]`, while SOOT and DOOP consider all `clone` methods as possible call targets. Thus, WALA and OPAL would add only a single call edge, whereas SOOT and DOOP would add one edge to each *clone* method defined within the analyzed program.

Obs.9: Non-conceptual differences have a significant impact on the computed CGs, indicating that it does not make sense to compare the results of static analyses that build upon CGs from different frameworks. Even if we use the implementations of the same algorithm across frameworks, the resulting CGs are vastly different.

In summary, there are significant differences between the frameworks in therms of performance. As we observe for different RTA CGs, it is impossible to relate the amount of supported features (cf. Chapter 5) to the algorithm's runtime. The use of different approximations—within a conceptually identical algorithm—also has a significant impact on the CG's precision and, therefore, its runtime [Bod18]. For example, imprecision is introduced by not utilizing casts. In addition, we still do not understand the performance consequences of partially supported features, e.g., reflection can not be supported precisely and, therefore, can be approximated in various ways.

Obs.10: From the observations above, we conclude that it is not possible to relate a CG's feature completeness to its runtime costs and its size. A CG's suitability needs to be analyzed in the context of a specific problem (domain).

6.2.4. Experiment 3: Project-specific Assessment

We assess JUDGE's suitability for project-specific evaluations using XCorpus' Xalan project. Xalan is a mid-sized project with a well-defined main class, for which we were able to run all CG algorithms within a 90 minutes limit. Xalan also uses features not handled by *any* CG implementation.

Table 6.3 shows an excerpt of the evaluation's results. The column *#Locations* shows whether a specific (un)used feature is prevalent in Xalan or in the JDK. Furthermore, it shows for each CG algorithm the reachable methods (*#RM*), its runtime, the number of reachable feature locations, and whether the respective feature is supported. Whereas ● indicates a supported feature, an unsupported feature is denoted by ⊖.

SOOT's CG algorithms are the only ones that handle all context-sensitive reflection test cases in a sound manner. This resulted in the biggest CGs whose computation also required much longer than those of WALA and OPAL.

However, all CGs contain reachable methods where unsupported features (⊖) are used, i.e., miss edges and are thus unsound. Though, OPAL's CG reaches the least number of *sources of unsoundness*, we also observe that OPAL's CG only contains 49 ($\approx 0.3\%$) methods from Xalan. WALA's RTA CG in contrast touches $\approx 50\%$ of all methods. A detailed investigation using JUDGE, starting from the identified sources of unsoundness, reveals that this is due to a single unsupported feature related to Java reflection. The cause is a helper method (`findProviderClass(...)`) in Xalan's `ObjectFactory`—it expects a class name as a parameter and loads the class via reflection. Soot and WALA are configured to act conservatively and, therefore, consider *all* available classes as *instantiable* when a `Class.newInstance` call is performed. As result, they add a call edge to *all* class' constructors which enables them to reach a large portion of Xalan's methods but also introduces a large amount of imprecision—as a manual analysis revealed.

Table 6.3.: Excerpt from the Project-specific Evaluation for Xalan.

F	#Locations		Soot$_{RTA}$		Soot$_{VTA}$		Soot$_{SPARK}$		Wala$_{RTA}$		Wala$_{0\text{-}CFA}$		Opal$_{RTA}$		Doop	
	#M	#M	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time	#RM	time
	16 389	251 239	58 560	2320s	28 248	322s	23 753	139s	15 343	15s	3 021	4s	6 834	22s	14 392	988s
	Xalan	JDK	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS	#RF	#FS
TR2	28	288	25	⊖	10	⊖	7	●	7	⊖	1	●	1	●	2	⊖
Ser3	1	97	1	⊖	0	⊖	0	⊖	0	⊖	0	⊖	0	⊖	0	⊖
MH1	0	107	13	⊖	8	⊖	13	⊖	0	⊖	0	⊖	0	●	14	⊖
LRR1	2	84	15	●	10	●	10	●	2	⊖	1	⊖	1	●	11	●
CSR1	38	176	49	●	34	●	31	●	20	⊖	6	⊖	4	⊖	7	⊖
JVMC4	2	23	4	⊖	3	⊖	5	●	2	⊖	0	⊖	0	●	0	⊖
J8PC1	81	9799	1165	⊖	450	⊖	396	⊖	236	●	42	●	221	●	316	⊖
Lambda1	0	621	30	⊖	14	⊖	14	⊖	5	●	0	●	3	●	1	⊖
MR3	0	1059	14	⊖	6	⊖	6	⊖	1	●	0	●	0	●	0	⊖
...																
Σ⊖	328	15012	1428	⊖	575	⊖	489	⊖	107	⊖	18	⊖	16	⊖		●
Σ●	328	15012	259	●	156	●	157	●	254	●	57	●	267	●		●

F=feature; M=methods; RM=reachable methods; RF=reachable features; FS=feature support

● indicates feature support and ⊖ an unsupported feature

Obs.11: The experiment shows that CGs contain methods that use unsupported features and are thus unsound. Unsupported features can have a devastating effect as, e.g., OPAL's poor coverage of Xalan demonstrates.

6.2.5. Experiment 4: Improving a Call Graph Manually

The experiments so far investigated the level of unsoundness of CGs due to incomplete feature/API coverage by CG construction algorithms. Whether unsoundness is tolerable or not depends on the use case. In this experiment, we consider usage scenarios where unsoundness cannot be tolerated, or, at least, needs to be minimized. An example for this is vulnerability analysis. To cover such use cases, OPAL provides a mechanism for manually specifying entry points that are taken into consideration by the CG algorithm. This mechanism can be used together with JUDGE, which provides assistance with analysing reachable methods that use unsupported features/APIs to understand the expected effect on the CG.

The goal of the experiment is to get an intuition of the effort needed to manually turn an unsound CG to a *reasonably sound* one. The subject was Xalan's CG produced by OPAL_{RTA}, which is unsound due to incomplete coverage of the reflection API. OPAL_{RTA} is used as it is most feature complete (cf. Table 5.2), hence, we expect to minimize the manual effort. What *reasonably sound* means depends in general on the use case. In this experiment, we consider a CG as being reasonably sound if it contains at least all results also found by dynamic analyses. We perform two dynamic analyses: (a) JVM profiling to log which methods are executed and (b) the dynamic analysis tool Tamiflex [BSS+11] for resolving reflective calls to record dynamic edges. Whereas we use the JVM profiling to check whether all executed methods are reachable in the CG, we use Tamiflex to examine whether the CG includes all reflective call edges that have been reported. Hence, when the CG contains both, we consider it reasonably sound. We profile Xalan using exemplary input and Tamiflex to record call targets of reflective calls and then iteratively use JUDGE along with OPAL_{RTA}'s mechanism to configure additional entry-point methods and types that must be considered as instantiated by the CG algorithm[5]. This way, we increase soundness manually step by step.

The initial CG covered 30% of all methods reported by a profiling run using exemplary input. None of the methods reported by TamiFlex were included. The manual analysis took ≈1.5 hours and required to analyze 10 reflective call sites, configure 17 types as instantiable, and configure 50 additional entry points. As a result, the CG covered 121 of 198 methods reported by TamiFlex. The remaining methods are related to code that is generated at runtime. Furthermore, the CG covered 1500 of 1653 methods (≈ 91%) when compared to the profile run; all non-reachable methods belong to the JDK. At this state, we consider the CG reasonably sound.

[5]The configuration of instantiated types is required since we are using a RTA CG which does not capture reflectively instantiated types.

Obs.12: The experiment indicates that the effort involved in manually increasing the soundness of CGs is high even for mid-sized projects and despite good tool support, i.e., manual correction is not proper compensation for better algorithms that automatically construct sound CGs. However, when repeatedly analyzing the same code base, the effort is well-spent, as the cost is worth the benefits.

6.2.6. Discussion

In the following, we discuss the implications of our study for both developers of CG algorithms and static analysis researchers that use the latter. Thereby, we highlight how JUDGE helps them to make more informed decisions, to reason about potential limitations of their tools and the root causes thereof, and to set up empirical evaluations and ensure reproducibility of their results.

Implications for Framework Developers Obviously, our experiments indicate that research on constructing high quality and practically useful CGs is still needed. We need new implementations that soundly cover features that are prevalent in real software, e.g., Java 8 call semantics. Furthermore, the implementations should support users in manually adjusting implementations and/or CGs, e.g., to integrate manually-defined parts of the graph in a project-specific way to handle encounters of unsupported features. Such a mechanism can help increase the soundness. So users can specify call edges that solve the most significant soundness/precision issues.

JUDGE can be useful for implementors of CG construction algorithms in several ways. It helps figuring out where manual adjustments of the CG are needed. For example, during debugging or when designing new abstractions. When implementing new or extending existing CG algorithms, it helps investigating the usage of unsupported features/APIs in practice. JUDGE—in particular its extensions to HERMES—can also help to create representative benchmark suites w.r.t. their used language features/APIs, enabling well-founded research that (in)directly relies on CG algorithms. What is still missing and needed, however, is support for understanding design decisions pertaining to precision. It is, in any case, important that every CG algorithm implementation documents its design decisions w.r.t. to approximations and optimizations. Finally, given that the JVM, the Java language, and its bytecode keep evolving, our comprehensive test suite (CATS) can be very useful as a regression test suite. Furthermore, it can be continuously enriched with new test cases for further domains/APIs that affect a CG's soundness by us or by others.

Implications for Static Analysis Researchers The results of our study directly inform developers of client analyses if a framework suits their specific needs. Soot and DOOP can be used to analyze Android code as their feature profiles match well the feature profile of this domain, while OPAL and WALA support analyses targeting Java 8 applications. In any case, researchers developing new static analysis tools/frameworks, should clearly specify the employed CG algorithm in order to increase reproducibility of their results.

In addition, it can be used to systematically evaluate CG algorithms w.r.t. their suitability to serve as a foundations for building analyses for a certain application (class), as it can provide an overview of the (prevalence of) features that are used in that application (class), so as to pick the most sound CG for the specific needs. Even Opal's broad feature/API support may be insufficient, if unsupported features, e.g., *CSR*, are used in the target applications. Knowing where the CG is unsound enables static analysis writers to understand whether a false negative originates from an unsound CG or is a problem of the analysis.

6.2.7. Threats to Validity

Internal threats to validity are the usage of incorrect Hermes queries. In that case, we may fail to identify the presence of language features/APIs that potentially cause unsoundness. To mitigate this threat, the implementations of the queries were reviewed carefully. Furthermore, we tested all Hermes on the respective test cases to ensure that the features are identified correctly.

An external threat is the usage of a non-representative corpus of programs. Our study has shown that an established corpus such as the XCorpus is not representative for modern Java code, as it does not contain usages for many relevant features. Other established corpora, e.g., Qualitas, DaCapo, etc. are even older than the XCorpus. Therefore, we used 7 different corpora of reasonable sizes and mixed in recent libraries from *Maven Central* as well as *GitHub* projects written in different JVM-hosted languages. This said, our work reveals the urgent need for a corpus that is representative for currently deployed applications.

6.3. Conclusion

In this chapter, we presented Judge for (1) the evaluation of language features and APIs that are relevant when building CG algorithms; (2) comparing CG algorithms; (3) evaluating how well-suited a specific algorithm is for a specific project, and (4) to facilitate the creation of project-specific sound CGs. Additionally, we performed extensive studies regarding the capabilities of four major Java static analysis frameworks and the prevalence of features that are not soundly handled. As discussed in the state-of-the-art in Chapter 2, we can confirm that the inequality pertaining to the researched features (cf. *Chlg. 4*) reflects in the CG algorithms capabilities. However, we were able to shed light on which features occur frequently in real-world code (cf. *Chlg. 5*), thereby enabling a prioritization of the next steps. All frameworks lack support for many features frequently found in-the-wild and—even for standard mono- and polymorphic calls—produce vastly different CGs. This renders comparisons of static analyses which rely on different CGs impossible and also considerably unsound.

Part III.

Modular Call-graph Construction for Libraries

The Next Step in Call-graph Construction

Despite the amount of previous work, call-graph (CG) construction remains a problem in practice [SDE+18, SDTF20, LSS+15]. Therefore, in Part II of this thesis, we empirically evaluated and compared existing CG algorithms using our automated test framework Cats (Chapter 5). Furthermore, we used Hermes (Chapter 4) in combination with Judge (Chapter 6) to study the relevance of individual programming language features and APIs in-the-wild and investigated their effect on a CG's unsoundness. Besides, we identified further challenges while discussing the state-of-the-art algorithms in Chapter 2. In both our study and our discussion, we identified several critical problems that affect a CG's soundness:

a) CG construction algorithms are designed for applications and lack a separate discussion of whether and how they can be applied to libraries in isolation, i.e., how the extension of library classes form the outside world affects the CG.

b) Call-graph algorithms provide limited configuration options that—if present—often pertain only to reflection. Support for other features is barely configurable.

c) Constructing CGs for individual programs requires the support for individual language features/APIs and, thus, each program has different requirements.

d) CG algorithms have poor comparability, not only because of the diverse feature support but also due to underlying design decisions.

Furthermore, problem a) is reinforced because application CG algorithms are used for libraries. A common hypothesis is that the algorithm will construct a sound library CG when all non-private methods are considered as entry points. However, this ignores that libraries are open worlds and that they usually provide a public API.

Such that CG algorithms can further improve, we must address all these problems appropriately.

In Chapter 7, we investigate the peculiarities of CG construction for libraries. We motivate the need for CGs dedicated to libraries and thoroughly discuss the design space for such CG algorithms during our investigation. Based on our discussion, we suggest two concrete CG algorithms based on the CHA algorithm: one can be used to identify security issues and one targets general software quality issues. We evaluate these algorithms, showing that classical CG algorithms do not serve the needs to analyze libraries.

In Chapter 8, we address the problem of configuring and individualizing CG algorithms. Specifically, we propose a novel generic approach together with a proof-of-

concept implementation in the Opal framework [EH14, EKH+18, HKR+20] for lattice-based fixed-point computations with support for lattices of any kind, including singleton-value-based, interval, and set lattices. To capture such a system's needs, we implement three case studies with somewhat different requirements. Founded on the requirements derived from the case studies, we then present and evaluate the approach. Its evaluation shows that it features modular analyses encoded as independently compilable, exchangeable, and extensible units. Therefore, it supports CG algorithms' needs to provide pluggability concerning precision, scalability, and sound(i)ness.

In Chapter 9, we report on the design of TACAI, a three-address code intermediate representation presented as case study in Chapter 8. According to the study of Chapter 5, the receiver-type information available in static analysis frameworks' intermediate representation significantly impacts CG construction. Therefore, the focus of Chapter 9 is to improve the comparability of CGs and further research the influence of an intermediate representation on CG construction. TACAI is an abstract-interpretation-based intermediate representation with exchangeable abstract domains. By exchanging the used domains, one can adapt the precision of the information encoded by the three-address code. We will discuss and evaluate how switching the used abstract domains affects the bytecode-to-TACAI transformation.

In Chapter 10, we present some advanced CG algorithms for libraries. We reimplemented four application CG algorithms from Tip et al. [TP00] and then extended them, thereby further exploring the knowledge gathered in Chapter 7. To implement these algorithms, we use the approach presented in Chapter 8 and TACAI, which we discussed in Chapter 9. Furthermore, we compare these four algorithms and determine their viability for library analysis.

7. Call-graph Construction for Java Libraries

Currently, the gold standard for constructing call graphs for libraries is to use a standard algorithm, such as Class Hierarchy Analysis (CHA) [DGC95], Rapid Type Analysis (RTA) [BS96], or Variable-Type Analysis (VTA) [SHR+00] and to consider all non-private methods as entry points. However, this ignores two properties that distinguish libraries from standalone applications. First, libraries are not closed worlds—their users can extend them via inheritance. Second, libraries consist of classes and interfaces that either define the public API or belong to the library-private implementation.

In this chapter, we discuss how ignoring the first property leads to the construction of call graphs that miss important call edges, while ignoring the second property leads to call graphs with many spurious edges. Consequently, we argue that call-graph algorithms for libraries must distinguish between two usage scenarios of the library.

In this chapter, we address these problems and make the following contributions:

- A motivation for call-graph algorithms dedicated to libraries and a thorough discussion of the design space for such algorithms.
- Two concrete call-graph algorithms for libraries based on adaptations of the CHA algorithm: one that can be used to identify security issues (LibCHA$_{\text{OPA}}$) and one that can be used to find general software quality issues (LibCHA$_{\text{CPA}}$).
- A comprehensive empirical evaluation which shows that call graphs computed by the classical CHA algorithm and those computed by LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$ are significantly different. The evaluation supports our claims that a) classical call-graph construction algorithms (specifically CHA) do not serve the needs of either security or general quality related analyses of libraries and b) we need two types of algorithms to address the respective needs.
- A case study that shows that using LibCHA$_{\text{CPA}}$ as the foundation of a dead methods analysis enables us to find ≈6 times more dead methods compared to a solution based on the classical CHA algorithm.

7.1. Why Library Call Graph Algorithms?

In this section, we motivate the algorithms presented in this chapter. We start by characterizing a library's private implementation versus its public interface. Then, we motivate the need to consider *all* possible extensions of the library by means of inheritance to

```
1   package library {
2     interface J { public void mj() }
3     public interface K { public void mk() }
4     class A { public void mk(){} /*API*/ }
5     class B extends A implements J,K {
6       public void mj(){} /*Impl.*/
7     }
8     class C {
9       public void mc(){} /*API*/
10      public void md(){} /*Impl.*/
11    }
12    public class D extends C {
13      public void md(){} /*API*/
14    }
15    class E implements K { public void mk()  }
16    class F { public void mj()  }
17    public class Factory {
18      public K createBK(){return new B();}
19      public Object create(){new E(); return new
            B();}
20    }
21  }
```

Listing 7.1.: An example of a simple library.

construct a sound library call graph[1]. Finally, we explain that different library usage scenarios may require different kinds of analyses, which in turn need different call graphs.

7.1.1. A Library's Private Implementation

Conceptually, a library's private implementation consists of all code that a library user cannot directly use. Under the open-package assumption (OPA), a library's private implementation consists of all methods and fields with private visibility. Under the closed-package assumption (CPA), the library's private implementation additionally includes (a) every code element (class, method or field) that has at most package visibility and (b) all protected and public fields/methods of a package visible class, unless they are indirectly exposed to the library's user. The latter happens, e.g., if the package visible class inherits from a public class or interface and overrides or implements a method declared by the super-type, or it has a subclass that is public (or implements a public interface), which inherits the respective method. In other words, a field/method of a package visible class does not belong to the private implementation, if a user can potentially directly access the field/method.

To illustrate CPA, consider the code in Listing 7.1. The types `A`, `B`, `C`, and `J` belong to the library-private implementation. The class `B` implements the public interface `K`,

[1]Unless reflection, native methods, or Java's Unsafe API are used.

```
1  public interface I {
2    public m();
3  }
4
5  public class C {
6    public m(){ ... };
7  }
```

(a) An example of a library's public API.

```
1  public AppClass
2    extends C implements I {
3  /**
4    Method m is not overridden.
5    Hence, it is inheritated from C.
6  */
7  }
```

(b) An example app depending on the library.

Listing 7.2.: Excerpts of two code pieces that have not been compiled together.

which defines the public method `mk` (Line 3). Hence, a method with declared return type K could actually return an object of type `B` (Line 18), enabling the user to call the method `mk` defined by `A` (Line 4). Therefore, `<A>.mk` belongs to the public API. In case of the public methods defined by `C` only the method `mc` (Line 9) belongs to the public API. This method is inherited by `D` and is not overridden. Hence, a user who calls `mc` on an instance of `D` actually invokes `<C>.mc`. The method `<C>.md` (Line 10) is overridden by `D` (Line 12) and therefore belongs to the private implementation.

Our approach is conservative in classifying elements as part of the library implementation in the sense that it may classify code as belonging to the public API, although a user of the library cannot actually use it. For example, the method `<E>.mk` (Line 15) would be identified as belonging to the library's public interface, because the class implements the public interface `K`. Even if `E` is never returned to a user: `E` does not escape the scope of the library. Hence, a user will never be able to invoke `<E>.mk`. However, by being conservative we ensure that we will not miss a call edge.

7.1.2. Covering Possible Library Extensions

Established algorithms ignore OPA usage scenarios, which is understandable given that extension code does not need to be considered, when analyzing standalone applications. Yet, extension code can lead to *direct call dependencies between library methods* that are not apparent from the class hierarchy.

For illustration, consider the situation displayed in Listing 7.2. Listing 7.2a shows an excerpt of a library containing an interface and a class independent of each other. They neither are related through inheritance not use each other. Using a standard call-graph algorithm to construct a call graph of this library, a call $<I>.m$ would not be resolved against $<C>.m$ as C is not a subtype of I. Yet, a user of the library, such as shown in Listing 7.2b, may later on create a subclass of `C`, such as *AppClass*, that also implements I, but does not override `m`. Hence, to produce a sound call graph for the library, the call $<I>.m$ also needs to be resolved against $<C>.m$.

Consequently, when constructing a library call graph, we need to perform so-called call-by-signature resolution for all interface-based calls. The need to do so is exemplified by a real-world security bug (CVE- 2010-0840) found in the JDK. To facilitate comprehension

of the bug, we will first introduce the basics of the Java Security Model before we will discuss the attack in detail.

The Java Security Model allows to execute untrusted code safely, i.e., even malicious code cannot do any harm to the executing environment. This is required whenever a user may not trust the provided application, e.g., a Java Web Start application. The Security Model is a stack-based access control mechanism that guards all sensitive actions by permission checks which verify that all the code on the current call stack is granted access to that sensitive action.

Often attacks exploit forgotten permission checks, but there have also been attacks, in which trusted code is used to call sensitive actions on behalf of the attacker. The trick to make this work is to make sure that the attacker's code is not on the call stack, when permissions are checked. This is possible, if library code accepts a callback that is provided by an attacker. But, the callback can not be implemented by the attacker, because the implementation would be unprivileged and present on the call stack. Instead, the attacker must find a suitable callback implementation that can be configured to fit his needs.

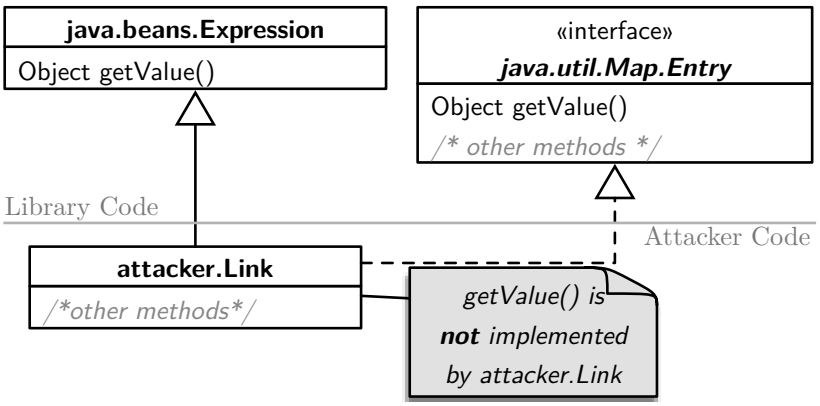

Figure 7.1.: Type hierarchy scenario for the trust-method chaining attack.

One such vulnerability is documented in CVE-2010-0840: Attackers exploited that checks consider the permissions associated with the declaring class of the method on the call stack but not its runtime receiver type. Hence, the runtime receiver type may belong to untrusted code. For illustration, consider Figure 7.1. Both the interface `Entry` and the class `Expression`, belonging to the library, declare the method `getValue()`. `Expression` neither implements `Entry` nor is otherwise semantically related to it. `Expression` encapsulates a reflective call, whose receiver, method, and arguments are specified by arguments to its constructor. The reflective call is performed in `Expression.invoke()`, which is called from within `getValue()`.

Now, consider the following trusted method chaining attack which bypasses the Java Security Model. The attacker must create an `Expression` instance that encapsulates a `System.setSecurityManager(null)` invocation. Yet, he cannot call `invoke` on the `Expression` object himself. Instead, he defines a class `attacker.Link` that extends

`java.beans.Expression` and implements `java.util.Map.Entry`, thus linking `Expression` and `Entry`: As a result, `Expression.getValue()` is now an implementation of `Entry.getValue()`. The last step is to find a library method that accepts an `Entry` object and calls `getValue()` on it, such that when this happens the attacker is not on the call stack.

```
1  HashSet<Map.Entry<Object,Object>> set =
2      new HashSet<>();
3  set.add(
4      new
           Link(System.class,"setSecurityManager",null));

5  JList list = new JList(new Object[] {
6    new HashMap<Object, Object>() {
7      public Set<Map.Entry<Object, Object>>
           entrySet(){
8        return set;
9    }}});
10 JFrame frame = new JFrame();
11 frame.getContentPane().add(list);
12 frame.setSize(50, 50);
13 frame.setVisible(true);
```

Listing 7.1: Code example that shows the attack's setup.

This behavior is provided by the user interface thread of AWT/Swing. This thread dispatches events, as illustrated in Listing 7.1. In Line 5, a `JList` object is created. A `JList` usually uses a `ListModel` to control its contents and representation. As an alternative, an arbitrary array can be used. The content representation is then based on each array element's String representation as returned by `toString()`. We add a custom `HashMap` implementation to `JList` (Line 6), which in its default implementation of `toString()` calls `getKey()` and `getValue()` on its entries accessed by `entrySet()`. The `entrySet` method (Line 7) of the custom `HashMap` returns a set containing the preconfigured `Link` instance (Line 4). Finally, the `JList` is added to a `JFrame` (Line 11) and the latter is made visible. This triggers a *paint* event which is processed by the user interface thread.

The shortened call stack of that processing is shown in Figure 7.2. The user interface thread transitively calls `toString()` on all its contents, when it *paints* `JList`. We provided the custom `HashMap` implementation as content, which does not override `toString()`, thus `AbstractMap. toString()` is called. `AbstractMap. toString()` iterates over the entry set and calls `getValue()` on each entry. The only element of the entry set is the attacker's `Link` instance. Therefore, it effectively calls `Expression.getValue()`, which, in turn, reflectively invokes `System.setSecurityManager(null)`. Setting a new security manager is dangerous and therefore guarded by a permission check, but—as illustrated in Figure 7.2—no method defined by the attacker is on the call stack, therefore access is granted.

To systematically find exploitable callback implementations, a static analysis must check that there exists no attacker callable method that transitively calls sensitive actions

```
java.lang.SecurityManager.checkPermission(RuntimeP...)
java.lang.System.setSecurityManager(SecurityManager)
java.beans.Expression.invoke()
java.beans.Expression.getValue(Object)
java.util.AbstractMap.toString()
...
javax.swing.JList.paint(Graphics)
...
java.awt.EventDispatchThread.run()
```

Figure 7.2.: The call stack at the time of the permission check.

without proper sanitization or permission checks. Such static analyses have already been proposed [BDF01, BBFG04, Cha06, KPK02, SRH95]. However, the static analysis has to furthermore consider that calls to callbacks are resolved to *all* possible trusted implementations. State-of-the-art call-graph algorithms will not include a call edge from call sites of `Entry.getValue()` to the method `Expression.getValue()`; though this edge is required to find the attack that we presented here. If this edge is included, data flow analyses looking for unguarded paths to sensitive actions are enabled to identify the vulnerability.

7.1.3. Closed-package Usage Scenarios

A call graph algorithm that considers all possible extension scenarios of the library, while being sound for analyses under OPA, is not appropriate for analyzing libraries under CPA. For illustration, consider the JRE 7's class `java.awt.datatransfer.MimeType`, which is package visible. This class belongs to JRE's private implementation, which is clearly suggested by the comment directly above the class:

> *THIS IS *NOT* - REPEAT *NOT* - A PUBLIC CLASS! DataFlavor IS THE PUBLIC INTERFACE, AND THIS IS PROVIDED AS A ***PRIVATE*** (THAT IS AS IN *NOT* PUBLIC) HELPER CLASS!*

This class defines the `public` method `match(String)`, which is not (no longer) used by code within the JDK and the class is also not exposed to the client by any means. Hence, this method belongs to the JRE's private implementation and does not have any intended users anymore; i.e., it is a dead method. JDK developers would certainly want to detect and remove such methods from the library, e.g., to improve code comprehension, to avoid useless maintenance, or to shrink the overall size of the library code. Yet, in a call graph that is constructed to cover all extension scenarios—as described in the previous subsection—the method would be treated as an entry point. Hence, an analysis searching for dead methods on top of such a call graph would not report it.

When a developer analyzes a library concerning general software quality attributes, such as the presence of dead methods or dead code, they want to treat the library as *closed*, i.e., he deliberately does not want to consider code that (eventually) extends,

accesses or calls library-private code, which makes it possible to construct a more precise call graph w.r.t. the intended usage of the library. For example, a developer of a library that uses the namespace prefix *x.y.z* to analyze the implementation of the library itself will not take into consideration what may happen if a user of the library puts code in the package *x.y.z*.

Private code is pervasive in many Java libraries. For example, the Oracle JDK 8[2] defines 8,330 ($\approx$ 40%) package visible classes. Additionally the public classes contain further 11,786 ($\approx$ 9%) package visible methods and 6,668 ($\approx$ 20%) package visible fields. Call graphs which take the distinction between the implementation of the library and its public API into consideration represent intended usage scenarios. In such a graph there will be no edge to the method `java.awt.datatransfer.MimeType.match(String)`, allowing a respective analysis on top of the call graph to spot it as dead.

7.2. The Call-graph Algorithms

The proposed call graph algorithms for libraries are build on top of the Java bytecode framework OPAL and are defined w.r.t. the JVM's semantics. Implementing them as bytecode analyses has two advantages. First, it makes them useable for security related analyses, because attackers can always directly craft bytecode. Second, we can also analyze libraries written in other languages such as Scala or Groovy. Both algorithms require that the bytecode of the library $\mathcal{L}$, for which we want to construct the call graph, and that of any library $\mathcal{L}_{Dep}$ used by $\mathcal{L}$ are available and can be analyzed; this includes in particular the JDK.

Both algorithms share the following main steps. First, they determine the set of entry points under their respective assumption (OPA or CPA). Second, they set each entry point method to be reachable. Third, a fixpoint computation is performed that computes the call graph. The fixpoint is computed when all methods that are marked as reachable are analyzed, which is an iterative process. For each method call found in a reachable method, both algorithms determine the set of potential call targets based on the class hierarchy; i.e., calls to methods may resolve to any subtype of the receivers static type. This resolution of call targets is the same as done by class hierarchy analysis (CHA) [DGC95].

If the receiver type is an interface, both algorithms additionally perform call-by-signature resolution to identify those methods defined by classes that have a matching signature (name, parameter types and return type), but where the class does not inherit from the interface type. In case of OPA, all these methods defined by non-final classes are potential targets. In case of CPA, the interface and also the declaring class visibilities are further evaluated to determine if the target method is a potential target. Each method that is a potential call target is then marked as reachable.

In the following, we elaborate on the two steps of the call graph algorithms that lead to the different call graphs: (1) the computation of entry points and (2) the computation of the call targets in case of call-by-signature resolution.

[2]The classes found in the `rt.jar` of the Mac Version of Oracle JDK 8 updated 66.

```
1  def isEntryPoint(declType, method):Boolean =
2    maybeCalledByTheJVM(method) ||
3    method.isStaticInitializer ||
4    (!method.isPrivate &&
5        (method.isStatic || declType.isInstantiable))
```

Listing 7.3.: Entry-point predicate in case of the open-package assumption.

```
1  def isEntryPoint(declType,method):Boolean =
2    maybeCalledByTheJVM(method) ||
3    (method.isSaticInitializer && declType.isAccessible) ||
4    (method.isClientCallable &&
5        ( method.isStatic || declType.isInstantiable))
6
7  def isClientCallable(declType,method):Boolean =
8    (method.isPublic || method.isProtected) &&
9    (declType.isPublic ||
10       declType.subclasses.exists{ subC =>
11           subC.isPublic && subC.inherits(m)})
```

Listing 7.4.: Entry-point predicate in case of the closed-package assumption.

7.2.1. Entry-point Computation

Roughly speaking, a method is an entry point if it can be called (a) by the JVM (e.g., `finalize`) or (b) directly by a user of the library. The differences between the two algorithms are described next. LibCHA$_{\text{OPA}}$ determines if a method is an entry point by the predicate shown in Listing 7.3.

The first test (Line 2) identifies those methods that may be called directly by the JVM. For example, the `finalize` method is called by the JVM. Another example are serialization related methods, e.g., `readObject`, in `Serializable` classes. These methods are implicitly called by the JVM during the (de-)serialization process. These methods are often `private` and would not be considered as entry points otherwise. The second test (Line 3) checks whether the method is a static initializer. The last test (Line 4) is true if a method is non-private and static or if the non-private instance method's decaring class is instantiable. In this scenario, a class is instantiable if the class has a non-private constructor or has a factory method that potentially creates and returns instances of the class. A factory method is every static method with a return type that is a supertype (reflexive) of the class type and which calls a private constructor.

LibCHA$_{\text{CPA}}$ determines whether a method is an entry point by the logic depicted in Listing 7.4.

The first test `mayBeCalledByTheJVM` (Line 2) is the same as in case of LibCHA$_{\text{OPA}}$. The second test (Line 3) is extended and now also tests if the static initializer's declaring class (`declType`) is accessible. The latter is the case if the class or a subclass of it can be

```
1   def cbsTargets(declIntf, mSig) : Set[Method]=
2     project.findConcreteMethods(mSig).filter { m ⇒
3       m.isPublic &&
4       !m.definingClass.isEffectivelyFinal &&
5       !(m.definingClass <: declIntf)
6       /*in case of CPA:*/
7       &&
8       ( m.definingClass.isPublic ||
9         m.definingClass.subclasses.exists{subC ⇒
10          subc.isPublic &&
11          !(subC <: declIntf) && subC.inherits(m)} )
12    }
```

Listing 7.5.: Pseudo-code showing how to compute call-by-signature targets.

referenced from client code. In general, a class is referenced whenever the name of the class can appear in the code without violating visibility constraints. Hence, all public classes and also all package private classes that have a public subclass are immediately accessible.

Each method that does not satisfy one of the first two tests needs to be callable by a library's user (Line 4) and either must be `static` or be defined by a type that is instantiable. A method is callable (Line 7) if the given method has public or protected visibility (Line 8) and the declaring class of the method is either public (Line 9) or has a public subclass (Line 10), which does inherit the method, i.e., the method is not overridden on the path from the declaring class to the public subclass. In this case, a class is instantiable if and only if it is instantiable as in case of LibCHA$_{\text{OPA}}$ and is accessible as dicussed in the previous paragraph.

7.2.2. Call-by-signature for Libraries

The algorithm to compute the edges that must be added to the call graph due to call-by-signature resolution (CBS resolution) in case of interface-based calls is depicted in Listing 7.5.

Given an interface as well as the signature of a method m_{sig} defined by the respective interface, the algorithm returns a set of all methods that are *only* resolved by signature, i.e., a method with the given signature that is defined in a class that implements the interface will not be returned.

The first step, which is shared by LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$, is to identify all call targets by finding all non-abstract, public instance methods that have the same method signature as $\texttt{m}_{sig}$ (Lines 2–13). For each such method – in the following referred to as `m` – we then check whether `m`'s defining class `C` is effectively final, i.e., whether `C` is declared `final` or if `C` only defines private constructors which makes it impossible to inherit from it. In both cases, `C` cannot be subclassed and hence, `m` cannot become a call-by-signature call target.

Analysis Context			Closed Library Assumption	CBS
Library	Security Issues (LibCHA$_{OPA}$)	in *our* library	no (Someone will try to break our library.)	yes
		in 3rd party libraries	no (Other libraries may trie to break it.)	yes
	Software Quality (LibCHA$_{CPA}$)	in *our* library	yes (We don't care about misuses of our library.)	yes
		in 3rd party libraries	yes (We are using the 3rd party libraries as intended.)	yes
Application	both security and general issues		(implicitly)	no

Table 7.1.: The design space for library call-graph algorithms.

In case of LibCHA$_{CPA}$, we additionally check (Lines 7–12) if either `m`'s defining class ist public or if `C` has a public subclass (Line 10) which does not implement the given interface `I` and which does not override `m` (Line 11).

7.2.3. Summary

As argued in Section 7.1, depending on the goal of the analysis, we must choose the respective call graph algorithm. The two different algorithms that we presented in this section, LibCHA$_{OPA}$ and LibCHA$_{CPA}$, serve this need.

As shown in Table 7.1, when we want to analyze a library (be it our own or a 3rd party library) w.r.t. security issues then we must make the most conservative assumptions and this requires that we analyze the library using the open-package assumption (OPA). Additionally, we must use call-by-signature resolution related to all interface method calls to ensure that the call graph is sound. We must make these conservative assumptions when we analyze a third party library, e.g., the JRE, because it is conceivable that another library `A` that we also want to use tries to attack `JRE`. To handle these cases we use LibCHA$_{OPA}$.

When we want to analyze a library (be it our own or a 3rd party library) w.r.t. general software quality issues then we shall create the call graph based on the assumption that the library is used as intended by its developers. Hence, we can treat the library as closed and analyze it under the closed-package assumption (CPA) and use LibCHA$_{CPA}$. As with LibCHA$_{OPA}$, we must consider call-by-signature calls but now only those that have a relation to the public API. For example, the package visible class `F` in Listing 7.1 defines a method `mj` (Line 16) that is also defined by the interface `J`. Under OPA every call to `<J>.mj` must be resolved against `<F>.mj`. Under CPA, `F` belongs to the library private

implementation, hence, the user cannot create a subclass of `F` that also implements the interface `J` and therefore a (library internal) call to `<J>.mj` is not resolved against `<F>.mj`.

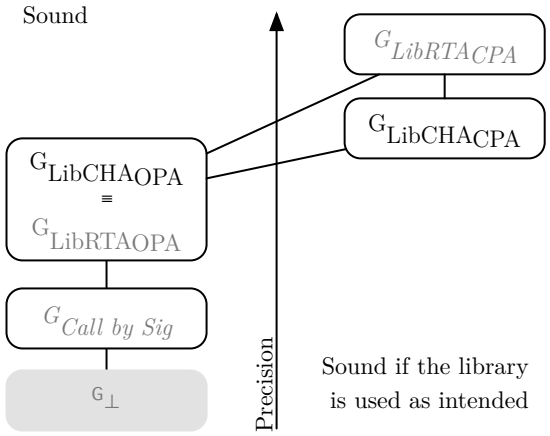

Figure 7.3.: Theoretical precision of call-graph algorithms.

The relation between the proposed/discussed call graph algorithms is depicted in Figure 7.3 and is inspired by the representation used in previous work [GC01]. It represents the relative precision of the algorithms compared to each other. As indicated in the figure, a call graph that is constructed using call-by-signature resolution for all types of methods calls would also be sound for libraries. As in case of applications, these call graphs are so huge that they are hardly useable [GC01].

The first meaningful call graph algorithm is $\text{LibCHA}_{\text{OPA}}$ which is sound but rather imprecise because the identification of the library's private implementation is most conservative. Only code that is technically not usable by any client, because it is private, is considered as belonging to the private implementation. This property makes the respective call graphs suited for security focused analyses. $\text{LibCHA}_{\text{CPA}}$, on the other hand identifies a library's private implementation based on generally agreed best practices and a thorough analysis of the visibility of the library's classes and methods. Hence, the resulting graphs are unsound if the library is not used as intended, but they much better approximate (and in non-security related cases still over approximate) all potential real runtime call graphs. Therefore these call graphs are better suited for general software quality analyses.

Theses ideas generalize to other call graph algorithms, i.e., it is conceivable to adapt other established call graph algorithms such that they can be used to analyze libraries. For RTA, e.g., we would also consider calls based on their signatures. Additionally, it is necessary to treat all classes that could be instantiated as instantiable. In case of OPA, it would result in a call graph that is identical to the one created by $\text{LibCHA}_{\text{OPA}}$. In case of CPA, it would be possible to define one scope per package and to associate all package private classes that are confined to their package with their respective scope. This could lead to a reduction in the number of call edges when compared to $\text{LibCHA}_{\text{CPA}}$.

7.3. Empirical Study of Library Call Graphs

We evaluate the proposed approach along two major dimensions: (1) by comparing a call graph computed using the classical CHA against the call graphs computed by our call-graph algorithms; (2) by comparing both proposed call-graph algorithms.

7.3.1. Setup

The empirical study is designed to help answer the following questions:

RQ1 Do we need call graph algorithms specialized for libraries?

RQ2 Is it necessary to distinguish between the open- and closed-package assumptions?

RQ3 Does a precise computation of the entry point set lead to a more precise call graph?

RQ4 What are the performance characteristics of the proposed algorithms?

We used the three algorithms to construct respective call graphs for a large set of libraries[3]: the 100 most used *distinct*[4] Java related libraries from Maven Central Repository [5]. The set is representative for a wide range of libraries. It contains very small (e.g., *JUnit*) to very large (e.g., *Scala Library*) libraries; libraries developed primarily in an industrial context (e.g., Guava) or in an open-source setting (e.g., *Apache Commons*); libraries from very different domains: testing (e.g., *Hamcrest*, *Mockito*), databases (e.g., *HSQLDB*), bytecode engineering (e.g., *cglib*), runtime environments (e.g., *Scala Runtime*), containers (e.g., *Netty*), and also general utility libraries (e.g., *osgi.core*). Additionally, it contains two libraries that have unusual properties: *jsr305* and *easymockclassextesion* both do not contain a single instance method call. The *jsr305* project is just a collection of annotations and *easymockclassextesion* only contains interface definitions and a few classes with static methods. Lastly, the set also contains libraries that are written in other languages, such as Scala (e.g., ScalaTest), whose compilers only use a subset of the JVM's concepts. The Scala compiler, e.g., does not use package and protected visibility. This significantly limits our possibilities to identify the library-private implementation (recall that LibCHA$_{\text{CPA}}$ identifies a library's private implementation based on the evaluation of the code elements' visibilities). For each library, we also downloaded all of its dependencies, downloaded via *sbt*[6], to build complete class hierarchies for them.

[3]The script to run the experiments can be downloaded from: `http://www.st.informatik.tu-darmstadt.de/artifacts/DLC/`.

[4]In case of libraries that appeared multiple times in the list, we just downloaded the most current version.

[5]`http://mvnrepository.com/popular` (as of Dec. 2015).

[6] `https://www.scala-sbt.org/` (checked on Dec 30, 2020).

Project	Classes and Interfaces (Ifc.)				Call Edges					
	Library		Dependencies		Naïve	LibCHA$_{OPA}$			Ratios	
	Class	Ifc.	Class	Ifc.	CHA	$\sum$	By Type	By Sig.	CHA Edges	CBS Edges
maven-plugin-api	22	3	28.661	3 257	8 118	19 320	8 117	11 203	99.99%	57.98%
httpcore	182	72	28.213	3 150	51 067	92 922	51 026	41 896	99.92%	45.04%
slf4j-log4j12	6	0	28.527	3 178	1 001	1 785	1 001	784	100.00%	43.92%
hsqldb	522	65	28.213	3 150	306 414	449 786	306 110	143 676	99.90%	31.88%
plexus-container-default	142	45	28.801	3 185	116 515	165 670	116 112	49 558	99.65%	29.67%
hamcrest-core	40	5	28.213	3 150	22 491	22 909	22 463	446	99.88%	1.82%
json	17	1	28.213	3 150	92 533	93 769	92 330	1 439	99.78%	1.32%
cdi-api	25	76	28.259	3 166	12 865	13 015	12 850	165	99.88%	1.15%
jsr305	5	30	28.213	3 150	88	88	88	0	100.00%	0.00%
easymockclassextension	5	2	28.213	3 150	133	133	133	0	100.00%	0.00%
mean (over all projects)									99.89%	16.78%
std dev (over all projects)									0.10%	9.90%

Table 7.2.: Number of call edges from LibCHA$_{OPA}$; our call graph using the open-package assumption and call-by-signature resolution.

Project	Types		Methods		Entry Points (EPs)			Call Edges
	Pub.	Pkg.	$\sum$	Pkg.	LibCHA$_{\text{OPA}}$	LibCHA$_{\text{CPA}}$	Reduction	Reduction
lombok	58	56	242	64	161	108	32.92%	0.02%
hsqldb	189	125	4 687	1 502	4 008	2 739	31.66%	2.73%
guava	370	1 350	13 200	3 562	11 714	8 402	28.27%	0.06%
derby	997	755	23 345	3 875	17 721	13 033	26.45%	0.54%
gson	59	106	957	185	826	608	26.39%	1.47%
scalacheck_2.10	1 997	0	8 651	0	8 266	8 266	0.00%	0.00%
scalac-scoverage-plugin_2.11	172	0	1 068	0	1 006	1 006	0.00%	0.00%
scalatest_2.10	6 755	0	82 680	0	79 197	79 197	0.00%	0.00%
jsr305	35	0	30	1	15	15	0.00%	0.00%
easymockclassextension	7	0	27	0	27	27	0.00%	0.00%
mean (over all projects)							8.04%	0.41%
std dev (over all projects)							7.27%	1.47%

Table 7.3.: Entry points in OPA and CPA; *Pub.* is used for public and *Pkg.* for package private visibility

Project	Visibility				Call Edges					
	Types		Methods		LibCHA$_{OPA}$		LibCHA$_{CPA}$		Call Edge Reduction	
	Pub.	Pkg.	Total	Pkg.	All	CBS	All	CBS	All	CBS
httpcore	238	16	1 652	62	92 922	41 896	65 104	15 204	29.94%	63.71%
hsqldb	446	141	10 196	1 880	449 786	143 676	342 577	43 060	23.84%	70.03%
spring-tx	168	37	1 108	70	60 808	17 551	48 331	5 595	20.52%	68.12%
groovy-all	2 905	1 497	37 150	1 467	3 125 366	794 454	2 493 057	280 747	20.23%	64.66%
plexus-container-default	182	5	1 142	17	165 670	49 558	133 112	17 000	19.65%	65.70%
scalac-scoverage-plugin_2.11	172	0	1 068	0	440 868	10 180	438 389	7 701	0.56%	24.35%
scala-compiler	8 557	37	59 147	200	14 476 580	629 696	14 408 370	561 514	0.47%	10.83%
scalatest_2.10	6 755	0	82 680	0	7 780 661	305 601	7 749 991	274 931	0.39%	10.04%
jsr305	35	0	30	1	88	0	88	0	0.00%	0.00%
easymockclassextension	7	0	27	0	133	0	133	0	0.00%	0.00%
mean (over all projects)									9.21%	50.06%
stddev (over all projects)									5.79%	15.64%

Table 7.4.: Reduction of call edges from LibCHA$_{OPA}$ compared to LibCHA$_{CPA}$

Project	Types	Methods	Naïve			LibCHA$_{OPA}$			LibCHA$_{CPA}$		
			eps	cg	$\sum$	eps	cg	$\sum$	eps	cg	$\sum$
easymockclassextension	7	27	0.0001	0.0035	0.0036	0.3590	0.6424	1.0014	0.3861	0.6415	1.0276
hamcrest-core	45	275	0.0002	0.2317	0.2319	0.3333	0.8544	1.1877	0.3824	0.8570	1.2394
json	18	128	0.0002	0.2214	0.2216	0.3362	0.8930	1.2292	0.3890	0.8687	1.2577
reflections	96	619	0.0002	0.1908	0.1910	0.3614	0.8317	1.1931	0.4026	0.8191	1.2217
aspectjrt	130	722	0.0002	0.1858	0.1860	0.3544	0.8830	1.2374	0.3885	0.8458	1.2343
groovy-all	4402	37150	0.0086	1.1409	1.1495	0.4074	2.3045	2.7119	0.4684	1.9579	2.4263
gwt-user	5497	46599	0.0092	1.5877	1.5969	0.4193	2.5968	3.0161	0.4786	2.4562	2.9348
scala-library	4899	59519	0.0127	1.4340	1.4467	0.4528	2.5445	2.9973	0.5175	2.3973	2.9148
scalatest_2.10	6755	82680	0.0160	3.2804	3.2964	1.0270	4.5554	5.5824	1.0599	4.9914	6.0513
scala-compiler	8594	59147	0.0578	5.7433	5.8011	1.1644	7.4983	8.6627	0.9313	7.6521	8.5834
mean (over all projects)					0.3826			1.5047			1.5359

Table 7.5.: Measured time for computing the entry point set and constructing the call graph (in seconds).

7.3.2. Discussion

The results of the empirical study are shown in Tables 7.2-7.5. They list only the top 5 and the bottom 5 libraries with respect to the measured effect. Furthermore, each table contains the mean as well as the standard deviation over all 100 libraries. Next, we will use this information to answer the research questions.

RQ1. To answer the first question, whether we need specialized call-graph algorithms for libraries or not, we compare the number of edges in call graphs computed by LibCHA$_{\text{OPA}}$ and the naïve CHA approach. The results are shown in Table 7.2.

In 98 of the 100 libraries, CBS resolution introduces additional edges; in some case the increase is up to 50%. These edges are missing in the graphs constructed by the naïve approach. As discussed, the lack of these edges may prevent security analyses from hinting at potential vulnerabilities. Given that CBS significantly impacts the call graph, we conclude that the naïve approach computes an unsound approximation most of the time; the exception are the projects without instance calls. This clearly supports the claim that specialized algorithms for libraries are needed. Additionally, we observe that the more precise entry point computation leads to less edges in most projects (shown in column *Type Edges* in Table 7.2), but the effect is small.

Obs.13: We observe that naïve approaches compute an unsound approximation when constructing library call graphs. Hence, we require specialized algorithms for libraries.

The experiments also show that CBS resolution does not lead to an explosion of the call edges count. The maven-plugin-api has the most significant increase in the number of call edges because it depends on three other libraries in which a lot of CBS targets are found. For example, the maven-plugin-api defines interfaces that declare methods with the following signatures: `java.lang.String getValue()`, `java.util.Iterator iterator()` or `java.lang.String toString()` and methods with these signatures are often found in the used libraries; in particular in case of `toString()`. Yet, even in this worst case, *only* $\approx 60\%$ of all edges are due to CBS resolution. The mean is only 16%, and from Table 7.4, we can further conclude that the number of CBS edges decreases significantly (up to 70%) when we apply the closed-package assumption.

Obs.14: A significant number of call-by-signature edges originate from methods with generic names. However, applying the closed-package assumption can help to reduce these.

RQ2. To answer this question, we compare the respective precision of the graphs constructed by LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$. We did a quantitative and a qualitative comparison. The former is discussed in the following; the latter in Section 7.4. The quantitative evaluation compares the number of call edges in the respective call graphs shown in Table 7.4.

In all cases—except of the projects without instance calls—the LibCHA$_{\text{CPA}}$ graph contains less edges; sometimes up to 30% less. Surprisingly, we observe differences even in case of the Scala libraries (scala-compiler, scalatest_2.10, etc.). A manual inspection of the code revealed that the libraries contain a minimal amount of Java code, which

uses package and protected visibility. Over all projects the mean call edge reduction is 9.21% and the number of edges that is added by CBS resolution is reduced by 50.06% on average. The latter is due to the fact that a client can not inherit package visible classes or interfaces under the closed-package assumption and therefore the amount of possible subtypes is lower in LibCHA$_{\text{CPA}}$.

Obs.15: Distinguishing between the open-package assumption and closed-package assumption is useful, especially for libraries that use Java's visibility modifiers.

RQ3. To answer this question, we measure how many entry points are identified by LibCHA$_{\text{CPA}}$ w.r.t. LibCHA$_{\text{OPA}}$ and how this affects the call graph.

The reduction in entry points is shown in Table 7.3. We observe a reduction of entry points in 93 projects (in the table only the last five of these seven projects are shown). In the remaining seven cases, the entry point sets are identical; these projects all have in common that they do not declare a single package-visible type and at most one package-visible method. Overall, LibCHA$_{\text{CPA}}$ identifies up to 33% less entry points and the mean reduction of entry points is 8.04%. However, the effort dedicated to precise entry-point computations done as part of LibCHA$_{\text{CPA}}$ are—again—useless, if the library does not make use of package visibility. For instance, as is the case for the Scala language and, therefore, for all Scala libraries in our set.

Obs.16: Discriminating between the open-package assumption and closed-package assumption is useful when computing entry points. However, the usefulness depends on the library's usage of visibility modifiers.

Interestingly, the reduced number of entry points in LibCHA$_{\text{CPA}}$ does not have an effect of the same magnitude on the overall call edges, though we can still observe some effect. For example, for the hsqldb project, we observe a reduction of the number of entry points by 30% but the effect on the call graph is only $\approx$ 2.7%. This is probably due to the choice of the CHA algorithm as foundation of our algorithms; most methods that are not in the initial entry point set are still included in the call graph. It is likely that the better computation of the entry points would have a more significant effect in combination with better, e.g., context-sensitive, call-graph algorithms.

Obs.17: The effect of a reduced amount of entry points seems minor. More research is needed to determine whether this effect originates from using the imprecise CHA algorithm.

RQ4. To understand the performance characteristics of the proposed algorithms, we measured the times to compute the entry points and the call graphs. Instead of analyzing all library dependencies, we consider only the public interface of all third party libraries. Otherwise, the performance and the set of call edges would be dominated by dependent libraries ($\mathcal{L}_{dep}$). For the largest library, 82% of all methods are defined by the used libraries ($\mathcal{L}_{dep}$); in 90% of all cases, the library defines less than 5% of all methods.

The measurements were taken on an Intel i7 (2.4Ghz) with 6GB memory. The results are shown in Table 7.5. As expected, the proposed algorithms are slower than CHA, but still scale well up to very large libraries. The performance of the naïve implementation

outperforms the two library call graphs in average by 1.15 seconds. The additional time is required to perform the more precise computations. However, this overhead is still acceptable given that the resulting call graphs are well suited for library analyses.

7.4. Case Study: Dead Methods in the JDK

To further understand the impact of the proposed algorithms on an analysis that builds on top of them, we conducted a case study. We implemented an analysis that uses a call graph to collect all non-entry point methods that are not called by another method (excluding self-recursive calls). These methods are then reported as being dead. For the case study, we build the analysis on top of the three different call graphs constructed by the naïve algorithm, LibCHA$_{\text{OPA}}$, and LibCHA$_{\text{CPA}}$. The subject library was the part of JDK 1.7.0 update 80 that defines Java's public API, but which also contains library-private code (specifically, we analyzed the code in the packages starting with `java` and `javax`). The results are reported in Table 7.6.

Algorithm	**naïve/LibCHA$_{\text{OPA}}$**	**LibCHA$_{\text{CPA}}$**
Reported Methods	218	2 119
Technical Artifacts	114	114
Swing PLAF related	4	1 325
Potentially Dead	100	680

Table 7.6.: Number of dead methods found in the JDK.

As shown in the second row of the table, the analyses using the call graphs computed by the naïve algorithm and LibCHA$_{\text{OPA}}$ initially reported the same 218 methods, a much smaller number compared to 2,119 methods reported by the analysis on top of the call graph constructed by LibCHA$_{\text{CPA}}$. A manual evaluation of the results revealed that some methods are dead "on purpose". For example, it is a common Java idiom to define a private default constructor to ensure that no instances of the class can be created. This idiom is, e.g., used by `java.lang.Math` and always results in an intentionally dead constructor. We call appearances of this idiom technical artifacts: Adapting the analysis revealed that 114 of the initially reported methods belong in this category (cf. third row in the table).

The manual evaluation further revealed that we must filter out methods in packages starting with `javax.swing.plaf.*`. The respective classes and methods are responsible for the look and feel of Java GUIs and are—as documented in the API—generally instantiated or called by reflection. Given that our case study analysis has no support to identify reflective calls, we decided to consider all methods in the respective packages as being called using reflection, hence not dead. This filtering left us with 100, respectively 680, dead methods reported by the analyses using the naïve or LibCHA$_{\text{OPA}}$-based call graphs, respectively the LibCHA$_{\text{CPA}}$ call graph.

Next, we randomly selected 80 out of the 680 presumable dead methods to perform a manual inspection. From these 80 methods, 40 methods are also reported based on the naïve/LibCHA$_{\text{OPA}}$ based call graph. Which revealed that, 32 out of the 40 methods (80%) were correctly classified as dead. From the remaining 40 methods, one further method was misclassified. Hence, 71 ($\approx$ 89%) out of the 80 reported methods are indeed dead. The majority of the latter methods are non-private methods defined in package visible classes. Some of them were marked as deprecated, some could be clearly identified as left-over debug or test code, some were unused method overloads, and others seemed to be overlooked due to the complexity of surrounding code. In nine cases, we concluded that the reported methods are (most likely) not dead, because they seem to be called from native code or via Java's reflection mechanism.

Overall, we are confident that we found at least 550 ($\approx$ 80%) true dead methods in the core of the Java Class Library using the LibCHA$_{\text{CPA}}$-based call graph.[7] Using a call graph computed by LibCHA$_{\text{OPA}}$ or the naïve call graph construction algorithm, we identified only $\approx 80 (\approx 15\%)$ of these methods.

Obs.18: A simple quality analysis that requires a call graph, such as a dead-method detection, benefits from distinguishing the open- and closed-package assumption.

7.5. Conclusion

In this chapter, we have discussed the design space for call-graph algorithms for libraries. We have in particular discussed the issues related to the use of established call-graph algorithms and have discussed how to adapt the classical CHA algorithm to make it useable for the construction of library call graphs. Constructing call graphs for libraries requires—compared to the construction of call graphs for applications and components—necessarily different algorithms to satisfy the needs of different categories of subsequent analyses. As the evaluation has shown, both algorithms are necessary as the number of call edges in the call graphs differ significantly and each algorithm is able to identify unique issues.

[7] These overall quality results are in line with the results reported by Eichberg et al. in [EHMG15].

8. Modular Collaborative Program Analysis

In this chapter, we present a framework for composable call-graph construction algorithms. Instead of building monolithic algorithms that address several language features and APIs, we propose an approach where various orthogonal analyses for single language features and APIs collaboratively compute a CG. As this framework is part of a more general architecture—reminiscent of blackboard systems [Cor91]—we will present the full approach. Overall, this chapter presents the following contributions:

- A list requirements on frameworks for collaborative static analysis that is distilled from three case studies.
- A novel approach that satisfies all these requirements and advances the state-of-the-art in implementing modular inter-dependent analyses.
- A thorough evaluation of the approach that supports our claims on generality, showcases its modularity features, points out performance improvements over Doop [BS09b], the state-of-the-art declarative framework, and provides promising results for parallelization.

As I already mentioned, while discussing my contributions (cf. Section 1.4.3), the general framework and the escape and purity analyses are joint work others. There I was a collaborator but not the primary contributor. My work concentrates on the modularization of CGs.

8.1. Motivation

Traditionally, static analyses have been implemented in an imperative monolithic style, i.e., one super-analysis computes the results of all sub-problems. Not only do monolithic designs become complex when mutually dependent problems are involved [BS09b]. More importantly, individual sub-analyses cannot be developed in isolation, cannot be reused for other analyses, and cannot easily be added, removed, and exchanged to trade-off between precision, sound(i)ness [LSS+15], and performance in a fine-tuned way, i.e., to enable pluggable precision/sound(i)ness/performance.

To address these requirements, it is desirable to encode solutions for sub-problems of a complex static analysis in separate modules. However, while encoded in independent modules, the execution of inter-dependent sub-analyses needs to be interleaved to enable exchanging intermediate results. The latter is often necessary for optimal precision, as has been proven by the theory of reduced products in abstract interpretation [CC79] and was more recently demonstrated for other kinds of analyses [BS09a, HKE+18, EKH+18].

Recently, declarative approaches to static analysis using the Datalog language [WL04, BS09b, MYL16] are gaining increased popularity—especially in the area of points-to analyses [WL04, BS09b, SBL11, TLX16]. Such approaches nicely support the requirements stated above. Analyses are implemented as sets of rules that are evaluated by an underlying constraint solver. Thus, complex analyses can be broken down into simpler, independently-developed analyses. The underlying solver transparently resolves their dependencies and propagates intermediate updates according to the specified rules, thus enabling interleaved execution. Moreover, the solver can (a) apply analysis-independent optimizations, e.g., by rearranging the computation order (although manual optimization is still necessary [BS09b, SB10]), and/or (b) automatically parallelize the execution [JSS16].

However, using Datalog and giving solvers full control comes with *drawbacks in terms of both performance and generality*. First, it is not possible to exploit analysis-specific knowledge in managing the execution and dependencies of the analyses. Such knowledge can help boost scalability. For example, an imperative purity analysis that determines whether a method is deterministic by, among others, checking the mutability of fields $f_1, ..., f_n$ could drop further checks as soon as any f_i is found to be mutable. A declarative analysis whose execution is driven by a general-purpose solver cannot take this short-cut. Analysis-specific knowledge is also valuable to correctly compose incompatible optimistic and pessimistic analyses (as defined in [GC01, LH03]). Second, the Datalog solver uses analysis-independent data structures and analyses cannot exploit data structures that are tailored for their specific needs. Such optimized data structures, like tries, can be crucial for achieving performance.

Finally, the fully declarative approach fosters a one-size-fits-all style, limiting generality. For instance, by relying on relations, Datalog-based approaches support only set-based lattices, while many common analyses require other kinds of lattices. Constant propagation, e.g., is usually implemented via singleton-value-based lattices, making it infeasible to implement it using Datalog [MYL16, SBEV18].

Next, we discuss the required terminology and then address the these issues of declarative approaches, without compromising on their benefits.

8.2. Background and Terminology

In this section, we shortly introduce blackboard systems and present terminology used throughout the chapter.

Blackboard Systems

Blackboard systems [Cor91] use a central data structure—*the blackboard*—to coordinate the collaborative work of otherwise decoupled *knowledge sources*. The latter contribute (partial) information to the blackboard towards collaboratively reaching an overall goal. The blackboard notifies knowledge sources about availability of new information they might require through a control mechanism that decides which knowledge sources should be executed in what order. The information can then be queried by the knowledge

sources, which execute and produce further information. Each execution of a knowledge source is called an *activation*. For instance, computing the purity of a method m requires the purity information of all callees of m. Whenever new information about these callees is recorded, the analysis for m may be activated. The order of activations is decided by the blackboard.

Terminology

Entity The term is used to characterize anything one can compute some information for. Entities can be concrete code elements, e.g., classes, methods, fields, or allocation sites, but also abstract concepts such as all subtypes of a class. The set of entities is not necessarily defined a priori and can be created on-the-fly while analyses execute.

Property Kind The term characterizes a specific kind of information that can be computed for an entity, e.g., mutability of classes, purity of methods, or callees of a specific method. Each property kind represents one lattice of possible values.

Property The term characterizes a specific value in the lattice of some property kind that is attached to some entity, e.g., a class can be mutable or immutable, a method can be pure or impure, a specific method may invoke a specific set of methods. Per entity at most one property of a specific kind can be computed.

Analysis The term characterizes an algorithm that given an entity computes its property of a certain kind. We say that *an analysis computes a property kind* as a shorthand for "an analysis that computes properties of that property kind for a given kind of entity". Analyses are knowledge sources in the sense of the blackboard architecture; the properties they compute constitute the information that they contribute to and/or query from the blackboard.

8.3. Case Studies

We motivate the need the required features for our modular static analysis framework by discussing case studies involving several interrelated sub-analyses to distill a list of requirements on static analysis frameworks. During the discussion, we *emphasize* concepts whenever they occur. The case studies represent very dissimilar kinds of analyses. In particular, they require different kinds of lattices, including singleton-value lattices (e.g. in Section 8.3.3) and set-based lattices (e.g. in Section 8.3.2). This motivates the first requirement: Static analyses frameworks must support varied domain lattices (**R1**).

8.3.1. Three-address Code

The first case study is an analysis to produce a three-address code representation (TAC) of JVM bytecode, presented in more detail in previous work [RKH+20]. In its basic version, TAC uses def/use, type, and value information (including constant propagation)

provided by an abstract-interpretation-based analysis (AI). To increase precision, AI may be enhanced with analyses that refine type and the value information for method return values and fields. However, such additional analyses may negatively affect the runtime. Hence, systematic investigation of the precision/performance trade-off is needed to decide whether to use such additional analyses on a case-by-case basis. To this end, a separation into modules that can be enabled/disabled is beneficial. In general, we derive the following requirements regarding support for modular pluggable analyses.

For systematically studying precision/soundness/performance trade-offs, static analysis frameworks should support en/disabling inter-dependent analyses (**R2**). To maximize pluggability, analyses should be defined in decoupled modules, and yet be able to collaboratively compute properties (*collaborative analyses*). As individual analyses can be disabled, it should be possible to specify soundly over-approximated *fallback values*[1] for the properties they compute, to be used by dependent analyses in lack of actual results (**R3**).

Moreover, an approach for modular collaborative analyses should support their *interleaved execution* without them knowing about each other's existence (**R4**). Two analyses are executed interleaved, if they can interchange *intermediate results*. This is important for optimal precision [CC79]: knowledge gained during the execution of some analysis A_1 may be used by the execution of some other analysis A_2 on-the-fly to refine its result and, in turn, this may enable further refinement for A_1. The precision of field- and return-value refinement analyses would profit from interleaved executions, as they depend on each other cyclically. If a method `m` returns the value of a field `f`, then the return value of `m` depends on `f`'s value. Similarly, if the value returned by `m` is written into `f`, then `f`'s value also depends on `m`'s return value.

However, interleaved execution must in specific cases be suppressed to ensure correctness. This is the case for the composition of *pessimistic* and *optimistic* analyses. Pessimistic analyses start with a sound but potentially imprecise assumption and eventually refine it. Optimistic analyses start with an unsound but (over)precise assumption and progress by reducing (over)precision towards a sound result. Field- and return-value refinement analyses are pessimistic—the declared return type of method m, say `List`, is a sound but eventually imprecise initial value for the return-value analysis; during the execution, the analysis may find out that m actually returns the more precise result, say `ArrayList`. AI is an optimistic analysis—it starts with the unsound assumption that all code is dead and refines it by adding statements found to be alive towards a sound, but potentially less precise result. Optimistic and pessimistic analyses are *incompatible* for interleaved execution, because they refine the respective lattices in opposite directions. As a result, exchanging intermediate results may cause inconsistencies, thereby violating monotonicity. Thus, the analysis framework must enforce that only *final results* of pessimistic analyses are passed to dependent optimistic analyses (and vice-versa), avoiding interleaving and *suppressing* non-final updates (**R5**).

[1] To minimize the effect of fallback values on precision, it makes sense to compute the fallback by using locally available information, e.g., using declared type information, instead of always returning the same over-approximated value.

For illustration, consider the example of some piece of code, say c, that contains a call to a method m_1 that is mutually recursive with a method m_2, but is conditioned on a field f being an instance of `LinkedList`. To analyze c, we combine a field-value analysis FA, an AI analysis, and a call graph construction algorithm, CG. Assume that FA, which is a pessimistic analysis, initially reports the type of the field f to be `List`. Given this information, AI would optimistically consider c to be live and CG, hence, will consider both m_1 and m_2 to be reachable. Because of the mutual recursion (and also because of the monotonicity requirement), this result cannot be changed later, if FA finds out that f can only contain `ArrayLists`. If, however, the latter information was present when AI analyzed the code, c would have been marked as dead, and CG would have marked m_1 and m_2 as unreachable. Thus, the results of this combination of analyses is non-deterministic and possibly incorrect (imprecise, if m_1 and m_2 are falsely reported to be reachable).

8.3.2. Modular Call-graph Construction

Inter-procedural analyses presume a call graph (CG): Given method `m`, CG provides information about (a) methods that may be invoked at a call site in `m` (callees) and (b) call sites from which `m` may be invoked (callers). We use the CG to motivate the need for supporting further kinds of execution interleaving (beyond **R4**) as well as further requirements. The previous case study illustrated the need for interleaved execution of inter-dependent analyses that calculate different properties and operate on different entities (composition of analyses for refining field and return values with TAC). The CG use case illustrates two further kinds of interleaved execution.

First, we need interleaved execution of multiple instances of the same analysis operating on different code entities to collaboratively compute a single property, whereby each instance contributes partial results (**R6**). For example, different executions of a CG analysis for different callers of a method m need to contribute their *partial results* to collaboratively derive all of m's callers (computing callers of a method is inherently non-local).

Second, we also need to support interleaving of independent analyses that collaboratively compute a single property (**R7**). Consider, e.g., the computation of the callees of `m`. A CG analysis can in principle consider m in isolation. A monolithic analysis for callees is nonetheless not suitable. It makes sense to distinguish between one sub-analysis that handles standard invocation instructions (e.g., CHA [DGC95], RTA [BS96], points-to-based [BS09b] analysis) and sub-analyses dedicated to non-standard ways of method invocation through specific language features, e.g., reflection, native methods, or functionality related to threads, serialization, etc. Non-standard invocation requires specific handling (e.g., one may deliberately not want to perform reflection resolution, or may want to perform it based on dynamic execution traces). By offering such specialized analyses as decoupled modules, they become highly reusable and can be combined with different call-graph analysis for standard invocation instructions. This makes the call graph construction highly configurable for fine-tuning its performance and sound(i)ness. Hence, not only a method's callers but also its callees need to be computed collabora-

tively. This time, different analyses targeting different language features, rather than different executions of the same CG analysis, contribute to the same property.

Handling special language features may even rely on integrating results of external tools or precomputed values (**R8**). For instance, one may choose to integrate the results of TamiFlex [BSS+11] for reflective calls, or external tools for analyzing native methods.

The CG case study also motivates support for specifying precise *default values* (**R9**) (in addition to sound fallback values). For illustration, consider the case of an unreachable method m. The CG analysis will never compute callees or caller information for m. However, this lack of results is an inherent property of the entity, as opposed to being the result of a missing/disabled analysis. An over-approximating fallback value to compensate the deactivation of the CG module for m may have to include all methods and hence be too imprecise. Instead, analyses depending on the CG should get the information that m is unreachable—the precise default value. The developer of the analysis knows such information and should be enabled to tell the framework.

Another requirement is motivated by the CG. The CG construction unfolds along the transitive closure of methods reachable from some entry points. Hence, it does not make sense to execute the decoupled modules collaboratively constructing the CG—each handling a particular language feature—globally on all methods of a program. Instead, they should be *triggered* only when the overall analysis progress discovers a newly reachable method. Hence, the framework must support triggering analyses once the first (intermediate) result for a property is recorded (**R10**).

Our previous work [RKE+19] provides empirical evidence that encoding an RTA sub-analysis and sub-analyses for language-specific features as collaborative interleaved modules, results in more sound call graphs and better performance compared to call graph analyses of the Soot [VRCG+10], WALA [IBM], and Doop [BS09b] frameworks.

8.3.3. Mutability, Escape, and Purity Analysis

The example analyses in this subsection illustrate the need for further kinds of activation modes in addition to triggered analyses, illustrated in the previous subsection: (a) *eager analyses*, which refers to computing an analysis for all entities in the analyzed program, and (b) *lazy analyses*, i.e., executing an analysis A_1 only for the entities for which the property that A_1 computes is queried by some (potentially the same) analysis A_2. A further requirement illustrated by the analyses in this subsection is that the framework should allow analyses to enforce an execution order that overrides the order determined by the solver.

The use case involves analyses for method purity, class and field mutability [PBKM00, HM12], and escape information [CGS+99, KM05]. The latter includes aggregated information on field locality and return-value freshness (cf. [HKE+18]). The analyses in this case study interact tightly and compute properties that may be relevant for both end users (e.g., method purity) and further analyses (e.g., escape information). Complex dependencies exists between all these analyses. To fine-tune the precision/performance trade-off, several analyses for these property kinds with different precision can be exchanged as needed; all are *optimistic* and use TAC and/or the CG information.

Since the results of analyses in this case study may be of interest to the end user, it is useful to compute them for all possible entities eagerly (**R11**), e.g., computing the mutability of all fields in the program. However, when the field mutability is only used to support, e.g., the purity analysis, it may be beneficial for performance reasons to compute it lazily (**R12**), i.e., only for the fields for which mutability is queried by the purity analysis. This illustrates that we need both eager and lazy execution modes. Eager and lazy versions of the same analysis should typically share the code and only be registered with the framework in different ways. The class mutability analysis also illustrates the need to configure the framework with analysis-specific execution orders (**R13**): For performance reasons, it makes sense to analyze classes in a program in a top-down order starting with parent classes before their children.

Our previous work ([HKE+18]) provides empirical evidence for the requirements stated in this section. An implementation of the purity sub-analysis of this case study (and through transitive use, the mutability and escape sub-analyses) as collaborative analyses with interleaved execution showed higher precision, more fine-granular results and similar performance characteristics compared to the then state-of-the-art purity inference tool ReIm [HMDE12].

8.3.4. Interim summary

Table 8.1 summarizes the requirements along the case studies motivating them. Existing frameworks do not satisfy all of them. Imperative frameworks lack support for modularity, especially **R5**, **R6**, and **R7**. Declarative approaches, e.g., Doop [BS09b], have other limitations: By being bound to relations for modeling properties, they lack the ability to express the range of different analyses represented by our case studies (**R1**). They also fail to support sound interactions between incompatible analyses (**R5**). By giving the solver full control, they do not support different kinds of analysis-specific activation modes of analyses (**R10-R13**).

8.4. Approach

Opal is the first static analysis framework to build upon the concept of blackboard systems: Static analysis modules correspond to knowledge sources; the store that stores and manages the computed properties corresponds to the blackboard. Opal combines imperative and declarative programming styles for analyses.

In Opal, the developer of an analysis `A`: (a) implements the lattice representation of the property values computed by `A` (8.4.1), (b) implements two *imperative* functions - so-called *initial analysis function* (IAF) respectively *continuation function* (CF) (8.4.2), (c) declares the property kinds computed by the analysis and its dependencies (8.4.3), and (d) defines how `A`'s results are reported to the blackboard (8.4.4). Guided by the declared dependencies, the blackboard and its fixed-point solver then coordinate the execution of the analyses, thereby (e) ensuring all execution constraints (8.4.5), (f) performing fixed-point computations, whenever circular dependencies are involved (8.4.6), and (g) automatically scheduling and parallelizing the execution of analyses (8.4.7).

Table 8.1.: Summary of Requirements

Lattices and values
R1 Support for different kinds of lattices (8.3.1, 8.3.2, 8.3.3)
R3 Fallbacks of properties when no analysis is scheduled (8.3.1, 8.3.3)
R9 Default values for entities not reached by an analysis (8.3.2)
Composability
R2 Support for enabling/disabling individual analyses (8.3.1, 8.3.2, 8.3.3)
R4 Interleaved execution with circular dependencies (8.3.1, 8.3.2, 8.3.3)
R5 Combination of optimistic and pessimistic analyses (8.3.1)
R6 Different activations contributing to a single property (8.3.2)
R7 Independent analyses contributing to a single property (8.3.2)
Initiation of property computations
R8 Precomputed property values (8.3.2, 8.3.3)
R10 Start computation once an analysis reaches an entity (8.3.2)
R11 Start computation eagerly for a predefined set of entities (8.3.3)
R12 Start computation lazily for entities requested (8.3.1, 8.3.3)
R13 Start computation as guided by an analysis (8.3.3)

8.4.1. Representing Properties

Values of a property kind constitute a lattice structure. In Opal, singleton value-based lattices, interval lattices, or set-based lattices are possible (**R1**). The lattice's bottom value models the best possible value (e.g., pure for method purity); its top value the sound over-approximation (e.g., impure). The lattices must satisfy the ascending (descending) chain condition to ensure termination of optimistic (pessimistic) analyses. When defining a property kind, developers can choose suitable data structures.

Developers can also specify *fallback* and *default* values. The blackboard will return the *fallback value* for some requested property, `p` of property kind `k`, if no analysis is available for `k` (**R3**). As it is a sound over-approximation, the lattice's top value is a good choice. However, the fallback value can also be provided by a "proxy" analysis function that does not query the blackboard, avoiding cyclic dependencies. The blackboard will return a *default value* for `p`, if an analysis is available, but did not produce any result for `p`'s entity (**R9**). For instance, call graph analyses only examine methods reachable from entry points - for any non-reachable method, `m`, a default value can be used to state that `m` is dead and has no (relevant) callees. A sound fallback value would include all possible methods as callees of `m`; thus, in this case, the default value provides more information than a fallback value. If no default value is declared, the fallback value is returned.

Developers implement property kinds by specifying an interface, which can be used to access and manipulate the property values. When the `PropertyKind` trait is extended, the framework assigns an identifier, which can be used to query the blackboard for properties of that kind. Listing 8.1 shows exemplary Scala code of a simple class mutability property kind. Lines 1 to 3 define the base trait for the property kind and give a sound fallback value in line 2. The two possible property values are defined in lines 4 and 5.

```
1 sealed trait ClassMutability extends PropertyKind {
2   def fallback(Type theClass) = MutableClass
3 }
4 case object ImmutableClass extends ClassMutability
5 case object MutableClass extends ClassMutability
```

Listing 8.1.: Example lattice describing class mutability.

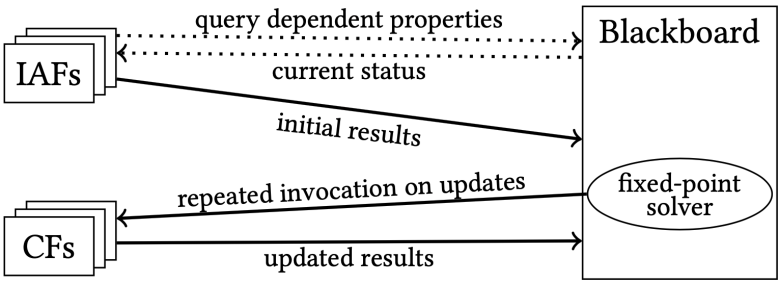

Figure 8.1.: Overview

8.4.2. Analysis Structure

An overview of Opal's analysis structure is shown in Figure 8.1. As already mentioned, Opal's analyses are structured in two parts: An *initial analysis function (IAF)* and one or more *continuation functions (CFs)*. These functions can be implemented in any way, as long as they provide their results as defined by the property kind.

For each entity `e` to be analyzed by `A`, `A`'s IAF is executed. The IAF collects information directly from `e`'s bytecode in order to compute its result. If it needs additional information pertaining to some other entity `e` or from another analysis that computes a property kind `k`, the IAF queries the blackboard for these dependencies, using the identifiers of `e` and `k` to find the relevant information (arrow 1. in Figure 8.1). The blackboard will return the currently available value (2.). This value may, however, not be available, or not final, either because the respective analysis was not yet executed or because it has dependencies that yet need to be satisfied. Once the IAF completes analyzing the entity, it returns to the blackboard (a) a result computed based on the currently available information and (b) any remaining dependencies, along with a continuation function (CF) (3.). Similar to the solver of *declarative* frameworks, the blackboard resolves dependencies and automatically invokes the CFs whenever updates to these dependencies become available (4.). On completion, CFs also return their updated results to blackboard (5.), potentially triggering the execution of other CFs. While the IAF is written imperatively (dotted queries in Figure 8.1), the subsequent execution is performed similar to declarative frameworks (straight lines) by having results declare their dependencies and the solver being responsible to satisfy them. Executions of the IAFs and CFs are called *analysis activations*. To ensure determinism, Opal executes the activations for a single property sequentially, while IAFs and CFs for other properties can execute concurrently.

As analyses get notified about dependency updates through the invocation of the CF, it is not necessary that dependencies are computed before or when they are queried. Instead, they can be computed asynchronously and lazily, i.e., on-demand (**R12**). This also allows OPAL to handle cyclic dependencies (**R4**).

Apart from adhering to this basic structure, developers may use any suitable strategy to implement an analysis `A`. `A` may, e.g., focus on specific statements instead of traversing the entire code of a method (OPAL provides pre-analyses to query specific parts of the code, e.g., all statements that access a specific field). Also, analyses can internally use any data structure suitable to achieve good performance. For illustration, Listing 8.2 shows an excerpt from a simple class mutability analysis' initial analysis function. The IAF is given the entity to analyze (Line 1). Lines 3 to 7 show how to retrieve and handle properties required to compute the IAF's result: The required property (the mutability of an instance field of the analyzed class) is queried from the blackboard (line 3) and based on the returned value, the IAF may compute its result (as in line 4) or keep the dependency in a list of dependees (line 6) to return it alongside an intermediate result later (line 9). Line 9 also specifies the continuation function to be invoked when any of the properties in `dependees` is updated. We do not show the code for that CF here, as its implementation is very similar to lines 4 to 9, i.e., based on the updated value, the (intermediate) result of the CF is determined.

There are two semantic constraints that the implementations of the analyses must satisfy, though. First, they must ensure *monotonicity of result updates* according to the used lattice. Analyses that optimistically start at a lattice's bottom value may only refine approximations upwards; pessimistic analyses only downwards. OPAL can automatically check the monotonicity of updates. Monotonicity allows analyses to know which refinements of intermediate results are still possible. Second, analyses must be *scheduling independent*: Whenever they receive the value of some other property they depend on, they must use the information provided by that value to compute the result of the current activation, i.e., they may not defer the incorporation of the newly gained information to a later activation of a continuation function. This ensures that all available information is used independent of whether the continuation is later scheduled for execution - an activation may never occur in case of cyclic dependencies. For example, once the mutability analysis of a class `C` knows that `C`'s instance field `f` is mutable, it may no longer report that `C` could be immutable. The developer of some analysis `A` must ensure that `A` is scheduling independent.

8.4.3. Declarative Specifications

On top of the IAF and CF, the developer of an analysis `A` specifies (a) the property kinds computed by `A`, (b) its dependencies, (c) on which entities `A` will be executed and (d) when the blackboard should start `A's` execution. These specifications are evaluated when the analysis is registered with the blackboard, before the latter takes over control of analysis activation. When registering analyses, developers may also report precomputed values to the blackboard (**R8**).

```
1  def analyze(Type theClass) = {
2    [...]
3    Blackboard.get(field, FieldMutability) match {
4    case _: MutableField => return Result(theClass,
         MutableClass)
5    case dependee: ImmutableField =>
6      if (!dependee.isFinal) dependees += (field ->
           dependee)
7    }
8    [...]
9    Result(theClass, ImmutableClass, dependees,
         continuation)
10 }
```

Listing 8.2.: Simplified example of a class-mutability analysis.

The specification of the computed property kinds also states whether intermediate results are optimistic or pessimistic and whether the analyses contributes to a collaborative computation or intends to be the only analysis computing the specified property kinds. Dependency specifications state other property kinds on which `A` depends (which `A` queries) and whether `A` can process optimistic/pessimistic intermediate values or final values only.

Analyses can eagerly select a set of entities (e.g., all methods of the analyzed program) if it is likely necessary to perform the analysis for all of these entities (**R11**). This is, e.g., useful for analyses that are of interest to the end user, e.g., if the user is interested in the purity of all methods. Alternatively, analyses can be registered to be invoked lazily [JMT10, Bod18]. Lazy analyses only compute a property if that property is queried (**R12**) by another analysis or by the end user. Finally, an analysis can specify a property kind k such that it is started for every entity for which k has been computed (**R10**).

Some analyses benefit from enforcing a specific order for computing the properties for different entities (**R13**). For instance, the class mutability analysis benefits from traversing the class hierarchy downwards, such that results for a parent class are available before any subclass is analyzed. In Opal, this is supported by enabling the developer of an analysis `A` to declare a number of computations to be scheduled whenever `A` returns a result to the blackboard.

For illustration, Listing 8.3 shows the registration code for a class mutability analysis. Line 1 declares that the analysis optimistically and lazily derives class mutability. Line 2 declares that in performing its computation, it may require field mutability and that it can handle intermediate results for this property if they were computed optimistically. This declaration is complete: No property kinds other than field mutability (and class mutability) may be queried by this analysis. Line 4 registers a predefined value stating that the base class `Object` is immutable (**R8**). The IAF `analyze` is registered as a lazy analysis in line 6, i.e., the mutability of a certain class will only be computed on demand, e.g., when a purity analysis queries it.

```
1 override def derivesLazily =
      Optimistic(ClassMutability)
2 override def uses = Set(Optimistic(FieldMutability))
3 override def register() = {
4   Blackboard.set(Type.Object, ImmutableClass)
5   val analysis = new ClassMutabilityAnalysis
6   Blackboard.registerLazyAnalysis(this,
        analysis.analyze)
7 }
```

Listing 8.3.: An example of registering the class-mutability analysis to the blackboard.

8.4.4. Reporting Results

As already mentioned, analyses write intermediate and final results to the blackboard. They can report results for each single entity individually or for multiple entities at the same time. A result consists of a single lattice value representing the new value for the property or of an update function (UF) for updating the property's current value (as recorded in the blackboard) to incorporate the new result.

A UF is used for properties whose computation is not localized to a specific part of the program, e.g., the callers of a method. For such properties, constraint-based analyses [Aik99, NNH05] have been used in the past; declarative analyses also provide such updates, called deltas, that only specify the change to the property value instead of the full new property value. The `UF` merges the results of one activation to the current state of the property (e.g., add a new caller to an existing set of callers). This way, activations of one or of different analyses can collaboratively contribute to a property (**R6**, **R7**).

8.4.5. Execution Constraints

Once the end user chooses a set of analyses to be executed (**R2**), OPAL uses the declarative specifications (cf. Section 8.4.3) to check and automatically enforce restrictions on analyses that can be executed together. First, it ensures that any property kind is computed by at most one analysis or collaboratively; this is to avoid that conflicting results are reported to the blackboard. Second, if several analyses derive a property kind collaboratively, OPAL ensures that they are all either optimistic or pessimistic. Finally, OPAL ensures that all property kinds required by any analysis are derived by another analysis or there is a fallback value provided; this is to ensure that dependencies can be satisfied.

OPAL's blackboard may run optimistic and pessimistic analyses simultaneously. But, when doing so, it ensures that no intermediate results are propagated between them (**R5**). Given property kind p that is computed optimistically and pessimistic analysis A depending on p, OPAL does not forward any intermediate values of p to A's CF. The latter is triggered only when a value of p is submitted marked as final. We say that the

dependency of A on p is *suppressed*. There are subtle interactions between dependency suppression and cyclic and collaborative computations, which we explain next.

First, there can be no cyclic dependencies between pessimistic and optimistic analyses. The correctness of cyclic dependency resolution relies on the assumption that all intermediate approximations have been processed and no further updates to any property involved in the cycle may happen (cf. Section 8.4.6). This obviously is not the case when updates are suppressed.

The interaction between dependency suppression and collaboratively computed properties is more involved. Assume a collaboratively computed property p_1 that (transitively) depends on another collaboratively computed property p_2 and consider the case when one or more of the transitive dependencies between them is suppressed[2]. In this case, OPAL must ensure that p_2's values are committed as final before p_1's values can be committed as final, too. This ensures that final values have been propagated along the suppressed dependencies. To this end, OPAL derives a *commit order* when checking the execution constraints before executing analyses. The commit order is a partial order between collaboratively computed property kinds: p_1 must be finalized later than any other collaboratively computed property kind p_2 on which p_1 depends when there is suppression between them.

Suppression of intermediate updates can also be used to improve performance: Consider the relation between TAC and AI. Both are optimistic and TAC could use intermediate AI results. But these results are typically not useful, hence, it can be beneficial to use suppression to avoid costly computation of these intermediate results and instead compute the TAC only once on the final AI result.

8.4.6. Fixed-point Computation

Computation is started for the entities selected by eager analyses (**R11**) (cf. Section 8.4.3). Whenever intermediate values for properties are submitted, the blackboard schedules activations of continuation functions, distributing updated results to analyses that depend on them. Additionally, the blackboard starts new computations by invoking the initial analysis function for properties that are requested lazily (**R12**), are triggered by some analyses reaching a certain entity (**R10**), or whenever it is guided to do so by running analyses (**R13**). This process of scheduling IAF and CF activations is performed until no further updates are generated – the blackboard has reached a *quiescent* state. At this point, however, the properties' values may not necessarily be final, as there still may be unresolved dependencies. There are three cases to be considered.

First, an analysis was scheduled for some property kind p, but it did not analyze some entity e, for which p was requested, e.g., because e was not reachable in the call graph. In this case, the *default value* (**R9**) is inserted, which may trigger further computations, until quiescence is reached again.

[2] On a chain of dependencies, more than one may be suppressed. Also, if p_1 depends on p_3 and p_4 and each of those depends on p_2, there is more than one path between p_1 and p_2, on which dependencies may get suppressed.

Second, properties that cyclically depend on each other are not finalized yet. If such properties form a *closed strongly connected component*, i.e., they do not have any dependees outside of the cycle (but other properties may still depend on them), they are now finalized to their current value. By requiring analyses to report their results in a monotonous and scheduling independent way (cf. Section 8.4.2), OPAL guarantees that the cycle resolution is deterministic and sound. Again, further computations may arise from resolving cyclic dependencies (including supplying more default values and resolving further cycles) until quiescence is reached again.

Finally, the blackboard finalizes values for collaboratively computed properties. It respects the *commit order* computed previously (cf. Section 8.4.5): After finalizing a set of collaboratively computed properties, computation is resumed again. Only once quiescence is reached again, the next property kinds, as given by the commit order, are finalized. This is repeated until all collaboratively computed properties have been finalized.

8.4.7. Scheduling and Parallelization

Blackboard systems require a control component that, upon updates of the blackboard, decides which knowledge sources to activate next. In our case, this control component determines the order in which activations of dependent analyses are executed and is called *scheduler*. The order in which dependent analyses are activated can have significant effects on performance [RL11].

OPAL allows for the scheduler to be easily exchanged in order to select the best performing one for any chosen set of analyses. Apart from general strategies such as first-in-first-out, more specific algorithms may use the dependency structure or the values of intermediate approximations to decide the scheduling order. This is similar to the control component of blackboard systems asking knowledge sources for an estimated information gain (cf. [Cor91]).

Blackboard systems lend themselves well to parallelization. The individual knowledge sources, i.e., analyses in our case, are decoupled and their activations (both the initial analysis and the continuations) can be executed in parallel on multiple threads. Updates to the blackboard, on the other hand, can be synchronized on a special thread or, if that becomes a bottleneck, distributed to several threads based on the property kind and/or entity. A simple implementation may consist of several threads that use a shared data structure holding the property data and use locks or other mechanisms to synchronize accesses to this shared storage.

8.4.8. Summary

OPAL's approach fosters strong decoupling of reified lattices (choice of data structures), analyses (choice of algorithm), and the solver infrastructure (the concrete fixed-point solving implementation). This enables exchanging and optimizing these parts independently. As reified lattices are the basis for all communication between analyses, different versions of analyses can be implemented at different trade-offs. The solver manages exe-

cution of analyses, tracks dependencies and propagates updates, performs monotonicity checks, and computes the fixed-point solution.

8.5. Evaluation

We evaluate our approach by answering the following questions:

RQ1 Does our approach support modularization of a broad range of static analysis kinds with varying requirements?

RQ2 Does exchangeability of analysis modules benefit the end user and the developer?

RQ3 Can the framework be parallelized?

RQ4 What is the benefit of analysis-specific data structures?

RQ5 How does the performance of Opal's analyses compare to state-of-the-art declarative approaches?

We implemented our approach on top of the OPAL framework for JVM bytecode analysis [EH14]. However, the approach is framework and language independent. We answer the above research questions using the case studies of Section 8.3 to analyze the DaCapo 2006 benchmark [BGH+06]. We choose DaCapo because Doop, which we compare to in Section 8.5.5, has special support for it. Both the implementation of Opal as well as the case studies are available in the OPAL GitHub repository[3].

All measurements were performed in a Docker container[4] on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPUs and 512 GB RAM. Analyses were run using OpenJDK 11.0.5+10 (64-bit) with 32 GB of heap memory and Scala 2.12.9. Experiments were run seven times and we report their median runtime. We report only excerpts of the results here[5].

8.5.1. Support for Various Analyses

To answer **RQ1**, we implemented the case studies from Section 8.3 using Opal and argue that these are representatives of different analysis kinds. The first case study represents pessimistic analyses in the context of improving precision of a three-address code representation (TAC)—it shows how basic analyses can be extended by analyses that are specialized to increase the precision of sub-problems' solutions. The modular call graph of the second case study involves tightly interacting yet decoupled analyses (e.g., points-to and call graph) and demonstrates how one can plug in further modular analyses that handle special cases of Java in order to increase the call graph's soundness. The third case study introduced several exchangeable analyses for different high-level properties (immutability, escape information, purity). The individual analyses are relatively simple

[3] `https://github.com/stg-tud/opal`.
[4] `https://doi.org/10.5281/zenodo.3872848`.
[5] The entire results can be found here: `https://doi.org/10.5281/zenodo.3972736`.

Table 8.2.: Purity results for different configurations (hsqldb).

Configuration	#Pure	#SEF	#Other	#Impure	Analysis
$PA_2/FMA_1/E_1$	417	482	245	2 635	2.42 s
PA_2/E_1	363	536	245	2 635	2.40 s
$PA_2/FMA_1/E_0$	417	481	241	2 640	1.93 s
PA_2	362	504	225	2 688	0.98 s
PA_1/FMA_1	415	431	0	2 933	0.93 s
PA_0/FMA_1	104	0	0	3 675	0.70 s
PA_0	100	0	0	3 679	0.13 s

and can focus on their respective property, but by using the results of other analyses, they can be more precise than a corresponding monolithic analysis of medium complexity.

As discussed in Section 8.3, to achieve this modularity, several requirements need to be satisfied (cf. Table 8.1). Section 8.4 already explained how Opal supports all of them. On the contrary, as we argue in Section 8.3.4, no current imperative or declarative framework supports all these requirements.

We additionally implemented a solver for *inter-procedural, finite, distributive subset problems* (IFDS) [RHS95], a well-known general framework for dataflow problems based on graph reachability. Similar to other IFDS solvers, e.g., Heros [Bod12], users provide a domain for their dataflow facts and four flow-functions that together specify the IFDS problem. The solver starts one computation per pair of method and entry dataflow fact and these tasks need to communicate their results. We chose IFDS as it is a general framework that allows implementing many dataflow analyses and it is dissimilar from the three case studies' analyses. In particular, it shows Opal's support for implementing general solvers as individual analyses.

Obs.19: Opal's programming model enables the implementation of dissimilar analyses, fostering their modularization into a set of comprehensible, maintainable, and pluggable units. Opal is the only static analysis framework satisfying all requirements from Section 8.3.4.

8.5.2. Effects of Exchangeability of Analyses

Opal strictly decouples property kinds from analyses that compute them. Thus, it can provide different analyses computing the same property kind to cover a wide range of precision, sound(i)ness, and performance trade-offs. Two experiments examine how this exchangeability fosters rapid probing, thus benefiting the analysis' developer and end user alike (**RQ2**): We explore the impact on precision in one experiment and impact on soundness in the second.

In our first experiment, we run various configurations of our purity analysis (cf. Section 8.3.3) with different supporting analyses for field mutability or escape information with different precision-scalability trade-offs. No other tool supports similar exchange-

Table 8.3.: Results for different call-graph modules for Xalan.

Configuration	#Reachable Methods	#Edges	Analysis
RTA	6 141	46 946	8.58 s
RTA_C	6 162	47 154	8.76 s
RTA_R	8 404	63 821	10.07 s
RTA_X	12 937	106 516	12.99 s
RTA_C_X	12 958	106 743	12.86 s
RTA_S_T_F_C_X	12 970	106 778	13.35 s

C=Configured native methods; R=Reflection; X=Tamiflex;
S=Serialization; T=Threads; F=Finalizer;

ability of interacting purity, mutability, and escape analyses. Table 8.2 shows the results for *hsqldb*. Higher indices indicate more precise analyses. Comparing the least precise analysis PA_0 with the most precise $PA_2/FMA_1/E_1$, we observe a reduction in the number of reported impure methods by ≈28%, but a runtime slowdown by 18.6x. Some configurations even have a large impact on runtime for almost no gain in precision, e.g., comparing the most precise one with that using simpler escape analysis E_0.

In the second experiment, we evaluate the RTA call graph with different supporting modules for different JVM features. While DOOP computes call graphs and offers some modularity, e.g., for reflection, no other tool so far includes such fine-grained modules for call graphs. Also, DOOP does not support RTA, but points-to based call graphs only. Results for *Xalan* are shown in Table 8.3, displaying the active modules, the number of reachable methods (RMs), call edges, and respective construction time. While some configurations discover more methods/edges than others, they may discover different sets of methods/edges. A configuration is only guaranteed to be strictly more sound if it uses a strict superset of modules. Compared to the baseline, RTA with support for preconfigured native methods (RTA_C), reaches 21 more methods and ≈200 more call edges. Reflection support (RTA_R) brings over 2 000 more RMs and 16 000 call edges; at the same time, construction time increases by about 15%. Using the Tamiflex (RTA_X) module instead increases call graph size (and soundness) more but introduces further slowdown. With all modules enabled, we reach 111% more methods and 127% more call edges, at the cost of a 55% increased runtime. Moreover, the data suggests that different modules benefit different projects. Tamiflex impacted *Xalan* and *jython*, reflection *fop*, and serialization *hsqldb*. Thus, which modules are more relevant than others may differ between different programs and it may be worth investigating tradeoffs even at the level of individual projects.

Overall, both experiments confirm that Opal maintains exchangeability benefits from Datalog-based analyses, while generalizing these results to a broader range of lattices.

Obs.20: Opal facilitates systematic investigation of different configurations, supporting users and developers in finding the best trade-off between precision, sound(i)ness, and scalability.

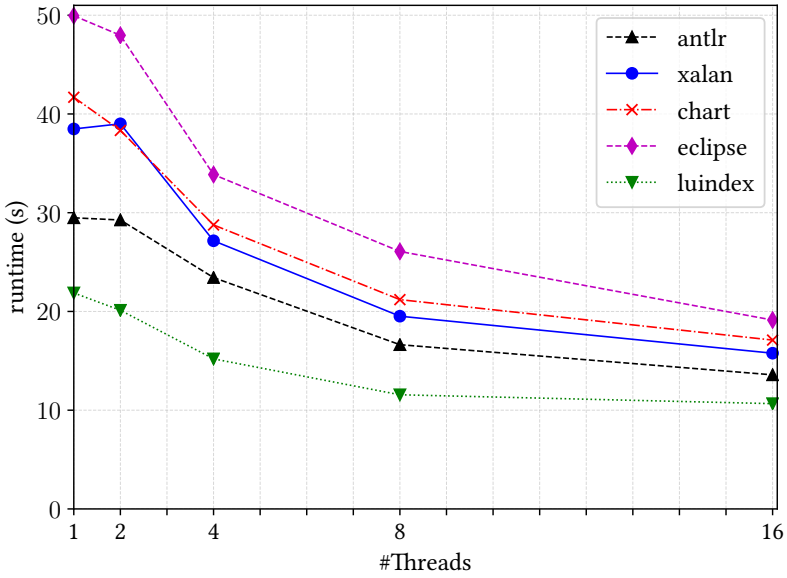

Figure 8.2.: Performance graph of Opal's parallel architecture.

8.5.3. Parallelization

We implemented a proof-of-concept parallel version of our blackboard control (**RQ3**). Using this, we measured the execution time for the points-to-based call graph with different numbers of threads. Results for five DaCapo projects are shown in Figure 8.2. The projects were selected to have similar runtime to facilitate graph readability, the other projects show similar behavior. Benefits of parallelization over one thread appear at two to four threads and we achieve speedups of up to 2x for 16 threads. Beyond this, further improvement is negligible; instead, it slightly decreases due to growing communication overhead. These results are encouraging, given that the parallel version is not at all optimized. An optimized version of it is expected to scale better. Designing such an optimized version requires further research to identify the optimal way to parallelize the computation.

Obs.21: Opal's computation can be parallelized and that parallelization holds potential for increased performance.

8.5.4. Benefits of Specialized Data Structures

To answer **RQ4**, we compare two versions of the same points-to based call-graph algorithm. Both encode points-to, caller, and callee information as integer values. The first version uses specialized trie-based data structures, the second one uses standard Scala sets.

Results are given in the sixth and last column of Table 8.4. Due to its high memory consumption, we had to run the version using Scala's data structures with 128 GB

Table 8.4.: Runtime and size of points-to-based call graphs.

	Doop				Opal		Opal
Project	Compile	Facts	Analysis	#RM	runtime	#RM	(Scala)
antlr	107 s	35 s	41 s	8 402	28.36 s	8 653	305.90 s
bloat	109 s	21 s	33 s	9 644	34.43 s	10 000	266.08 s
chart	109 s	38 s	45 s	12 058	40.13 s	12 268	516.37 s
eclipse	109 s	19 s	17 s	7 163	44.89 s	13 429	343.69 s
fop	110 s	41 s	35 s	7 300	18.87 s	7 509	56.64 s
hsqldb	109 s	38 s	32 s	7 097	19.65 s	7 455	55.69 s
jython	108 s	24 s	90 s	12 901	77.65 s	13 161	3 341.62 s
luindex	108 s	21 s	19 s	7 608	19.34 s	7 972	62.57 s
lusearch	108 s	21 s	20 s	8 281	21.03 s	8 540	70.55 s
pmd	109 s	39 s	36 s	8 817	21.47 s	9 028	75.47 s
xalan	108 s	37 s	30 s	7 111	35.59 s	13 330	246.97 s
geo. ∅	108.54 s	29.09 s	32.51 s		29.68 s		191.26 s

of heap space; *jython*'s analysis even required 256 GB. Using tailored data structures, Opal's runtime decreased by 65% to 98% compared to naively using Scala's sets.

Obs.22: Selecting suitable data structures adapted to the specific analysis needs is an important factor for analysis performance. While the analysis developer can freely select optimized data structures in Opal, strictly declarative approaches do not support such choices.

8.5.5. Comparison with Declarative Approaches

After evaluating individual unique features of Opal in isolation, we present the results of an experiment that directly compares the performance of Opal with that of Doop [BS09b] (**RQ5**) - a highly optimized state-of-the-art tool for declarative Java points-to and call-graph analyses on top of the Soufflé [JSS16] Datalog engine. Its declarative approach assembles a fair comparison as it supports similar modularity and configurability and good trade-offs between pluggable precision/recall. Also, Doop's and Soufflé's authors repeatedly claimed its good performance [BS09b, SB10, BS09a, JSS16]. Specifically, we compare our points-to based call-graph's runtime from Section 8.3.2 to Doop's.

For better comparability, we disabled the reflection support in both tools, because the respective approaches are different. The applications were analyzed together with OpenJDK 1.7.0_75 (used for the TamiFlex data in Doop's benchmarks). Minor differences (less then 5% difference in the number of RMs, except for eclipse and xalan) remain, but

these are in Doop's favor, since they result in more work to be done by Opal[6]. Still, the sixth column of Table 8.4 shows that our complete analysis, including all preprocessing, is often faster than Doop's analysis (9% in the geometric mean). Further, Doop additionally requires time for rule compilation and fact generation.

We used Opal's single-threaded implementation since it seems that Doop is hardly parallelized (fact generation was done with 128 threads, but did not significantly vary with other values for the `fact-gen-cores` parameter and the `souffle-jobs` parameter did not show any effects). Using a parallel version, Opal should be able to outperform Doop even more as shown in Section 8.5.3.

Obs.23: Despite being more general, i.e., not tuned for points-to analyses but supporting many different kinds of analyses, Opal clearly outperforms Doop.

8.6. Threats to Validity

One threat to the validity of our evaluation is the use of the relatively old and small DaCapo benchmark. It is, however, widely used to evaluate Doop [BS09b] and to compare other approaches with Doop [AL12, AL13, PM14, TLX16]. Doop's special support for the benchmark makes it a particularly fair evaluation set. Furthermore, our experiment design, based on relative comparisons, should yield the same results with any well-assembled benchmark.

Also, our results are threatened if our points-to analysis is not sufficiently similar to Doop. To achieve comparability, we tailored our points-to analysis to be as similar as possible, i.e., the call graph derived from the points-to results should be almost identical. In order to ensure this, we systematically studied Doop's Datalog rules, validated the resulting call graphs using Judge [RKE+19] and manually inspected points-to sets from deviating call graphs. As a result, our analysis produces always call graphs with slightly more—never less—reachable methods and call edges.

8.7. Related Work

In this section, we discuss several related approaches in various areas of static analysis as well as in blackboard systems.

8.7.1. Blackboard Systems

The blackboard metaphor was introduced by Newell [New62] and implemented for speech-recognition in HEARSAY-II [EHRLR80]. Blackboard systems were used for image recognition [LDDC+95], vessel identification [NFA82], or industrial process control [DCYB09]. For these domains, no efficient, deterministic algorithm is known, leading

[6]For instance, Opal does handle some cases of reflection more soundly even with reflection handling disabled in order to process the DaCapo benchmark correctly.

to several problems mentioned by Buschmann et al. [BMR+96]: nondeterminism makes testing difficult, good solutions are not guaranteed, performance suffers from wrong hypotheses, and development effort is high due to ill-defined domains. As static analyses have a well-defined domain and deterministic algorithms, these do not apply to our approach.

The structure of blackboard systems is described, e.g., by Nii [Nii86], Craig [Cra88], and Corkill [Cor91]. Corkill also discusses concurrent execution of knowledge sources and the control component [Cor89], similar to OPAL. OPAL resembles a more modern interpretation of blackboard systems [Cra93]: its blackboard is not hierarchical and analyses may keep state between activations. Information is, however, never erased and all communication is done via the blackboard.

Brogi and Ciancarini used the blackboard approach to provide concurrency for their Shared Prolog language [BC91]. Like static analyses, this domain is well-defined. Their knowledge sources are restricted to be Prolog logical programs, while OPAL's analyses can be implemented in a way best suited to the analysis needs.

Decker et al. [DGHL91] discuss the importance of heuristics for scheduling concurrent knowledge source activations. Focusing on static analyses and well-defined dependency relations, OPAL provides good general heuristics which are agnostic to individual analyses.

8.7.2. Abstract Interpretation

Cousot et al. [CC79] have proven that multiple (possibly cheap) abstract domains (i.e., analyses) can be combined using the reduced product to increase overall precision. In abstract interpreters, such as Astrée [CCF+06] or Clousot [FL10], dependencies between domains are restricted by the execution order. Thus, the same program statement must be analyzed multiple times which is superfluous with OPAL's explicit dependency management. Also, abstract interpretation typically aims to compute abstract *approximations* [CC77] of concrete values, such as an integer variable's value. OPAL further allows natural expression of analyses on all granularity levels. Keidel et al. [KPE18, KE19] provide modular and reusable abstract semantics for different language features allowing soundness proofs from composition of already sound components. The analyses again approximate single concrete program values. OPAL supports analyses to be based on abstract interpretation and includes such analyses, but generalizes to a much broader range of static analyses.

8.7.3. Declarative Analyses Using Datalog

Datalog is often used to implement static analyses in a strictly declarative fashion [Rep95, WL04, LWL+05a, HVDM06, Wha07, EKKM08]. Properties are represented as relations and rules specify how to compute them. This enables modularization, as rules can be easily exchanged and/or added (e.g. for new language features). The DOOP [BS09b] framework, building on top of the highly optimized Datalog solver Soufflé [JSS16], has shown that the rule-based approach enables precise and scalable points-to analyses. For

this reason, Doop became the state-of-the-art for such analyses [SBL11, KS13, SBKB15, TLX16, TLX17]. Datalog-based frameworks, however, are limited in their expressiveness by using relations, i.e., set-based abstractions, to represent all analysis results. Opal's approach combining imperative and declarative features provides similar benefits as Datalog-based approaches, while allowing for more expressive ways to represent data and to implement analyses.

Datalog's limitation to relations has also been pointed out by Madsen et al. [MYL16]. They propose Flix to overcome this using a language inspired by Datalog and Scala to specify declarative pluggable analyses using arbitrary lattices as in Opal. However, Flix focuses on verifying soundness and safety properties of static analyses and not on performance. For instance, Flix does not allow optimized data structures or scheduling strategies. We wanted to compare our approach against Flix and contacted the authors, but they answered that their IFDS implementation is dysfunctional now and suggested comparing against Doop with the Soufflé engine, which we did in Section 8.5.5. Szabó et al. [SBEV18] also extend Datalog to allow arbitrary lattices for static analysis. Their solver IncA focuses on incrementalization. Opal allows optimizations, e.g., of used data structures or scheduling strategies. Furthermore, analyses' coarser granularity compared to individual rules reduces overhead in parallelization.

8.7.4. Attribute Grammars

Attribute grammars [Knu68] used in compilers such as JastAdd [EH07] enable modular inference of program properties by adding computation rules to the nodes of a program's abstract syntax tree (AST). In traditional attribute grammars, attributes may only depend on parent, sibling, and child nodes. Circular reference attribute grammars [Far86, Jon90, Hed00, MH07] enable attributes to depend on arbitrary AST nodes and allow circular dependencies. Still, analyses are tightly bound to the AST, impeding natural expression of analyses based on different structures, such as a control-flow graph. Similar to Opal, JastAdd enables pluggability for new language features. However, JastAdd requires at least one attribute in a cyclic dependency to be marked explicitly, while Opal handles this transparently.

Öqvist and Hedin [ÖH17] proposed concurrent evaluation of low complexity attributes in circular reference attribute grammars. Opal on the other hand supports arbitrary granularity of concurrent computation. Opal's explicit dependency management enables analyses to drop dependencies and commit final results early for improved performance. Finally, as memorization of properties is done in Opal's blackboard, temporary values are garbage collected automatically, whereas JastAdd requires explicit removal.

8.7.5. Imperative Approaches and Parallelization

Lerner et al. [LGC02] proposed modularly composed dataflow analyses which communicate implicitly through optimizations of the analyzed code or explicitly through *snooping*. A fixed-point algorithm repeatedly reanalyzes the code, while Opal's explicit dependencies avoid reanalysis. They support only dataflow analyses, while Opal enables a wide

range of analyses including dataflow analyses.

CPAchecker [BHT07] is a tool for configurable software verification and analysis. For any combination of analyses, CPAchecker requires defining a compound analysis to integrate results of individual analyses and manage their interaction. For CPA+ [BHT08], combined analyses must work with the same domain and provide an explicit measure of result precision. In contrast, Opal enables tight interaction and interleaved execution of independently-developed analyses without requiring a compound analysis or explicit measure of precision.

Johnson et al. [JFB+17] present a framework for collaborative alias analysis. Clients ask queries which are processed by a sequence of analyses. Each analysis can answer the query or forward it to the next one. Analyses can also generate additional (premise) queries. To ensure termination, a complexity metric must be defined and premises must be simpler than the queries they originate from. Therefore, cyclic dependencies, required for optimal precision, and results combined from different analyses are not supported.

Parallel execution of static analyses is performed by Magellan [EMK+06]. In this framework, dependencies are given by the data processed instead of explicitly by the analyses.

Haller et al. [HGES16] concurrently execute tasks based on lattices and apply this to static analysis. Their framework requires dependencies to be managed fully by the client while Opal manages them automatically based on declarative specifications. In recent work [HKK+20], we extended this approach to support mutable state and found that exchangeable scheduling strategies significantly impact performance. Both concepts are supported in Opal.

8.8. Conclusion

We presented a novel approach for modular collaborative static analyses implemented in the Opal framework. Like with declarative frameworks such as Doop, Opal's analyses, while developed in isolation, can be quickly composed to complex analyses by collaboratively computing results during interleaved executions. Sub-analyses can be reused in various complex analyses, and one can easily exchange sub-analyses of complex analysis for fine-tuning precision, sound(i)ness, and performance.

Instead of relying on a general-purpose solver, Opal combines imperative and declarative features to overcome limitations of fully declarative frameworks. Individual analyses can be implemented in imperative style using whatever data structures and implementation strategies are appropriate for their specific needs. Interdependencies and other characteristics necessary for guiding their interleaved execution are declaratively specified and automatically managed by a custom solver resembling a blackboard architecture. Due to its approach, Opal (a) is more general in terms of the analyses supported—it is, in particular, the first framework to explicitly support lazy collaboration of optimistic and pessimistic analyses—and (b) enables analysis-specific optimizations, which lead to outperforming state-of-the-art declarative analyses.

Whereas the approach itself is generally applicable, it also exhibits enormous benefits

for the modular construction of call graphs. Section 8.5.2 demonstrated Opal's capabilities to reconfigure a call graph quickly. This pluggability enables both a) easy adaption of a call-graph algorithm to the needs of an individual program and b) exchanging single analyses to find the best trade-off between precision, scalability, and sound(i)ness. Besides, our study from Chapter 6 showed that many features remain unsupported, and Opal's modular design allows for prototyping and testing new feature abstractions in isolation.

9. TACAI: An Intermediate Representation based on Abstract Interpretation

All Java static analysis frameworks presented in Section 3.1 rely on an intermediate presentation (IR) of Java Bytecode to facilitate the development of static analyses. Initial observations from Section 6.2.3 reveal that especially Soot's Jimple, WALA IR, and Opal's IR apply different optimizations during the transformation from bytecode to the IR such that the available type information varies in precision. That lead to enormous differences in the precision of the framework's RTA call graphs. Therefore, we want to further investigate the differences within different IRs and their affect on CG construction.

This chapter presents the design and implementation of TACAI, an intermediate code representation based on abstract interpretation with exchangeable domains. We implemented TACAI as IR of OPAL. In the previous chapter, we presented TACAI as one of the case studies (cf. Section 8.3). Exchanging the abstract domains TACAI uses for the abstract interpretation allows to enrich the resulting IR with additional information, e.g., more precise type information for a callsite's receiver objects. We will discuss and evaluate how switching the used abstract domains affects the bytecode-to-TACAI transformation. Furthermore, we compare TACAI to Shimple, a well-established IR provided by Soot. We chose Shimple because it is an SSA-based TAC representation and is thus closer to TACAI than Jimple. WALA's IR does not provide any refined type information over the types available directly in the bytecode.

9.1. Approach

TACAI, our approach for a three-address code (TAC) representation, is based on the results of an intra-procedural abstract interpretation (AI) of a method's bytecode. This features two main properties: First, it enables intermediate-representation (IR) derivation at different precision levels by exchanging the underlying domains. Second, all information is computed at the same time in one step, which offers improved performance when compared to classical compiler frameworks that typically compute comparable information in a step-wise manner [Muc97]. In this step-wise approach, collected information is oftentimes not shared between steps and, therefore, recomputed to reduce dependencies. While performing the AI, Opal always computes the method's control-flow graph (CFG) and def-use/use-def information on-the-fly. Therefore, the CFG and def-use information immediately benefit from better domains and lead to simpler and less IR code. The CFG and def-use information are also made explicit in TACAI.

```
1  trait TypeLevelDomain extends Domain
2  with DefaultReferenceValuesBinding
3  with DefaultTypeLevelIntegerValues
4  with DefaultTypeLevelLongValues
5  with TypeLevelLongValuesShiftOperators
6  with TypeLevelPrimitiveValuesConversions
7  with DefaultTypeLevelFloatValues
8  with DefaultTypeLevelDoubleValues
9  with TypeLevelFieldAccessInstructions
10 with TypeLevelInvokeInstructions
```

Listing 9.1: Example `TypeLevelDomain` configuration.

We reuse Opal's domains starting with those operating at the type level, which lead to an IR that has similar precision as Soot's Shimple representation. However, Opal also provides domains that enable constant propagation and constant folding for primitive types. For reference values, there are domains that, e.g., precisely track the nullness, provide must-alias information, compute intersection and union types, or resolve local `Class.forName` calls. Using these domains enables the computation of a more precise IR when compared to typical IRs offered by other frameworks. Furthermore, it is possible to tailor the precision at a fine-grained level to one's needs.

Opal uses Scala's mixin-composition to configure the AI and to implement the semantics for different sets of instructions. The default, namely TACAI^{L0}, performs all operations at the type level and is shown in Listing 9.1. The semantics for each set of closely related instructions is implemented by one specialized trait. Opal provides one trait for `integer`, `long`, `float`, and `double` based computations, one for method invocations, one for field accesses, and one for reference-value-based operations. The latter handles, e.g., `instanceof` checks, casts, and tests against `null`. Interactions between the traits are facilitated by requiring the implementation of a shared set of query methods. For example, every implementation that handles reference values has to implement the globally defined method to test if a value is `null`. The result of these methods is typically a three-state answer: Yes, No, or Unknown. For example, the method returning a reference value's nullness is used by the domain, which is responsible for handling method calls. The latter checks—for each method invocation—if the receiver is `null`. If the receiver is known to be `null`, the target method is not invoked, and a `NullPointerException` will be thrown instead. If the answer is `Unknown`, the behavior can further be configured such that only the call is considered or additionally an exception is considered to be thrown.

Besides the TACAI^{L0} configuration, two further configurations for a more precise TAC are preconfigured. In the first one (TACAI^{L1}), the `DefaultReferenceValuesBinding` (Line 2) is exchanged for an implementation that computes intersection and union types as well as must-alias information for reference values. Furthermore, special support for calls of the native method `System.arraycopy` is provided, which checks for the non-nullness of the arrays and also validates the range that is to-be-copied. If this validation

fails, appropriate exceptions are thrown, which have to be correctly represented.[1] Lastly, constant folding and propagation is performed for integer values by exchanging the `DefaultTypeLevelIntegerValues` (Line 3) domain. The latter is required to identify `if` statements where the conditions evaluate to constant values and are therefore useless.

The most precise configuration (TACAI^{L2}) builds on top of TACAI^{L1} and additionally performs method inlining for monomorphic calls. This is useful, e.g., for builders (e.g. `StringBuffer`), which provide a fluent interface enabling call chaining by always returning the current instance. In such cases, it is then possible to determine that all calls actually happen on the same instance. For that, Scala's stackable trait pattern is used to adapt the handling of method invocations, i.e., an additional trait is configured.

Table 9.1 shows TACAI's output for method `m` (cf. Listing 9.2) at all three levels. TACAI^{L0} almost directly reflects the bytecode: The type of the variable `p1` (Line 2) is considered to be `Cloneable` after the cast operation. The code also contains the (useless) reference comparison (Line 7), comparing the newly created `StringBuffer` (Line 4) with the reference returned by the `append` call (Line 6). TACAI^{L1} is able to correctly identify that `p1`'s type is `Serializable with Cloneable`. This intersection type significantly restricts the set of subtypes when compared to the previous version. Additionally, both `p1` and `lv4` are found not to be null: `p1` because of the explicit nullness check (Line 0), the second because it is freshly allocated (Line 4). This guarantees that the invocations on `p1` (Line 3) and `lv4` (Lines 6 and 10) will not cause `NullPointerException` s. Although the chosen domain is able to track must-alias information intra-procedurally, the (useless) reference comparison is still found in the TAC. The identification of the must-alias relation in this case requires to know that the value returned by `append` is the self-reference `this`. By performing inlining, as done when computing the TACAI^{L2}, this information becomes available and, therefore, the useless comparison can be removed and subsequently, the `if` statement is removed as well as the `throw` statement. A `NOP` statement (TACAI^{L2} Line 7) is added because the CFG is not rewritten during the initial transformation, which requires that every basic block contains at least one instruction. It is straight-forward to remove NOPs and update the CFG in a second step.

```
1  RuntimeException e() { return new RuntimeException(); }
2  void p(String s) { System.out.println(s); }
3
4  void m(Serializable serializable) {
5    if(serializable == null) return ;
6    Object o = (Cloneable) serializable;
7    String s = o.toString();
8    StringBuffer sb0 = new StringBuffer();
9    StringBuffer sb1 = sb0.append(s);
10   if(sb0 != sb1) throw e();
11   p(sb0.toString());
12 }
```

Listing 9.2: Java code used to generate TACAI.

[1] Special handling is provided for `System.arraycopy` because it is by far the most widely used native method in the JDK.

Table 9.1.: Transformed TACAI bytecode from Listing 9.2 using OPAL's *Level 0*, *Level 1*, and *Level 2* domains. Blue lines mark differences compared to lower levels. Light-blue lines are only syntactic changes.

TACAI^{L0}	TACAI^{L1}	TACAI^{L2}
void m(Serializable) {	void m(Serializable) {	void m(Serializable) {
0: if(p1 ! = null) goto 2	0: if(p1 ! = null) goto 2	0: if(p1 ! = null) goto 2
1: return	1: return	1: return
2: (Cloneable) p1	2: (Cloneable) p1	2: (Cloneable) p1
p1 ¡: Cloneable	*p1 ¡: Serializable with Cloneable*	*p1 ¡: Serializable with Cloneable*
	p1 not null	*p1 not null*
3: lv3 = p1.toString()	3: lv3 = p1.toString()	3: lv3 = p1.toString()
4: lv4 = new StringBuffer	4: lv4 = new StringBuffer	4: lv4 = new StringBuffer
	lv4 not null	*lv4 not null*
5: lv4.¡init¿()	5: lv4.¡init¿()	5: lv4.¡init¿()
6: lv6 = lv4.append(lv3)	6: lv6 = lv4.append(lv3)	6: lv4.append(lv3)
		/* expression value ignored */
7: if(lv4==lv6) goto 10	7: if(lv4==lv6) goto 10	7: ; /* NOP */
8: lv8 = p0.e()	8: lv8 = p0.e()	—
9: throw lv8	9: throw lv8	—
10: lva = lv4.toString()	10: lva = lv4.toString()	8: lv8 = lv4.toString()
11: p0.p(lva)	11: p0.p(lva)	9: p0.p(lv8)
12: return	12: return	10: return
}	}	}

9.2. Evaluation

Next, we evaluate the costs and benefits of our IRs along the following four dimensions:

RQ1 How does computing TACAI affect the performance; the time required to compute the IR?

RQ2 How does TACAI affect the overall number of three-address code statements?

RQ3 In how many cases is it possible to provide more precise receiver-type information when compared to the representation offered by the Soot Framework?

RQ4 How does exchanging domains affect the precision of subsequent analyses; in particular call-graph algorithms?

Setup. We perform three experiments to answer the above questions. We analyze five programs with `main` method from the XCorpus [DSST17]: jasml, javacc, jext, proguard, sablecc. This is necessary for the call graphs in the third experiment. All measurements are taken on a Mac Pro with a Xeon E5 with 8 cores@3GHz and a JVM with 24GB of heap space.

Experiment 1. The first experiment aims to answer RQ1 and RQ2 and evaluates how exchanging the abstract interpretation domains affects TACAI's output and its transformation performance. In order to compare the results, we generate Shimple, TACAI^{L0}, TACAI^{L1}, and TACAI^{L2} (cf. Section 9.1) for all methods of the programs that are subject to our evaluation.

Table 9.2 shows the experiment's results. The first three columns show the analyzed project, the number of its classes and methods, respectively. Column four shows the IR to which the values in columns five to ten belong to. Those columns present the total number of instructions, the average instruction count per method, its median, and standard deviation. The last column presents the time it takes to generate the IR.

Comparing the runtimes reveals that TACAI^{L0}, TACAI^{L1}, and TACAI^{L2} are computed significantly faster than Shimple. Only on *javacc*, TACAI^{L2} was slower than Shimple. The best speedup w.r.t. Shimple of roughly 4.5× is achieved on *proguard*. We conclude that TACAI's design is feasible. TACAI can be generated faster than Shimple, even using the most precise configuration TACAI^{L2}. The overhead to compute TACAI^{L1} compared to TACAI^{L0} is negligible. Computing more precise information takes time. However, when the extra information (e.g. nullness) provided by TACAI^{L1} and TACAI^{L2} are required by an analysis, this time consumption is justifiable.

When we consider the number of instructions, its reduction is less than 1 %. This is, however, expected because if it would be otherwise, it would indicate dead code [EHMG15].

Obs.24: Combined with the applied optimizations, the reduction of three-address statements is negligible for our evaluation programs.

Table 9.2.: Performance results from Experiment 1.

project	#classes	#methods	representation	#instructions	avg.	median	st. dev.	#call edges	runtime
jasml	**50**	**265**	Shimple	-	-	-	-	5 792	7.6s
			TACAI^{L0}	14 164	53.5	12	307.5	5 195	3.5s
			TACAI^{L1}	14 163	53.5	12	307.5	5 065	3.9s
			TACAI^{L2}	14 066	53.4	12	307.5	5 065	6.9s
javacc	**154**	**2151**	Shimple	-	-	-	-	73 884	10.9s
			TACAI^{L0}	81 917	38.1	11	150.2	71 515	4.2s
			TACAI^{L1}	81 683	38.0	11	150.2	71 003	5.4s
			TACAI^{L2}	81 651	38.0	11	150.0	70 985	11.5s
jext	**466**	**2799**	Shimple	-	-	-	-	40 670	19.2s
			TACAI^{L0}	73 428	26.2	6	119.8	17 335	4.6s
			TACAI^{L1}	73 358	26.2	6	119.8	17 297	5.0s
			TACAI^{L2}	73 334	26.2	6	119.7	17 291	6.4s
proguard	**645**	**5237**	Shimple	-	-	-	-	49 260	26.3s
			TACAI^{L0}	70 203	13.4	5	140.4	46 218	4.4s
			TACAI^{L1}	70 194	13.4	5	140.4	46 096	4.7s
			TACAI^{L2}	69 859	13.4	5	140.4	43 535	5.8s
sablecc	**286**	**2274**	Shimple	-	-	-	-	57 021	10.3s
			TACAI^{L0}	35 717	15.7	5	50.6	52 076	4.1s
			TACAI^{L1}	35 715	15.7	5	50.6	50 939	5.0s
			TACAI^{L2}	35 715	15.7	5	50.6	50 939	6.3s

Experiment 2. Here, we compare the type information that is available in Shimple, TACAI^{L0}, and TACAI^{L2} in order to answer RQ3. To perform the experiment, we compare each IR's receiver-type information of all potentially polymorphic method invocations.

The comparison across Soot's Shimple and Opal's TACAI is carried out as follows: First, we generate Shimple for all program methods. Afterward, we traverse each method's Shimple linearly and memorize for each polymorphic invocation its surrounding method, the invoked method's signature, the line number it occurred in, and its receiver type. Linear traversal allows us to distinguish multiple invocations within the same line. Then, we generate TACAI in its current configuration and match each call site with those recorded by Soot. Next, we compare the call site's receiver types to determine if Shimple's type information is more precise than ours or vice versa. If both types are equal, we consider them equally precise if TACAI does not know that its type information is precise, i.e., the exact runtime type is known. In case of precise type information, TACAI is only considered more precise when the precise type has subtypes. When intersection types are inferred, we always consider them to be more precise. However, when TACAI reports union types, we only consider them to be more precise if each type contained in the union is more precise than Shimple's receiver type. Call sites are marked as incomparable if they are not present in either representation.

All results are reported in Table 9.3, which shows the evaluated project, the compared representations, the project's invocation count, the number of unmatchable call sites, the total number of receiver types that are known to be non-null, the number of invocations with precise receiver-type information, as well as for how many call sites the receiver-type information provided by Shimple is equal, better, or worse when compared to TACAI.

Table 9.3's shows that we could match most call sites across Shimple and TACAI's representation. While comparing both IRs on Proguard, 520 remain unmatched. An investigation revealed that Shimple falsely optimizes exception handlers that pertain to JVM-level exceptions (e.g. ArrayIndexOutOfBoundsException), introducing many unmatchable call sites in *Proguard*. Additionally, the unmatched call sites in case of *javacc* are caused by the selective inlining of TACAI^{L2}.

When we only consider matchable call sites, we observe that the receiver-type information across Shimple, TACAI^{L0}, and TACAI^{L2} are mostly equal. Whereas Shimple never provides more type information than even TACAI^{L0}, TACAI^{L2} can maximally improve on *jext*, where it has more precise information for 467 receivers. However, the overall number of improvements pertaining to receiver-type information is small. When comparing the availability of nullness information, i.e., the number of cases where we definitively know that a receiver is non-null and no `NullPointerException` can be thrown, between TACAI^{L0} and TACAI^{L2}, we observe that non-nullness information is at least available in 11 % of all cases in *sablecc* and up to 40 % of all cases in *jasml*.

Table 9.3.: Receiver-type information of Experiment 2.

project	representation	#inv.	#failed	not null	precise	#equal	#Shimple >	#TACAI >
jasml	Shimple vs TACAI^{L0}	2 094	37	0	843	2 057	0	0
	Shimple vs TACAI^{L2}	2 094	37	838	1 028	1 987	0	70
javacc	Shimple vs TACAI^{L0}	9 883	0	0	4 709	9 878	0	5
	Shimple vs TACAI^{L2}	9 722	20	3 551	4 925	9 546	0	164
jext	Shimple vs TACAI^{L0}	15 457	2	0	2 803	15 450	0	5
	Shimple vs TACAI^{L2}	15 455	2	5 709	3 406	14 986	0	467
proguard	Shimple vs TACAI^{L0}	9 961	520	0	3 560	9 439	0	2
	Shimple vs TACAI^{L2}	9 959	520	3 694	4 168	9 083	0	356
sablecc	Shimple vs TACAI^{L0}	35 717	0	0	4 542	35 716	0	1
	Shimple vs TACAI^{L2}	35 715	0	4 143	5 180	35 262	0	453

Obs.25: TACAI improves little over Shimple concerning receiver-type information. However, TACAI^{L1} and TACAI^{L2} provide additional information useful for static analysis, e.g. nullness.

Experiment 3. Our last experiment evaluates how exchanging abstract domains influences CG construction, answering RQ4. To measure the effect, we construct a class hierarchy analysis (CHA) CG since it is solely based on the declared types. However, other algorithms (e.g. RTA [BS96]) may also benefit from more precise type information.

Table 9.2 provides the CG's size in the number of edges in the second last column. Whereas we can observe a great reduction of call edges compared to Shimple (up to 58%), the reduction of call edges between different versions of TACAI remains minuscule (up to 6%).

Obs.26: CHA based on TACAI improves over a Shimple-based CHA. The direct impact on call graphs between our IRs for our evaluation set is minor. However, the analyzed programs are rather small in size which lets us assume that the effect on larger programs could increase. More research is necessary to definitely answer RQ4.

9.3. Related Work

Static analysis tools often work on an intermediate representation (IR) of bytecode which facilitates static analysis. For instance, Soot [VRCG$^+$10] provides several IRs to operate on: Baf, Jimple, Grimple, and Shimple. However, Jimple and Shimple are the only TAC-based representations. Jimple is generated in 5 steps [VRH98]. At first, a naïve, verbose, and typeless TAC is generated. Step 2 takes the generated TAC and applies several code optimizations, such as constant propagation and dead code elimination. Step 3 splits, step 4 types, and step 5 packs local variables so that they are reused as often as possible. Shimple is produced by converting Jimple into SSA form. In contrast to TACAI, neither Jimple nor Shimple perform all optimizations in one step. Compared to Jimple/Shimple which always provide a type bound, TACAI can provide union and intersection types and derives the information if a specific type is an upper-type bound or a concrete type. Further, TACAI provides a comparable IR when it is configured with its cheapest domain but can be computed faster. Moreover, when advanced domains are configured, TACAI can directly provide additional information, such as def-use information or a variable's nullness.

Demange et al. [DJP10] tackle the problem that an analysis result's correctness strictly depends on the correctness of the performed transformation from the original bytecode to the IR. To mitigate the risk, they provide a formal semantics for an untyped, stack-based Java-like bytecode language, called BC. BC, however, lacks several Java bytecode features (e.g. static fields). Using the defined semantics, they provide a one-pass transformation algorithm that takes BC as input and then generates a TAC-based intermediate representation, called BIR. BIR is proven to preserve the code's semantics. During the transformation, a symbolic stack is used to decompile bytecode into TAC. However,

proposed transformation works only for a subset of Java bytecode and does not aim at making the precision configurable.

9.4. Conclusion

In this chapter we presented TACAI, an abstract-interpretation-based intermediate representation with configurable abstract domains. Our intermediate representation directly comes with three preconfigured abstract domains which—when used—result in three-address codes with different levels of precision regarding nullness or available type information. TACAI's configurable approach provides an important building block for many analyses and enables pluggable precision and scalability at the lowest level.

Our evaluation shows that TACAI is faster to compute than Soot's Shimple. Furthermore, TACAI^{L1} and TACAI^{L2} encode more information. Whereas the improvements over Shimple concerning the provided type information are minor, Chapter 6 showed that a precise IR can have a significant impact on a CG's precision.

10. Modular Call-graph Construction for Java Libraries

Previously, in Chapter 7, we systematically discussed call-graph (CG) algorithms for libraries, finding that their construction must a) consider a library's extension via inheritance and b) distinguish a library's public API and its private implementation. However, both aspects are influenced by the CG's usage scenario, implying either of the following two assumptions. In one scenario, we assume an unrestricted library usage (*open-package assumption* in short OPA). In the other scenario, we assume that a library user uses and extends only classes of the library's public API (*closed-package assumption* in short CPA). In the same chapter, we also modified the CHA CG algorithm [DGC95] to support both scenarios and evaluated their effect. Our experiments have shown that distinguishing the above usage scenarios is necessary as the number of call edges in the CGs differs significantly and each algorithm can identify unique issues. However, we performed the experiments with a CHA-based algorithm, which is fast but imprecise, limiting the scalability of subsequent analyses [Bod18].

In this chapter, we investigate whether the *open-* and *closed-package assumption* generalizes to CGs from the propagation-based CG algorithm family proposed by Tip and Palsberg [TP00]. Furthermore, we explore how to adapt CG algorithms of this family to analyze libraries to find a better base CG algorithm than CHA. We present how these algorithms can be adapted to the analysis of libraries and determine the best-suited algorithm concerning precision and performance. We implemented all CGs of the algorithm family and the library support—for both OPA and CPA—using OPAL's modular and collaborative analysis framework (cf. Chapter 8).

10.1. The Algorithms

We will use and extend Tip and Palsberg's set-based framework [TP00] to present both the existing algorithms and our extensions to those. This will enable straightforward comparison and puts our work in context. Figure 10.1 shows the relationships between the four algorithms we will adapt and their cost/accuracy relation compared to other well-known algorithms. We will focus on the four algorithms shown in the shaded area, namely CTA, MTA, FTA, and XTA[1]. The ordering from left to right corresponds to the number of sets used by each algorithm to approximate the runtime receiver types at a callsite, resulting in greater cost and accuracy, while potentially limiting their scalability. For example, rapid-type analysis (RTA) uses a single global set to approximate a

[1] Please note that these are the algorithms' complete names and not acronyms. However, like the rapid-type analysis (RTA), these algorithms are also type-based analyses.

program's runtime types. In contrast, XTA captures the program's runtime types using one set for each field and for each method. The rationale behind using multiple sets is to provide somewhat local type information and, thus, resolve callsites more accurately.

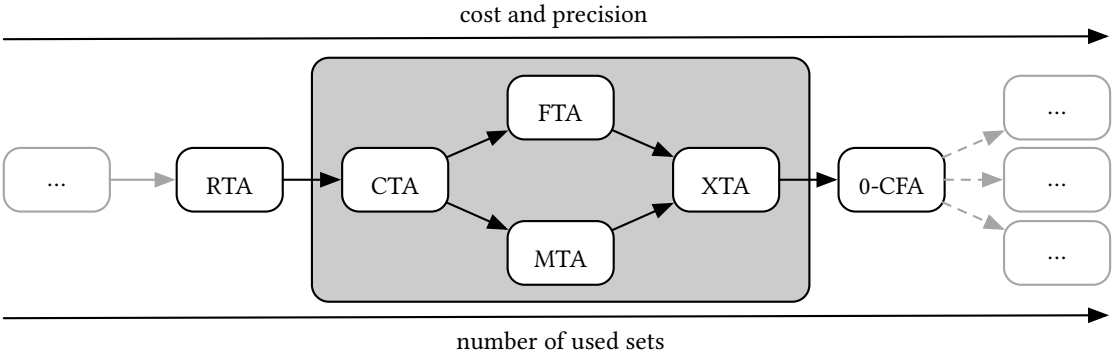

Figure 10.1.: Overview of the algorithms adapted and studied in this chapter.

10.1.1. Call Graphs fro Applications

Next, we will introduce the application CG algorithms and their propagation constraints as defined by Tip and Palsberg [TP00]. First, we will define RTA as a baseline. Second, we will define the most precise algorithm XTA. Finally, we will adapt the definition of XTA to define CTA, FTA, and MTA.

All definitions only define how an algorithm resolves virtual calls. Other language features, APIs, or non-virtual calls are not discussed as these are orthogonal to virtual call resolution and can be handled equally throughout all algorithms.

RTA The rapid-type analysis [BS96] captures type-instantiation information in a global set variable IT (instantiated types). Whenever the algorithm encounters a constructor call during CG construction, it adds the type of the constructed object to IT.

To define the CG's constraints, we use the following notion. R denotes the set of reachable methods, *StaticType(e)* denotes the static type of the expression e, *SubTypes(t)* denotes the set of declared subtypes of t, and *StaticLookup(C,m)* denotes the definition of a method m that is found on method lookup in the class C. Furthermore, we use *eps* to denote the set of entry points. For example, when constructing a whole-program application CG, *eps* contains only the program's *main* method.

The following three contraints[2] define RTA:

1. $eps \in R$
2. For each method M, each virtual callsite e.m(...) occurring in M, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) =$ M': $(M \in R) \wedge (C \in IT) \Rightarrow (M' \in R)$.
3. For each method M, and for each `new C()` occuring in M: $(M \in R) \Rightarrow (C \in IT)$.

Intuitively, the first constraint reads: all entry-point methods are reachable. Constraint two reads: if a method m is reachable and its body contains a virtual callsite *e.m(...)*, then every method with m's signature[3] that is a) inherited[4] by a subtype of e's static type and b) declared in an instantiated type, becomes reachable. The third constraint expresses: IT contains all classes that a reachable method instantiates.

XTA In contrast to RTA, the remaining CGs use multiple sets to capture the instantiated types. Instead of using a global set, they associate multiple sets with different program entities, such as classes, methods, or fields. XTA uses a distinct set IT_M for each method M and a distinct set IT_F for each field F. To define XTA's constraints, we introduce further notation. *ParamTypes(M)* denotes the set of static types of a method M's formal arguments[5]. *ReturnType(M)* denotes M's declared return type. Additionally, we must extend SubTypes($\cdot$) to work on a set of types:

$$SubTypes(TS) = \bigcup_{t \in TS} SubTypes(t)$$

Using these notions, we can define XTA with the following five constraints:

1. $eps \in R$
2. For each method M, each virtual callsite e.m(...) occurring in M, and each class $C \in SubTypes(StaticType(e))$ where $StaticLookup(C, m) =$ M' : $(M \in R) \wedge (C \in IT_M)$
$$\Rightarrow \begin{cases} (M' \in R) \wedge & (a) \\ SubTypes(ParamTypes(M')) \cap IT_M \subseteq IT_{M'} \wedge & (b) \\ SubTypes(ReturnType(M')) \cap IT_{M'} \subseteq IT_M \wedge & (c) \\ C \in IT_{M'}. & (d) \end{cases}$$
3. For each method M, and for each `new C()` occuring in M: $(M \in R) \Rightarrow (C \in IT_M)$.

[2]Implications can become true if the left-hand side of the implication is false. Hence, we are only interested in the minimal sets of *eps*, R, and IT that fulfill these three constraints.
[3]We consider a method's signature to consist of its name, formal type parameters, and return type.
[4]Please note that the subtype relation is reflexive and, therefore, each class C is also a subtype of itself.
[5]The method's *this* pointer is excluded.

4. For each method M in which a read of a field F occurs:
   $(M \in R) \Rightarrow IT_F \subseteq IT_M$.

5. For each method M in which a write of a field F occurs:
   $(M \in R) \Rightarrow (SubTypes(StaticType(F)) \cap IT_M) \subseteq IT_F$.

Again, all entry points are reachable. Constraint two refines the respective RTA constraint as follows: a) for *M'* to become reachable, its declaring class C must be available in method M's instantiated type set IT_M, b) all types that are locally available in M and compatible with the types of the callee's formal paramterers flow to the type set of *M'* ($IT_{M'}$), c) all types that are locally available in *M'* and compatible with its return type flow to M's type set IT_M, and d) type C is always available in *M'* through its implicit *this* reference. The third constraint restricts the corresponding RTA rule by adding the instantiated type C only to the local type set IT_M. The fourth constraint reflects type flow from a field's type set to the method M's set whenever a field read of F occurs. Analogously, the fifth constraint models the type flow from M's set to the set of a written field F.

Next, we will define additional constraints to obtain MTA, FTA, and CTA. These constraints focus on unifying the type sets IT_M and IT_F in different ways.

MTA This algorithm uses a distinct set variable for each class C (IT_C) and for every field F (IT_F), thereby unifying the type information for all methods within a class. As in XTA, the type sets for fields remain separated. MTA can be obtained by adding the following constraint to the definition of XTA:

1. If a class C defines a method M: $IT_C = IT_M$.

FTA This algorithm uses a distinct set variable for each class C (IT_C) and for every method M (IT_M). In contrast to MTA, FTA unifies the type information for all fields, still separating methods. Hence, FTA can be obtained by adding the following constraint to the definition of XTA:

1. If a class C defines a field F: $IT_C = IT_F$.

CTA This algorithm uses a distinct set variable IT_C for each class C. Hence, compared to XTA, the type information for all methods and fields of a class is unified. The algorithm is obtained by adding the following constrains to XTA's definition:

1. If a class C defines a method M: $IT_C = IT_M$.
2. If a class C defines a field F: $IT_C = IT_F$.

10.1.2. Library Considerations

In Chapter 7, we discussed how to extend existing CG algorithms for libraries. Besides the advanced entry-point computation and the call-by-signature resolution for interface-based calls, adapting the previously presented CG algorithms for libraries poses an

additional challenge. As our program is no longer a closed world, we must consider the external world's influence. Thus, we must initialize the instantiated type sets of all entry-point methods of each algorithm accordingly. Like the extensions discussed in Chapter 7, the initialization of the instatianted type sets also requires a distinction between the *open-package assumption* (OPA) and *closed-package assumption* (CPA). Chapter 7 presented how these two assumptions directly influence which program entities are accessible. Therefore, we will not distinguish OPA and CPA explicitly to define the algorithm extensions. However, we will use the concept of accessibility to enable concise definitions.

Next, we discuss all library extensions and define additional XTA constraints. As for the application algorithms, adding the constraints for CTA, MTA, or FTA will result in the respective algorithm.

Entry Points An entry point is a method that is considered reachable ($M \in R$) at the start of CG construction. The computation of the entry points is described in Chapter 7. While Listing 7.3 shows the algorithm in case of OPA, Listing 7.4 refines the entry-point computation for CPA. Here, we can use the same algorithms to determine our initial set of entry points *eps* with respect to OPA and CPA.

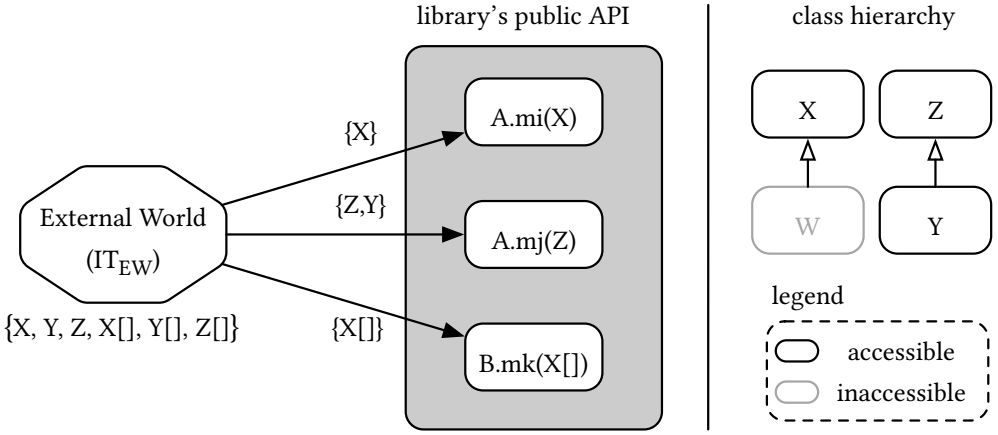

Figure 10.2.: Type-set instantiation example in the presence of an external world.

Type-set Initialization The entry points define the library's public API, i.e., they comprise all methods that can directly be called by a client of the library. Whereas an application's main method always has the type `String[]` as formal parameter, library entry points may have formal parameters with many different types, provided by the caller.

The problem is illustrated by Figure 10.2. It shows how the external world—denoted by IT_{EW}—can access a library's public API and how these accesses can affect the constructed CG. The displayed library has internal classes that cannot be accessed from

the external world. In our example, the class W belongs to the library-private implementation. All other classes are accessible, i.e., the external world can construct objects of them. As the library's public API has three methods with formal parameters, the type sets associated with these methods must be initialized accordingly. Assuming an algorithm that keeps separate type sets for each method (e.g., XTA), we must initialize `A.mi(X)`'s type set $IT_{A.mi}$ as $\{X\}$. Please note that we do not have to consider X's subtype W since it is not accessible. In contrast, the method `A.mj(Z)` has a formal-parameter type Z which is subtyped by the accessible type Y. Therefore, $IT_{A.mj}$ must be initialized with $\{Z, Y\}$. As `B.mk(X[])` shows, array types can be handled analogously.

To recap, the set of accessible types, methods, and fields depends on the library-private implementation (cf. Chapter 7). Intuitively, all classes, methods, and fields that can be accessed from the external world are considered accessible. Moreover, all classes with constructors that are externally callable must be assumed to be instantiated by a client of the library. To define the additional constraints, we will use C_{acc} to denote all accessible classes and F_{acc} to denote all accessible fields. $SubTypes_{acc}(t)$ denotes the set of subtypes of t that are accessible. Analogously, we adapt the $SubTypes_{acc}(\cdot)$ predicate that works on a set of types.

We add the following constraints to the definition of XTA:

1. For each method M:
   $(M \in eps) \Rightarrow (SubTypes_{acc}(ParamTypes(M)) \cap IT_{EW}) \subseteq IT_M$.
2. For each field F:
   $(F \in F_{acc}) \Rightarrow (SubTypes_{acc}(StaticType(F)) \cap IT_{EW}) \subseteq IT_F$.

Intuitively, the first constraint expresses that each entry point's instantiated type set initializes to all types from the external world that a client might pass to the method. Constraint two says that each accessible field's instantiated type set initializes to all types that can be assigned to the field.

Call-by-signature Resolution The goal of call-by-signature (CBS) resolution is to cover possible inheritance scenarios where the external world extends library classes. To deal with these scenarios, we must perform CBS resolution at interface-based callsites. All details pertaining to CBS resolution were discussed in Chapter 7. However, CBS resolution concerns only classes that are not (yet) in an inheritance relation with the callsite's interface and is therefore orthogonal to virtual call resolution.

We add the following constraint to the definition of XTA to enable CBS resolution:

1. For each method M, each interface callsite e.m(...) occurring in M where at least one type in $SubTypes(StaticType(e))$ is accessible, and each class $C \in \neg SubTypes_{acc}(StaticType(e))$ where $StaticLookup(C, m) = M'$:
   $(M \in R) \wedge C \in (IT_{EW} \cap SubTypes_{acc}(StaticType(e)) \Rightarrow (M' \in R)$.

The constraint reads: if a method is reachable and contains a virtual call on an interface type *e.m(...)*, then every method with the same name m that a) is not declared in a subtype of that interface e but b) is declared in or inherited by an accessible class is also reachable.

10.2. Implementation

We implemented the previous section's version of the XTA algorithms using Opal's framework for modular and collaborative static analysis presented in Chapter 8. Our implementation consists of four generic core analyses that derive three property kinds.

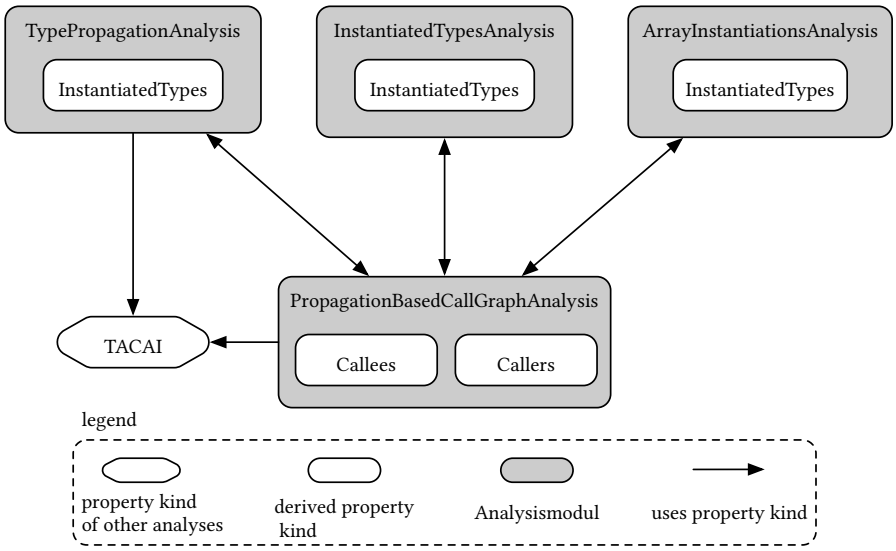

Figure 10.3.: Overview of the different analysis modules that collaboratively compute the call graph.

Figure 10.3 gives an overview of these components and depicts their relations. The *PropagationBasedCallGraphAnalysis* depends on our intermediate representation TACAI (cf. Chapter 9) and the *InstantiatedTypes* property kind. Based on that information, it resolves direct and virtual method calls. Furthermore, the *InstantiatedTypesAnalysis*, the *ArrayInstantiationsAnalysis*, and the *TypePropagationAnalysis* collaboratively compute the instantiated types. The regular *InstantiatedTypesAnalysis* is responsible for adding newly instantiated types to an entity's type set. Analogously, the *ArrayInstantiationsAnalysis* exclusively handles array types and their elements. The *TypePropagationAnalysis* handles the propagation of these type sets. Collaboratively computing the instantiated types brings two advantages. First, the separate analysis for array handling allows implementing various models of arrays. Second, the separation of the instantiated types and the type propagation enables isolated development, and we can implement several type-propagation strategies.

Abstracting over the concrete entities that are associated with a type set, this design allows to instantiate all CG algorithms from the set-based framework. The implementation is parameterized over a function, the *TypeSetEntitySelector* that given an entity (e.g., a field or a method) returns the entity its type set is associated with (e.g., a method's class). While the proposed design allows to instantiate many different CGs from the set-based framework, we currently provide entity selectors to model RTA, CTA,

FTA, MTA, and XTA.

10.2.1. Call-graph Construction Lifecycle

The CG is computed in 5 steps. First, we must compute the entry points. Second, we must configure the CG's entity selector. Third, given the entry points and the entity selector, we must preinitialize the type sets. Fourth, we can register additional analyses to support additional language features. Finally, we can start the CG construction at all entry points, triggering the fixed-point computation. When the fixed-point is reached, the CG becomes available. Next, we shortly discuss each step.

Entry Points Our implementation relies on the entry points derived by OPAL's *EntryPointsFinders*. In addition to application entry points, these can compute library entry points that are conform either to OPA or CPA. Hence, the concrete entry-point finder determines whether the CG is suitable for applications, libraries with OPA, or libraries with CPA.

Entity Selection To instantiate a concrete algorithm from Tip and Palsberg's set-based CG framework, we use the following trait to define a type-set-entity-selector function:

```
trait TypeSetEntitySelector extends (Entity ⇒ TypeSetEntity)
```

We use this function to determine the type set representing the available types at a given entity. Listing 10.1a and Listing 10.1b show examples of set-entity selectors for CTA and XTA. CTA's function maps methods and fields to their respective class file (cf. Line 6 and 7), i.e., their instantiated type sets are associated with their class, unifying the type sets of a class' methods and fields. In contrast, XTA's function maps fields and methods to themself, keeping separate type sets (cf. Line 6 and 7). Additionally, it is possible to define how array types as well as fields and methods that were not found on the classpath are handled. In our examples, we associate the type sets of unknown entities with the external world (cf. Line 10 and 12).

Preinitalizing Type Sets Before we can start to intialize the type sets of entry-points methods and accessible fields, we must initialize the type set of our external world (IT_{EW}). Hence, we must compute which classes might be instantiated by a client of the library. However, this initial set of instantiated types depends on our CG construction scenario, i.e., whole-program, library with OPA, or library with CPA. OPAL's *InstantiatedTypesFinder* offers that functionality and computes IT_{EW} according to the respective scenario. Using IT_{EW}, we intialize the entry point's type sets as described in Section 10.1 (cf. Figure 10.2). Similarly, we initialize the type sets of accessible fields.

Registering Additional Analyses OPAL's framework for modular and collaborative static analyses allows us to define additional analyses that can contribute to the CG. In particular, we can add further analyses that collaboratively compute the CG's caller

```
1  object CTASetEntitySelector
2    extends TypeSetEntitySelector {
3
4    override def apply(e: Entity):
         TypeSetEntity =
5      e match {
6        case m: Method  ⇒ m.classFile
7        case f: Field ⇒ f.classFile
8        case at: ArrayType ⇒ at
9        case em : ExternalMethod
10         ⇒ ExternalWorld
11       case ef: ExternalField
12         ⇒ ExternalWorld
13     }
14 }
```

(a)

```
1  object XTASetEntitySelector
2    extends TypeSetEntitySelector {
3
4    override def apply(e: Entity):
         TypeSetEntity =
5      e match {
6        case m: Method   ⇒ m
7        case f: Field ⇒ f
8        case at: ArrayType ⇒ at
9        case em : ExternalMethod
10         ⇒ ExternalWorld
11       case ef: ExternalField
12         ⇒ ExternalWorld
13     }
14 }
```

(b)

Listing 10.1.: Definition of the CTA and XTA entity selectors.

and callee property kinds.[6] While our *PropagationBasedCallGraphAnalysis* computes the call targets of direct or virtual calls, additional analyses can contribute this information for different features and APIs, e.g., reflection, serialization, threads, or finalizers. Moreover, when analyzing a library, we can add an analysis that contributes CBS targets.

10.2.2. Design Decisions

Using the algorithms on real-world applications requires us to make several design choices. In the following, we discuss the most relevant ones:

Array Handling We model arrays as unique entities. For each n-dimensional array type A_n in the program, we use a special *ArrayType* entity that represents all of its elements (IT_{A_n}). Hence, all arrays of the same type and dimension share one type set.

We use the same methods to track and propagate array types as we use for fields and classes. For example, whenever we observe a new array's construction, we add the array type to the local type set (e.g. $AT \in IT_M$). Thus, the type sets carry information about classes and arrays available to an entity. We use $ElementTypes(AT)$ to denote the set of types that an array can hold[7]. If an array AT is read by a method M, we propagate $(IT_{A_n} \cap ElementTypes(AT))$ to IT_M. When an array AT is written by a method M, we propagate $(IT_M \cap ElementTypes(AT))$ to IT_{A_n}.

We chose this representation as it is a global but correct model for arrays. However, we modularized the computation of the instantiated types and can therefore quickly

[6] Analogously to the case study presented in Section 8.3.

[7] Multi-dimensional arrays hold lower-dimensional arrays whose element types are not returned by *ElementTypes*.

adapt our array modeling.

Exception Handling Tip and Palsberg [TP00] argue that precise exception tracking is very complex and might not be worthwhile. We agree and use a separate global set IT_{Ex} to approximate the runtime types of all exceptions. This set contains only subtypes of `java.lang.Throwable` and is only used to resolve calls on exception objects.

Generics Generic types are common in Java. For instance, they are heavily used in Java's collection classes to facilitate type-safe access. However, due to *type erasure*, generic type information is lost during compilation. Therefore, if an object of a generic collection is retrieved and the type has been removed, all following method calls are performed on the root type `java.lang.Object`. However, the compiler sometimes adds a `checkcast` instruction after a call on a generic object, restoring the lost information. We explicitly check for these instructions to maintain precision.

Incomplete Code While we have described how our algorithms construct CGs for libraries, any realistic implementation must deal with unknown code elements, i.e., code elements that are not in the scope of the analyzed program. Such a scenario is typical for non-whole-program analyses. For example, when analyzing a program with incomplete dependencies, one may encounter field reads and writes or method calls that cannot be further analyzed. Hence, we must approximate their behavior. Since our set IT_{EW} already models the external world, we chose to also use it in the following cases:

- If a method M calls an externally declared method M', we propagate $(IT_M \cap ParamTypes(M'))$ to IT_{EW}. When M' is a virtual method, the `this` pointer is additionally propagated. In turn, for methods that return a reference type, we propagate $(IT_{EW} \cap ReturnType(M'))$ to IT_M.

- If a method M writes to an externally declared field F, we propagate $(IT_M \cap StaticType(F))$ to IT_{EW}. Reading such a field F causes a propagation of $(IT_{EW} \cap StaticType(F))$ to IT_M.

Please note that all entry points always depend on IT_{EW}. Therefore, whenever new types are become available in IT_{EW}, these will be propagated to all entry-point methods.

Modeling Java Features Due to our approach's modular and collaborative design, we can add support for additional features or APIs by defining new independent analyses. Our current implementation provides analyses for reflection, serialization, threads, finalizers, and some native methods. If support for a particular feature is needed, one can add the respective analysis to the modular CG construction algorithm. Once we develop an analysis for a feature, one can reuse it in every CG that our implementation can potentially instantiate.

10.3. Evaluation

We evaluate the proposed approach along two major dimensions: a) by comparing a call graph (CG) computed using the classical RTA against the call graphs computed by our CG algorithms; b) by comparing the proposed algorithms with each other.

RQ1 What is the performance overhead of initializing the different CG construction algorithms' type sets?

RQ2 What is the effect of OPA and CPA on the initialized type sets?

RQ3 What is OPA and CPA's effect on the set-based CG algorithms with respect to their CG's call edges and reachable methods?

Setup We use ten algorithms (RTA_{OPA}, CTA_{OPA}, MTA_{OPA}, FTA_{OPA}, XTA_{OPA}, RTA_{CPA}, CTA_{CPA}, MTA_{CPA}, FTA_{CPA}, and XTA_{CPA}) to construct respective call graphs[8] for a large set of libraries: the 100 most used *distinct* Java related libraries from the Maven Central Repository. Specifically, we used the same set of libraries as in Chapter 7. The data set's full description is available in Section 7.3. We also downloaded all dependencies to build a complete class hierarchy.

Instead of analyzing all library dependencies, we consider only the public interface of all third-party libraries. Otherwise, the performance and the set of call edges would be dominated by dependent libraries. For over 90% of our evaluation projects, the analyzed library defines less than 5% of all methods itself, while 95% of the code base's methods are introduced by used third-party libraries.

All measurements are taken on a Mac Pro with a Xeon E5 with 8 cores@3GHz and a JVM with 24GB of heap space.

Experiments While we performed the experiments with all CG algorithms, we show only excerpts of our data. Because CTA and MTA as well as FTA and XTA exhibit very similar behavior, we group these algorithms. Tables 10.1-10.7 show selected results of our experiments. Most of these tables list only the top five and the bottom five libraries with respect to the measured effect. Furthermore, those tables contain the mean and the standard deviation over *all* 100 libraries. Next, we will use this information to answer the research questions.

10.3.1. Performance Overhead of Type-set Initialization

To answer RQ1, we measure for each algorithm the size of the initialized type sets and the time it takes to compute them. Table 10.1 presents the number of type sets, their average size[9], and the initialization time of RTA and XTA variants. While RTA's initialization time slightly increases when more types are available in IT_{EW}, XTA's

[8] Each CG was constructed with a timeout of 600 s.

[9] The size of RTA's initial type set is equivalent to the size of IT_{EW}.

Table 10.1.: XTA and RTA type-set information directly after initialization on the top and bottom five projects from our data set.

Project	Algorithm	#TS	∅ TS Size	🕓 Initialization
slf4j-log4j12	RTA_{OPA}	1	3 016	1.1 s
	RTA_{CPA}	1	1 662	1.1 s
	XTA_{OPA}	1 086	223	1.8 s
	XTA_{CPA}	371	63	1.5 s
commons-fileupload	RTA_{OPA}	1	2 902	1.1 s
	RTA_{CPA}	1	1 653	1.1 s
	XTA_{OPA}	1 105	48	1.7 s
	XTA_{CPA}	428	41	1.5 s
easymockclassextension	RTA_{OPA}	1	3 021	3.3 s
	RTA_{CPA}	1	1 667	3.1 s
	XTA_{OPA}	1 027	72	4.6 s
	XTA_{CPA}	285	75	1.5 s
hibernate-jpa-2.0-api	RTA_{OPA}	1	2 750	1.1 s
	RTA_{CPA}	1	1 526	1.1 s
	XTA_{OPA}	1 014	82	1.7 s
	XTA_{CPA}	355	81	1.5 s
json	RTA_{OPA}	1	2 750	1.1 s
	RTA_{CPA}	1	1 528	1.1 s
	XTA_{OPA}	1 085	99	1.7 s
	XTA_{CPA}	459	98	1.6 s
hibernate-core	RTA_{OPA}	1	6 196	3.5 s
	RTA_{CPA}	1	3 661	3.2 s
	XTA_{OPA}	26 732	474	74.8 s
	XTA_{CPA}	20 108	289	32.8 s
hsqldb	RTA_{OPA}	1	3 230	3.3 s
	RTA_{CPA}	1	1 791	3.2 s
	XTA_{OPA}	14 030	166	24.9 s
	XTA_{CPA}	7 795	101	8.7 s
groovy-all	RTA_{OPA}	1	6 821	3.3 s
	RTA_{CPA}	1	4 104	3.4 s
	XTA_{OPA}	32 109	2 070	201.9 s
	XTA_{CPA}	23 817	1 310	95.0 s
scala-compiler	RTA_{OPA}	1	15 151	1.1 s
	RTA_{CPA}	1	13 845	1.2 s
	XTA_{OPA}	52 225	1 986	128.6 s
	XTA_{CPA}	48 650	1 898	112.7 s
scala-library-2.10.4	RTA_{OPA}	1	5 899	3.5 s
	RTA_{CPA}	1	4 658	3.1 s
	XTA_{OPA}	53 448	1 088	229.0 s
	XTA_{CPA}	49 936	929	175.3 s

TS = Type set.

initialization time additionally depends on the program's number of accessible methods and fields. Hence, the computation of type flows from IT_{EW} to XTA's methods and fields becomes more expensive with a) larger type sets and b) an increased number of methods and fields, leading to a big increase in initialization time for larger projects, such as *scala-library-2.10.4* or *scala-compiler*. The named projects' initialization time increases by several orders of magnitude, up to 116.3× of RTA. However, Table 10.2 shows that the overhead from type-set initialization increases only by a factor of 4.7-7.6 on average, depending on the algorithm. In relation to the full CG construction time, type-set initialization consumes 11.4%-16.6%.

Obs.27: The type-set initialization for all algorithms of the XTA-family has on average a non-negligible overhead of a factor between 4.7 and 7.6 compared to a RTA. Thus, it does not scale well to large libraries that come with many dependencies.

Table 10.2.: Information on type sets and runtime of each algorithm. All of the present information are means over all libraries.

Algorithm	∅ **#Type Sets**		∅ **Type-set Size**		∅ **Time**	
	initialization	final	initialization	final	initialization	total
RTA_{OPA}	1	1	4 053	4 104	2.9 s	13.4 s
CTA_{OPA}	1 069	5 034	968	873	22.6 s	144.2 s
MTA_{OPA}	1 836	6 143	635	704	19.1 s	143.4 s
FTA_{OPA}	5 765	10 753	456	742	19.7 s	138.1 s
XTA_{OPA}	6 480	12 098	363	643	17.0 s	118.9 s
RTA_{CPA}	1	1	2 487	2 661	2.7 s	8.9 s
CTA_{CPA}	673	3 119	763	794	14.7 s	128.5 s
MTA_{CPA}	903	3 887	578	682	12.6 s	118.3 s
FTA_{CPA}	4 531	8 529	328	536	15.4 s	106.7 s
XTA_{CPA}	4 720	9 226	288	494	12.5 s	92.7 s

10.3.2. Comparing Type Sets across CPA and OPA

To answer the second question, we compare the initial type sets' size across the algorithms that operate on OPA and those operating on CPA. Across the four base algorithms, XTA is affected the most. That is, as described in Chapter 7, because CPA reduces the number of initially instantiated types, the number of accessible fields, and the number of entry points and, hence, XTA has the most potential for improvement. On some programs (e.g., guava, guice, and gson), we observe a reduction of the number of initial type sets between 65-78% and a decrease in the average initial set size of up to 90% for all algorithms. While these projects make good use of Java's visibility modifiers, unfortunately, the Scala compiler does not. Again, as in Chapter 7, we observe the least effect on Scala programs. Interestingly we find that for some programs, the

average type-set sizes only slightly decreases, although the size of IT_{EW} and the number of initialized type sets massively dropped. For example, in *easymockclassextension* the use of CPA shrinks $IT_{EW} \approx 45\%$ and reduces the number of type sets by 72% compared to OPA. Nevertheless, the average size of all type sets increases by three. An inspection revealed that the remaining entry points and fields had `java.lang.Object` as a formal parameter or declared type respectively, often leading to the propagation of the entire set IT_{EW}. Unification of all entities that use `java.lang.Object` into a set IT_O may reduce the overhead of maintaining multiple sets.

Obs.28: Frequent use of `java.lang.Object` as formal parameter or field type in entry points or accessible fields generates numerous large sets.

Considering the average number of initial type sets (cf. Table 10.1) and the average initial type-set size together with their final state (cf. Table 10.2), we observe that all CPA algorithms benefit from respecting the library-private implementation. Although RTA uses only a single set, it can leverage CPA. Yet, as RTA_{CPA}'s average type-set size indicates, its type set is often initialized to 80-93% the size of the final type set. Instead, RTA_{OPA} adds almost all available types during the initialization phase.

Obs.29: When using CPA, RTA does not degenerate to a CHA, implying that a significant number of libraries do not use `java.lang.Object` as a formal parameter in their entry points. Thus, under CPA RTA can improve over CHA, while under OPA it practically is equivalent to CHA.

All other algorithms profit from CPA too. The average number of initial type sets of each algorithm decreases between 20% (FTA_{CPA}) and 38.4% (CTA_{CPA}), reducing the average number of final type sets of CTA_{CPA} by 37%, MTA_{CPA} by 50.8%, FTA_{CPA} by 21.4%, and XTA_{CPA} by 27%. Additionally, the decrease of each type set's size lets us assume that less information is provided at each call site, leading to faster execution.

Obs.30: CPA has a positive effect on the average number and size of an algorithm's type sets. Additionally, it reduces each algorithm's initialization time.

10.3.3. Comparing Advanced Library Call Graphs

To answer research question three, we compare CGs computed by all algorithms in the context of OPA and CPA. First, we compare RTA_{OPA} to CTA_{OPA}, MTA_{OPA}, FTA_{OPA}, and XTA_{OPA}. Second, we compare RTA_{CPA} to CTA_{CPA}, MTA_{CPA}, FTA_{CPA}, and XTA_{CPA}. Finally, we measure the differences between XTA_{OPA} and XTA_{CPA}.

OPA Call Graphs We group algorithms with similar performance. Whereas Table 10.3 shows the results for CTA_{OPA} and MTA_{OPA}, Table 10.4 provides results FTA_{OPA} and XTA_{OPA}. With an exception of *org.apache.felix.scr.annotations* where all algorithms can reduce the number of call edges by 92% compared to RTA_{OPA}, CTA_{OPA} and MTA_{OPA} perform consistently worse than FTA_{OPA} and XTA_{OPA}. Over all projects, they can reduce the number of call edges only by 6-7%. Considering that these al-

gorithms take approximately more than 20 times longer to compute than RTA_{OPA}, the gains seem negligible. Despite also being way slower than RTA_{OPA}, FTA_{OPA} and XTA_{OPA} can considerably reduce the number of call edges by 27% and the number of reachable methods by 9% on average.

Obs.31: Set-based algorithms that unify the type flow for all methods can not improve significantly over RTA_{OPA}. Hence, they seem not well suited in an open-package library scenario. However, when each method is represented by a separate type set, as in FTA_{OPA} and XTA_{OPA}, the number of call edges can be reduced by more than 25% without losing any soundness. Despite being slower than RTA_{OPA}, XTA_{OPA} is a viable option for OPA CGs.

CPA Call Graphs Observing similar behavior for CTA_{CPA} and MTA_{CPA} as well as for FTA_{CPA} and XTA_{CPA}, we group their results. Table 10.5 displays the results for CTA_{CPA} and MTA_{CPA}, while Table 10.6 shows FTA_{CPA} and XTA_{CPA}'s results. All CG construction algorithms exhibit similar behavior as under OPA. However, due to the smaller initialized type sets, the overall CG size can be reduced by 11% (CTA_{CPA} and MTA_{CPA}), or 26% (FTA_{CPA} and CTA_{CPA}) respectively. While Tip and Palsberg [TP00] observed a smaller effect of an average reduction of only 7% (XTA) on their application benchmark, these algorithms show more significant benefits on libraries. Moreover, we observe that the resulting CGs from algorithms that unify the type sets for methods and those that do not are very similar. Hence, CTA and MTA produce similar CGs and FTA and XTA, respectively. While the average reduction of call edges or reachable methods is almost identical, we find that for some programs (e.g., *protobuf-java*) XTA_{CPA} produces slightly smaller CGs than FTA_{CPA}. Still, XTA produces the resulting CGs faster (cf. Table 10.2). In general, XTA_{CPA} maintains more but smaller type sets than FTA_{CPA}, reducing the cost to compute set intersection while propagating type sets. Compared to RTA_{CPA}, XTA_{CPA} reduces the number of reachable methods by 12% and the number of call edges by 26%. Unfortunately, this improvement comes with high runtime cost. Our experiments show that XTA_{CPA} is $\approx 10\times$ slower than RTA_{CPA}.

Obs.32: XTA_{CPA} is the most precise and fastest algorithm and, therefore, is superior to all other CG algorithms of Tip and Palsberg's propagation-based CG-family. The gains compared to RTA_{CPA} are more significant on libraries then they reported in an application setting.

Table 10.3.: Number of reachable methods and number of call edges produced by the RTA_{OPA}, CTA_{OPA}, and MTA_{OPA} algorithms for the top and bottom five libraries from our data set.

Project	**RTA_{OPA}**		**CTA_{OPA}**				**MTA_{OPA}**			
	#RM	#E	#RM	#E	↓ #RM	↓ #E	#RM	#E	↓ #RM	↓ #E
org.apache.felix.scr.annotations	665	4 824	167	395	75%	92%	167	395	75%	92%
validation-api	178	234	112	130	37%	44%	112	130	37%	44%
commons-cli	977	3 408	519	2 243	47%	34%	517	2 079	47%	39%
hibernate-jpa-2.0-api	340	657	267	420	21%	36%	267	420	21%	36%
easymockclassextension	47	60	40	39	15%	35%	40	39	15%	35%
hamcrest-library	766	1 748	766	1 748	0%	0%	766	1 748	0%	0%
hamcrest-core	822	1 676	822	1 676	0%	0%	822	1 676	0%	0%
json	1 238	16 109	1 238	16 109	0%	0%	1 238	16 109	0%	0%
jsr305	83	75	83	75	0%	0%	83	75	0%	0%
asm	422	1 799	422	1 799	0%	0%	422	1 799	0%	0%
mean (over all projects)					4%	6%			4%	7%
std dev (over all projects)					11%	13%			11%	13%

#RM = number of reachable methods; #E = number of call edges. ↓ reduction in the number of reachable methods or call edges.

Table 10.4.: Number of reachable methods and number of call edges produced by the RTA_{OPA}, XTA_{OPA}, and FTA_{OPA} algorithms for the top and bottom five libraries from our data set.

Project	**RTA_{OPA}**		**FTA_{OPA}**				**XTA_{OPA}**			
	#RM	#E	#RM	#E	↓ #RM	↓ #E	#RM	#E	↓#RM	↓#E
org.apache.felix-.scr.annotations	665	4 824	166	366	75%	92%	166	366	75%	92%
maven-project	2 800	27 551	1 801	7 237	36%	74%	1 734	6 903	38%	75%
maven-core	7 140	123 717	5 873	36 313	18%	71%	5 873	36 035	18%	71%
commons-cli	977	3 408	433	1 036	56%	70%	433	1 036	56%	70%
testng	7 378	199 643	7 152	81 055	3%	59%	7 152	80 210	3%	60%
fest-assert	1 191	2 564	1 186	2 492	0%	3%	1 186	2 492	0%	3%
scalacheck_2.10	13 689	231 394	13 172	228 367	4%	1%	13 172	228 367	0%	1%
asm	422	1 799	418	1 791	0%	0%	418	1 791	0%	0%
jackson-annotations	609	649	607	647	0%	0%	607	647	0%	0%
jsr305	83	75	83	75	0%	0%	83	75	0%	0%
mean (over all projects)					9%	27%			9%	27%
std dev (over all projects)					13%	18%			13%	18%

#RM = number of reachable methods; #E = number of call edges. ↓ reduction in the number of reachable methods or call edges.

Table 10.5.: Number of reachable methods and number of call edges produced by the RTA_{CPA}, CTA_{CPA}, and MTA_{CPA} algorithms for the top and bottom five libraries from our data set.

Project	**RTA_{CPA}**		**CTA_{CPA}**				**MTA_{CPA}**			
	#RM	#E	#RM	#E	↓ #RM	↓ #E	#RM	#E	↓ #RM	↓ #E
org.apache.felix.scr.annotations	474	3 106	166	395	65%	87%	166	395	65%	87%
slf4j-log4j12	117	202	49	45	58%	78%	49	45	58%	78%
commons-cli	744	2 069	420	1 183	44%	43%	420	1 183	44%	43%
org.osgi.core	1 327	7 036	1 081	4 403	19%	37%	1 081	4 403	19%	37%
lombok	660	1 091	516	722	22%	34%	516	722	22%	34%
asm	405	1 782	396	1 771	2%	1%	396	1 771	2%	1%
jackson-databind	7 982	54 008	7 925	53 750	1%	0%	7 925	53 750	1%	0%
maven-model	1 334	9 523	1 331	9 505	0%	0%	1 331	9 505	0%	0%
jsr305	83	75	83	75	0%	0%	83	75	0%	0%
json	958	11 085	958	11 085	0%	0%	958	11 085	0%	0%
mean (over all projects)					7%	11%			7%	11%
std dev (over all projects)					11%	14%			10%	14%

#RM = number of reachable methods; #E = number of call edges. ↓ reduction in the number of reachable methods or call edges.

Table 10.6.: Number of reachable methods and number of call edges produced by the RTA_{CPA}, XTA_{CPA}, and FTA_{CPA} algorithms for the top and bottom seven libraries from our data set.

Project	**RTA_{CPA}**		**FTA_{CPA}**				**XTA_{CPA}**			
	#RM	#E	#RM	#E	↓ #RM	↓ #E	#RM	#E	↓ #RM	↓ #E
org.apache.felix- .scr.annotations	474	3 106	164	357	65%	89%	164	357	65%	89%
jboss-logging	807	3 083	710	532	12%	83%	710	532	12%	83%
slf4j-log4j12	117	202	49	42	58%	79%	49	42	58%	79%
org.osgi.core	1 327	7 036	839	2 167	37%	69%	839	2 167	37%	69%
json	958	11 085	907	4 324	5%	61%	907	4 282	5%	61%
hamcrest-core	628	1 162	624	1 129	1%	3%	624	1 129	1%	3%
scalacheck_2.10	13 552	231 108	12 883	224 549	5%	3%	12 883	224 549	5%	3%
jackson-annotations	483	523	466	511	4%	2%	466	511	4%	2%
asm	405	1 782	392	1 744	3%	2%	392	1 744	3%	2%
jsr305	83	75	83	75	0%	0%	83	75	0%	0%
mean (over all projects)					12%	26%			12%	26%
std dev (over all projects)					12%	17%			12%	17%

#RM = number of reachable methods; #E = number of call edges. ↓ reduction in the number of reachable methods or call edges.

OPA vs CPA Finally, we measure how OPA and CPA affect our advanced call graphs. During our evaluation of LibCHA$_{\text{OPA}}$ and LibCHA$_{\text{CPA}}$, we observed a negligible reduction (0.41%) in the number of call edges from LibCHA$_{\text{CPA}}$ over LibCHA$_{\text{OPA}}$. In contrast, the number of entry points could be significantly reduced by 30% (cf. Section 7.3). This time, we compare XTA$_{\text{OPA}}$ and XTA$_{\text{CPA}}$. While we also compared the other algorithms, we do not discuss them here as their differences across the scenarios are of similar magnitude. Table 10.7 shows CG information of XTA$_{\text{OPA}}$ and XTA$_{\text{CPA}}$. Constructed under CPA, CGs generated by XTA generally provide 26% less edges and discover 18% less methods. The effect of CPA on type-set initialization (↓ 70%)—which is also influenced by the computed entry points—carries over to the final CG. However, it affects neither the number of reachable methods nor the number of all edges to the same extent.

Obs.33: Respecting the library-private implementation under CPA has a significant impact on the number of reachable methods and the number of call edges on advanced CG construction algorithms. Advanced algorithms have a better carry-over effect from CPA's initial benefits and, hence, can leverage the assumption better.

In Chapter 7 we found that library CGs for projects with little to no use of Java's visibility modifiers barely show a difference between OPA and CPA. We observe the same behavior for CGs from the set-based framework. Additionally, we find that an entity's type set can easily be polluted when an entry point declares at least one formal parameter with a type high up in the class hierarchy, e.g., Java's root type `java.lang.Object` is the worst-case scenario. Other examples are collection classes or a visitor design pattern with many elements. However, CGs with fewer sets are more vulnerable to the pollution of type sets as their type propagation is more coarse grained.

Obs.34: Advanced library CGs from the set-based framework do bring little to no improvement over a CHA when the target library a) does not make use of Java's visibility modifiers and b) declare entry points and accessible fields that use `java.lang.Object` or any other type with many subtypes.

10.3.4. Threats to Validity

One threat to the validity of our evaluation is the use of libraries from 2015. However, to enable comparability to the results presented in Chapter 7, we chose to use the same benchmark projects. Furthermore, most of the libraries are still under the top 100 most popular maven artifacts[10].

Additionally, the presented results could be subject to bugs in our implementation. To mitigate the risk of significant bugs, we tested the final call graphs with Cats (cf. Chapter 5) and compared the resulting algorithm profiles to Opal's RTA. Besides, we systematically wrote test cases concerning the general implementation and type propagation.

[10] `https://mvnrepository.com/popular` (checked on Apr 26, 2020).

Table 10.7.: Number of reachable methods and number of call edges produces by the XTA_{OPA} and XTA_{CPA} algorithms for the top and bottom five libraries from our data set.

Project	**XTA_{OPA}**		**XTA_{CPA}**			
	#RM	#E	#RM	#E	↓ #RM	↓ #E
jboss-logging	821	3 088	710	532	14%	83%
slf4j-log4j12	119	190	49	42	59%	78%
reflections	2 081	11 068	860	2 719	59%	75%
guice	4 567	22 422	2 172	7 288	52%	67%
maven-core	5 873	36 035	3 722	14 702	37%	59%
org.apache.felix- .scr.annotations	166	366	164	357	1%	2%
scalacheck_2.10	13 172	228 367	12 883	224 549	2%	2%
scalac-scoverage-plugin_2.11	8 133	19 685	8 120	19 671	0%	0%
jsr305	83	75	83	75	0%	0%
easymockclassextension	40	39	40	39	0%	0%
mean (over all projects)					18%	26%
std dev (over all projects)					12%	17%

#RM = number of reachable methods; #E = number of call edges. ↓ reduction in the number of reachable methods or call edges.

10.4. Conclusion

In this chapter, we discussed how to extend CG construction algorithms from the set-based framework to analyze libraries. In particular, we discussed how to apply the concepts presented in Chapter 7 to adapt RTA, CTA, MTA, FTA, and XTA algorithms to make them usable for the construction of library CGs.

These algorithms, in contrast to CHA, use type sets to approximate the receiver types at a given call site. When building application CGs, this set is usually empty initially and then filled with the types seen during CG construction. However, libraries are not closed worlds and their methods and fields can therefore be accessed externally. Thus, in addition to the extension presented in Chapter 7, the type sets of CGs from the set-based framework must be initialized, i.e., the type sets must reflect possible uses from the external world.

Our evaluation shows that these CG algorithms are more precise than $\text{RTA}_{\text{CPA}}$and, thus, also improve over $\text{LibCHA}_{\text{CPA}}$. Other than $\text{LibCHA}_{\text{CPA}}$, our set-based algorithms can leverage the reduced number of entry points and contain significantly fewer call edges and fewer reachable methods. However, the gain in precision comes with a price: The CGs must hold and merge many large sets, which considerably decreases their performance. Nevertheless, the increase in precision is with 10% fewer reachable methods and 25% fewer call edges beneficial.

Our modular implementation of the set-based CG framework allows instantiating many more CG variants. Some of these algorithms may exhibit different runtime behavior than the evaluated algorithms. Moreover, the modular design facilitates the support of individual language features and APIs that, when implemented once, can be reused for all algorithms of the framework.

Part IV.

Conclusion and Outlook

11. Conclusion

In this chapter, we present a brief overview of the findings and contributions of this thesis. We start with a summary of our results and follow with a closing discussion.

11.1. Summary of Results

We provide a holistic view on sources of unsoundness of CG construction algorithms and on the state-of-the-art in CG construction. We provide an automated test framework, Cats, which we use to compare existing CG algorithms qualitatively. Based on Cats and Hermes, our code-query engine for assessing and constructing minimal project corpora for benchmarking software analysis techniques and tools, we present a methodology and a toolchain, Judge, for systematically evaluating a CG algorithm's sources of unsoundness in a project-specific manner.

Additionally, we discussed the design space of CG algorithms for libraries and presented OPA and CPA algorithms suiting different needs. To implement the propagation-based library CG algorithms, we used our framework for collaborative modular static analyses that eases the development of modular CG algorithms, allowing pluggable precision, sound(i)ness, and scalability. Finally, we present improved library call graphs, as a cornerstone for future work.

Comprehending and Minimizing Corpora We proposed Hermes, our code-query engine, to better understand available test corpora and to compute corpora that enable effective testing of static analysis techniques. We find that we can already use very primitive queries to better comprehend the nature of given corpora better. Furthermore, Hermes can be used to generate minimized integration test suite that are feature-wise equivalent to a large corpus. As Hermes is extensible, researchers can quickly write new queries to search arbitrary real-world projects suiting their needs and, thereby, enable meaningful evaluations.

Automated Test Suite for Call-graphs Algorithms We studied the Java Virtual Machine Specification and reviewed state-of-the-art CG construction algorithms to identify what introduces sources of unsoundness in CGs. Using the sources of unsoundness we identified, we build Cats, the first automated test framework for CG algorithms. Cats enables systematic, comparable, and reproducible experiments, measuring and documenting a CG algorithm's capabilities. Therefore, it enables to document which features and APIs a CG construction algorithm supports. Moreover, Cats is easily extensible by new test cases and additional CG construction algorithms for further experiments.

Performance of State-of-the-art Call-graph Algorithms We use CATS for a systematic evaluation and comparison of CG algorithms from four state-of-the-art static analysis frameworks. We find that all of their CG algorithms support rudimentary programming language features and APIs. Otherwise, they exhibit uneven support for other features, passing on average only ≈60% of our test cases. While the failing test cases partially cover dynamic program behavior (e.g. reflection), others pertain to newly introduced features from Java 8 or newer. Only 29% of tests were passed successfully by all algorithms, i.e., they provide a vastly different feature and API support. In addition to varying feature support, we find that CG algorithms from different frameworks employ different design decisions, leading to large differences in the constructed CGs. Unfortunately, that makes them hardly comparable.

We suggest using CATS as a reference or test suite to improve the evaluated CG algorithms and increase their comparability.

Prevalence of Java Features and APIs In-the-wild Using CATS' test cases for sources of unsoundness in CG algorithms, we developed HERMES queries that find the responsible features in real-world applications. We employed these queries to investigate the relevance of these sources of unsoundness in-the-wild. Among others, our findings include that a) Java 8 support is a must for most code bases, b) serialization and reflection is frequently used, c) many features are only required in specific scenarios, and d) a lot of commonly used evaluation corpora (e.g., Qualitas Corpus [TAD+10]) are not up-to-date in terms of coverage of recent features and APIs. As our infrastructure is open source, researchers can verify these results or investigate these features' prevalence on additional corpora.

Project-specific Unsoundness Analysis for Call Graphs We use JUDGE for a project-specific evaluation of a call graph's sources of unsoundness. Our case study on *Xalan* revealed that even mid-sized programs use many unsupported features, leading to unsound CGs. Furthermore, we find that these unsupported features can have a devastating effect, e.g., OPAL's CG contained only ≈0.3% of Xalan's methods. JUDGE's findings enabled us to quickly identify the problem's root cause, empowering us to improve OPAL's CG. Using our experiments' results, we derived several implications for framework developers and static analysis researchers to improve CG construction algorithms, their comparability, and their documentation.

Design Space for Library Call-graph Algorithms We motivate the need for CG algorithms dedicated to libraries and present a thorough discussion of the design space for such algorithms. We propose two concrete algorithms for libraries based on adaptations of the CHA algorithm within that design space: One algorithm can be used to identify security issues ($\text{LIBCHA}_{\text{OPA}}$), and the other algorithm is meant to identify general software quality issues ($\text{LIBCHA}_{\text{CPA}}$). Evaluating these algorithms shows that we need two types of algorithms to address a library analysis's individual needs. Additionally, with $\text{LIBCHA}_{\text{CPA}}$, we discovered 550 dead methods within in the Java Development Kit.

Modular Construction of Call Graphs Our findings from our studies concerning the support of programming language features and APIs motivated the need for modular CG construction and we proposed such a framework as part of a more generalized framework for modular static analyses. Our approach allows various isolated, orthogonal analyses for individual language features and APIs to collaboratively compute a single CG. Therefore, it facilitates systematic investigation of different configurations, supporting users and developers in finding the best trade-off between precision, sound(i)ness, and scalability.

Improved Library Call Graphs Using our framework for collaborative static analysis and our library design space, we investigated how to adapt the CG algorithms of the set-based framework [TP00] for libraries. After discussing how to extend these CGs, we present five algorithms that use the *open-package assumption* (OPA) and five algorithms that use the *close-package assumption*. While all CGs show similar improvements in precision, the algorithms based on RTA are the fastest, and the algorithms based on XTA are the most precise. Besides, we find that the algorithms are less scalable when used on libraries. However, the set-based framework facilitates the instantiation of many more CG algorithms that might scale better, which however needs more investigations in the future.

11.2. Closing Discussion

After summarizing this thesis' contributions and findings, we will briefly relate them to current trends and look at future challenges.

Interprocedural static analysis tools are an integral ingredient to ensure software quality. Software developers use such tools on their programs. Thereby, they must consider which kind of issues they are interested in and the nature of the software they are analyzing. The work presented in this thesis shows that the call graphs (CG) these tools build on are only partially up to the task, i.e., they are unsound concerning various language features and APIs and do not reflect the nature of libraries sufficiently.

Researchers have dedicated much work to the construction of CG algorithms. This thesis consolidates several decades of research for qualitative and quantitative assessment of the state-of-the-art. We find that existing CG algorithms cover only a small subset of language features and APIs relevant in real-world programs. Moreover, we find that even CG algorithms covering similar features, result in CGs that differ significantly, i.e., rendering even the comparison of the same base algorithm implemented in two different frameworks impossible. Furthermore, we investigate their sources of unsoundness and compare CGs to determine the respective root causes. Based on our findings, we provide recommendations to both static analysis researchers and framework developers.

Despite the maturity of CG algorithms, their practical applicability must be further improved. Existing CG construction algorithms focus only on precision/scalability and on analyzing applications (cf. Chapter 2). Furthermore, they support only approximately 52% of relevant language features (cf. Chapter 5). The work presented in this thesis

shows that with a through systematic analysis of the problems of state-of-the-art CG algorithms, we can pinpoint their weaknesses and propose a design space that shows how to adapt library CGs to the needs of client analyses. At the same time, we propose a modular framework that eases to address these weaknesses in isolation and allows high customization of CG algorithms. Our results on library CG construction algorithms show that there is further potential for improvement, which is why we are convinced that CGs can become an even better foundation for interprocedural static analyses. In order to reach this goal, we briefly present challenges that future work faces.

First, current CG construction algorithms do not support many programming language features and APIs relevant in practice. Although some of these features are hard to support statically [LSS+15], we must develop new abstractions that beneficially trade-off precision, scalability, and sound(i)ness concerning the analyzed program. JUDGE can guide the development of these abstractions.

Second, CG algorithms from different frameworks are subject to different design designs, e.g., they rely on type information provided by different intermediate representations. Our empirical results show that these decisions highly influence the outcome of CG construction. We should systemically research and document these design decisions to better understand their effects on client analyses. Otherwise, the results of two static analyses using different CGs are hardly comparable.

Third, CG algorithms for libraries are still in an early stage of research. Unsoundness of CGs is deliberately accepted to improve precision and, thus, to reduce the number of false positives on client analyses [LSS+15]. Analogously, the community might accept unsound library CGs. With call-by-signature resolution, we propose a technique that is indispensable to receive a sound library CG while reducing a CG's precision. However, our results on library CGs indicate that it is essential to distinguish between the *open-package assumption* (OPA) and *closed-package assumpiton* (CPA). For the latter, we provide empirical evidence that these algorithms are beneficial. Nevertheless, we should further research the impact of library CGs on client analyses. Primarily, we must show that OPA-algorithms positively affect security analyses. Simultaneously, we need to research ways to improve CG algorithms' precision for libraries further, e.g., by using escape analysis to approximate the library-private implementation more accurately or using machine learning to learn from actual library usages about the actual types passed to entry points. Our results on library CGs further show that Java's visibility modifiers are crucial for CGs operating under CPA. We should develop analyses that support developers to decide if they can declare classes, methods, and fields with more restrictive visibility modifiers, e.g., private or package visibility.

Fourth, our results show that the customization of CGs and trading-off precision, scalability, and sound(i)ness is essential when analyzing real-world applications. While our modular CG-algorithms framework shows how to customize CG algorithms, it is not apparent how we can reuse our analyses for individual features across context-sensitive CG algorithms. Context-sensitivity has many facets (e.g., object sensitivity or callsite sensitivity) that must either be compatible with each analysis module or one must combine the general CG algorithm with context-insensitive modules. Moreover, it would be attractive to employ self-adaptive analyses that automatically add a module when

the analysis encounters a specific language feature and, therefore, requires support. A self-adapting analysis would reduce not only the configuration efforts but also improve the CG algorithms performance.

12. Future Work

In this chapter, we present our ideas for future. First, we discuss future work with respect to creating benchmarks for static analyses. Second, we present ideas how to test and benchmark call-graph (CG) algorithms. Last, we will explore possible directions for follow-up work in the area of CG construction.

12.1. Benchmarking Static Analyses

Creating Use-case-based Corpora Even though we can use Hermes to create evaluation corpora that suit our needs, the researchers must publish the created data set and the queries to create transparency and enable comparison. If the relevant data is not published, the data set bears the danger of not being representative, e.g., problematic projects could be unknowingly removed or the queries could not represent the problem. To mitigate this threat, we image a similar system to the Normalized Java Resource (NJR) [PL18]. NJR's goal is to provide an infrastructure that consists of 100 000 executable Java programs that can be queried to find projects based on dynamic measurements. However, Hermes uses static measurements, which are better tailored to the needs of static analysis research. Future work should explore how Hermes can be used to systematically create data sets from common sources (e.g., Maven Central, GitHub, or app stores) based on a common but extensible set of metrics. The *Delphi* project[1] makes the first step in that direction by running Hermes' queries on artifacts from Maven Central, creating a searchable index that allows users to find artifacts with particular features. Yet, Delphi neither allows to publish created data sets nor to add custom queries.

Minimizing Corpora With Hermes, we advocate for a more detailed inspection of the corpora to assure their representativeness while keeping the evaluation data set minimal to assure a practical evaluation. Still, the generated set is only minimal in the sense that each feature occurs within the subset. While it is globally configurable how often a feature must occur, it is not possible to weight certain features over others. Hence, Hermes' optimizer is rather primitive and does not support multi-objective optimizations. We can extend its optimizer by setting the following two goals: the first is the broadest most coverage of metrics relevant to the analysis. The second is the minimum of a size metric such as the number of projects, number of classes, number of instructions, or file size. The first criterion ensures that our evaluation dataset is

[1] `https://delphi.cs.uni-paderborn.de/` (checked on May 13, 2020).

representative for the analysis targeted. The second criterion ensures the smallest input size, which we assume to affect the analysis runtime directly.

Using this methodology, researchers can create evaluation sets for their respective program analyses, targeted at the analysis purpose while keeping the effort minimal. This ensures that analyses have been properly tested on every case that might occur in real-world code, without making the evaluation run longer than necessary. Additionally, it saves valuable time for analysis experts and may expose implementation issues much faster than previous efforts.

Analysis experts must specify the criteria that their analysis is sensitive to, i.e., develop the queries that derive features relevant to their evaluation and if necessary also prioritize the derived features. Different analysis types may react differently to various program size measures. Therefore, it is also necessary that analysis experts select optimization criteria for program size, such as the number of methods, number of fields, number of callsites, or file size.

12.2. Evaluating Call-graph Algorithms

Improve Test Framework Infrastructure Cats uses two kinds of test cases, *standard* test cases with automatic compilation and *advanced* test cases that require a manual compilation process. Among other cases, these tests comprise features that belong to Java 9 or higher, which is no longer compatible with lower Java versions and, therefore, need a different compiler. The latter kind of test cases is tedious to add to the test suite.

Future work should extend Cats test case compilation. First, the compilation process must be modularized to support multiple compilers, e.g., one for each Java version. Then, one can easily add new compilers on new Java releases. Second, each test case should then specify the required compiler. This extension would improve Cats' extensibility, i.e., one can add new test cases for new features and APIs only becoming relevant to newer recent Java versions. Furthermore, it would improve the transparency of existing tests, as some are only available as pre-compiled units. Another area for future work would be to broaden the test suite's scope by adding test cases related to practice-relevant frameworks, such as Android.

Test Suite with Precision We found evidence that CG construction algorithms that are implemented using different frameworks significantly vary in precision (cf. Section 6.2.3). For example, the RTA algorithms from Soot, Wala, and Opal exhibited different behavior due to implementation design decisions. Unfortunately, Cats measures only recall and, therefore, is only useful as a reference for checking feature support.

Hence, future work should investigate how to build a framework for testing a CG's precision, i.e., we need to test whether a call edge should be or should not be contained in a CG built by a specific algorithm. Like in Cats, one could define the test suit's ground truth via annotations. These annotations could then specify for each callsite all possible call targets. Additionally, one must define for each call targets which CG algorithm (e.g., RTA, XTA, VTA, or 0-1-CFA) must contain the respective call edge.

This manual process is laborious and error-prone. Groove and Chambers [GC01] discussed many existing CG algorithms and arranged them within a lattice ordered by their precision. For example, such a lattice allows one to state that a call edge should not be contained in a CG more precise than RTA. Hence, it would be correct for RTA, CHA, or signature-based CGs to contain this call edge but not for VTA, XTA, or others. However, not all CG algorithms are directly comparable in terms of precision, e.g., FTA and MTA are orthogonal to each other such that neither FTA is a subset of MTA nor the other way around. Using such a lattice while annotating test cases concerning precision could tremendously reduce the annotation effort. Instead of specifying every algorithm individually, it would be possible to specify different, comparable CG algorithms.

Studying the Impact of Unsoundness on Client Analyses Current CG construction algorithms deliberately lack support for many features. While this indirectly increases the precision of interprocedural client analyses, it remains unknown what percentage of findings are missed due to the CG's sound(i)ness. However, some features (e.g., reflection) are not entirely statically supportable and developing naïve support can lead to high imprecision that might impede the scalability of client analyses [BSS+11]. Hence, we must find approximations with reasonable trade-off between sound(i)ness, precision, and scalability. Independently from problematic features, future work should investigate the effect of not supporting individual features on the results of various client analyses. Our configurable modular CG framework would be beneficial for such an investigation.

12.3. Advancing Call-graph Construction

Leveraging Advanced Static Analysis Techniques One of the biggest weakness of type-based CG algorithms is the lack of precise information at callsites that involve large type hierarchies. The prime example is a `toString()` method call on a receiver typed with `java.lang.Object`. Other examples of frequently used APIs are `java.lang.Iterator`'s `next()`, `hasNext()`, or `remove()` methods. Additionally, using types with many subtypes as formal method parameters, return types, or as field types, potentially leads to an enormous type flow in CGs build with the set-based framework. Also, due to the lack of information on the external world when constructing library CGs, we must assume worst-case scenarios, e.g., all existing and accessible types are used outside the library.

To gather additional information at callsites, one could apply advanced static analysis techniques. OPAL's architecture for collaborative static analysis facilitates the use of information derived by such static analyses while constructing CGs. In the following, we discuss three such analyses that future work should investigate:

a) CG algorithms should employ *type refinement*, an analysis that refines method signatures or a field's declared type, if possible. It can be employed for all method parameters, method return types, and fields, where all usage contexts are known. That is, a whole-program analysis can employ type refinement for all methods and fields, while a library analysis can use this technique only for elements of the library-private implementation.

```
1  class Parser {
2
3    final Map<Long, String> fieldValues = new HashMap<>();
4
5    private void doSomething() {
6      Iterator<Entry<Long, String>> it = fieldValues.entrySet().iterator()
7      /* ... */
8    }
9  }
10
11 class HashSet <K,V> extends AbstractMap<K,V>
12   implements Map<K,V>, Cloneable, Serializable {
13
14   public Set<Map.Entry<K,V>> entrySet() {
15     Set<Map.Entry<K,V>> es;
16     return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;
17   }
18
19   final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
20     public final Iterator<Map.Entry<K,V>> iterator() {
21       return new EntryIterator();
22     }
23   }
24 }
```

Listing 12.1.: Code example to demonstrate the benefits of type refinement.

b) CG algorithms should employ *immutability analysis* that determines whether a given reference is read-only or an object immutable.

c) CG algorithms should employ *escape analysis* to newly instantiated objects, to see whether these objects escape the local scope. If objects are only used locally, they must not be considered as globally instantiated.

The first analysis could be precious for the type-based CG algorithms discussed in this thesis, as refined method signatures could significantly reduce the number of propagated types. Fortunately, many programs use specialized types, thereby providing knowledge in code that we can leverage to refine types and thus constrain type sets. A good example is Java's collection API structure, which defines many specialized types for various collections, such as lists, sets, or maps. Listing 12.1 shows simplified code from `java.time.format.Parsed` and `java.util.HashSet` that indicates the possible gains from type refinement. When we consider the method call chain at Line 6, the number of involved types is immense when we only consider the declared types. First, `entrySet()` is called on `fieldValues` which is of type `java.util.Map`. The latter has over 100 overrides that all return a `java.util.Set` which has in turn over 90 subtypes. Then the `entrySet`'s result is used to retrieve the final iterator, which has more

than 250 subtypes. When all these types are considered instantiated, this simple call chain introduces an incredible amount of call edges. However, that must not be the case when we use refined type signatures. Then `fieldValues` (Line 3) would be known to be `java.util.HashMap`, the `entrySet` method (Line 16) would be known to return `java.util.HashMap$EntrySet`, and the iterator (Line 21) would be known to be `java.util.HashMap$EntryIterator`. Using type refinement might reduce the number of types that will be propagated to a method or field. However, we can only apply type refinement to methods and fields where all usage contexts are known, i.e., they belong to the library-private implementation. When we additionally employ an immutability analysis, an analysis that determines whether an object or a reference is immutable, we can determine effectively final fields[2]. The latter's type can then also be refined. While an immutability analyses is hard to incorporate in a monolithic analysis, it is trivially possible within OPAL's modular static analyses framework.

We could also employ an escape analysis to improve the CG. Some classes, e.g., iterators or comparators, have a large type hierarchy but are often used only locally, i.e., their objects are only used within a single method. Yet, once such an object is instantiated, e.g., when using a RTA, it is considered at every other relevant callsite. However, when an instantiated type never leaves its scope, it must be considered only locally. Furthermore, escape analyses can be used to approximate the library-private implementation more precisely. For example, internal classes that provably do not escape the library-private implementation must never be considered instantiated in the external world. Moreover, Java 9 introduced a new module system where applications and libraries can be divided into modules. Each module must declare its public interface and, thus, can be separated in its public API and its module-private implementation. Again, escape analysis can be employed to determine which types escape the module and, therefore, must be considered within other modules.

Improving the Call-graph Product-line Our novel framework for modular CG construction enables CGs to be modularly composed of different modules—each handling a specific, well-delimited language feature—that are reusable across any method for resolving call targets. We identified such language features in Chapter 5. While our approach has multiple benefits, we investigated only the modeling of context-insensitive CGs. We argue that it would be interesting to extend the approach to cover both context-insensitive and context-sensitive CGs of various levels of complexity, performance, and precision. Thereby supporting algorithms ranging from CHA, over set-based algorithms, to points-to-based call graphs. This might be achieved by introducing modules that abstract over the concrete properties stored in the blackboard to allow individual CG modules to be agnostic of the concrete source of type information required to resolve calls. Such a system would enable the ultimate framework for CG construction that is highly configurable with respect to precision, scalability, and sound(i)ness.

[2] An effectively final field is a field that is not declared *final* but is written only once.

Approximating the External World Our results show that library CGs must consider the external world and that how we approximate the external world significantly influences the CG's precision. We believe that it is possible to improve our model of the external world by gathering more context about the analyzed library. A promising source to approximate the context could be actual clients of the library. Instead of analyzing the library in the context of a concrete application, we can preanalyze a number of known clients. To get to know a library's clients, we could either scan build files from code repositories (e.g., GitHub) or use data collected by Maven Central as they provide dependee information for published libraries. For each client, we could then extract a) which part of the library they access and b) which types they pass to the library. Hence, when constructing a library CG, we can use all previously gathered library clients to approximate the library's entry points, accessed fields, instantiated types, and transferred types. Future work should investigate if such approximations benefit library CGs precision and scalability as they only abstract over known scenarios instead of all theoretically possible use cases.

Contributed Implementations and Data

Working on this thesis's research topics, we have implemented several research prototypes and collected data. We provide both implementations and datasets to enable other researchers to comprehend our work and support further research. We believe that this fosters good scientific practice and ask others to follow this example.

Hermes

We provide the implementation of Hermes as part of Opal in the following repository:

`https://github.com/stg-tud/opal/tree/develop/TOOLS/hermes`

Hermes builds on top of Opal. However, it can be used separately as a command-line tool. Within its resource folder lies a *Readme* given instructions on how to use it. Additionally, the resource folder provides one markdown file per query, explaining its features.

Cats and Judge

We provide the implementation of Cats and Judge in the following repository:

`https://bitbucket.org/delors/cats/src/master/`

The repository contains many independent projects that modularize the actual test framework, the test cases, and all test adapters. The repository contains a detailed description explaining its structure and demonstrates how to add test cases.

Modular Call-graph Construction Framework

We provide the implementation of our modular framework for call graphs and static analysis in general as well as our library call graphs here:

`https://github.com/stg-tud/opal/`

The framework became an integral part of Opal's static analysis infrastructure. The element, the blackboard, can be found under the name `PropertyStore`. Furthermore, all analyses presented in Section 8.3 can be found under their respective name.

Study Artifacts

We provide the results of our evaluation and study pertaining to Hermes, Cats, and Judge within the following docker container:

`https://hub.docker.com/r/mreif/jcg`

We provide the results of our evaluation and our dataset pertaining to our study of library call graphs here:

`http://www.st.informatik.tu-darmstadt.de/artifacts/dlc/`

The results of our evaluation of the modular framework for static analysis and a docker container to reproduce them are provided here:

`https://doi.org/10.5281/zenodo.3972736`

`https://hub.docker.com/r/mreif/blast`

Bibliography

[Aik99] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2-3):79–111, 1999.

[AKS15] Nicholas Allen, Padmanabhan Krishnan, and Bernhard Scholz. Combining type-analysis with points-to analysis for analyzing java library source-code. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–18. ACM, 2015.

[AL12] Karim Ali and Ondřej Lhoták. Application-only call graph construction. In *European Conference on Object-Oriented Programming*, pages 688–712. Springer, 2012.

[AL13] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming*, pages 378–400. Springer, 2013.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, Patrick McDaniel, Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM SIGPLAN Notices*, 49(6):259–269, June 2014.

[ARL+14] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. Constructing call graphs of scala programs. In *European Conference on Object-Oriented Programming*, pages 54–79. Springer, 2014.

[ARL+15] Karim Ali, Marianna Rapoport, Ondřej Lhoták, Julian Dolby, and Frank Tip. Type-based call graph construction algorithms for scala. *ACM Trans. Softw. Eng. Methodol.*, 25(1):9:1–9:43, December 2015.

[BBFG04] Frederic Besson, Tomasz Blanc, Cédric Fournet, and Andrew D Gordon. From stack inspection to access control: A security analysis for libraries. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 61–75. IEEE, 2004.

[BBM96] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering*, 22(10):751–761, 1996.

[BC91] Antonio Brogi and Paolo Ciancarini. The concurrent language, shared prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):99–123, 1991.

[BDF01] Massimo Bartoletti, Pierpaolo Degano, and GianLuigi Ferrari. Static analysis for stack inspection. *Electronic Notes in Theoretical Computer Science*, 54:69–80, 2001. ConCoord: International Workshop on Concurrency and Coordination (Workshop associated to the 13$^{\text{th}}$ Lipari School).

[BGH+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[BHT07] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518. Springer, 2007.

[BHT08] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. Program analysis with dynamic precision adjustment. In *2008 23$^{\text{rd}}$ IEEE/ACM International Conference on Automated Software Engineering*, pages 29–38. IEEE, 2008.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*, volume 1 of *Pattern-Oriented Software Architecture*. Wiley, 1996.

[Bod12] Eric Bodden. Inter-procedural data-flow analysis with IFDS/IDE and Soot. In *SOAP*, pages 3–8, July 2012.

[Bod18] Eric Bodden. The secret sauce in efficient and precise static analysis. In *International Workshop on State Of the Art in Java Program analysis,(SOAP)*, pages 84–92, 2018.

[BS96] 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*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM.

[BS09a] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA, pages 1–12. ACM, 2009.

[BS09b] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. Association for Computing Machinery.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE, pages 241–250. ACM, 2011.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL, pages 238–252. ACM, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL, pages 269–282. ACM, 1979.

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Annual Asian Computing Science Conference*, ASIAN, pages 272–300. Springer, 2006.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *OOPSLA*, 1999.

[Cha06] Byeong-Mo Chang. Static check analysis for java stack inspection. *SIGPLAN Not.*, 41(3):40–48, March 2006.

[Cor89] Daniel D Corkill. Design alternatives for parallel and distributed blackboard systems. In *Blackboard Architectures and Applications*, Perspectives in Artificial Intelligence. 1989.

[Cor91] Daniel D Corkill. Blackboard systems. *AI expert*, 6(9):40–47, 1991.

[Cra88] Iain D Craig. Blackboard systems. *Artificial Intelligence Review*, 2(2):103–118, 1988.

[Cra93] Iain D Craig. A new interpretation of the blackboard metaphor. Technical report, Technical report, Department of Computer Science University of Warwick, 1993.

[DCYB09] Roland Dodd, Andrew Chiou, Xinghuo Yu, and Ross Broadfoot. Industrial process model integration using a blackboard model within a pan stage decision support system. In *2009 Third International Conference on Network and System Security (NSS)*, NSS, pages 489–494. IEEE, 2009.

[DEB16] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. Toward an automated benchmark management system. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, pages 13–17. ACM, 2016.

[DGC95] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101. Springer, 1995.

[DGHL91] Keith Decker, Alan Garvey, Marty Humphrey, and Victor R Lesser. Effects of parallelism on blackboard system scheduling. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 15–21, 1991.

[DHS15] Jens Dietrich, Nicholas Hollingum, and Bernhard Scholz. Giga-scale exhaustive points-to analysis for java in under a minute. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 535–551, New York, NY, USA, 2015. ACM.

[DJP10] Delphine Demange, Thomas Jensen, and David Pichardie. A provably correct stackless intermediate representation for java bytecode. In *Asian Symposium on Programming Languages and Systems*, pages 97–113. Springer, 2010.

[DSST17] JB Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. Xcorpus–an executable corpus of java programs. 2017.

[Duj10] Jozo Dujmović. Automatic generation of benchmark and test workloads. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 263–274. ACM, 2010.

[DWA20] Lisa Nguyen Quang Do, James Wright, and Karim Ali. Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 2020.

[EH07] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the ACM on Programming Languages*, OOPSLA, 2007.

[EH14] Michael Eichberg and Ben Hermann. A software product line for static analyses: the OPAL framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM.

[EHRLR80] Lee D Erman, Frederick Hayes-Roth, Victor R Lesser, and D Raj Reddy. The hearsay-ii speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys (CSUR)*, 12(2):213–253, 1980.

[EKH+18] Michael Eichberg, F Kübler, D Helm, M Reif, G Salvaneschi, and M Mezini. Lattice based modularization of static analyses. In *ISSTA Companion/ECOOP Companion*. ACM, 2018.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, ICSE, pages 391–400. ACM, 2008.

[EMK+06] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. Integrating and Scheduling an Open Set of Static Analyses. ASE, 2006.

[Far86] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *ACM SIGPLAN Notices*, volume 21, pages 85–98. ACM, 1986.

[FKS18] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 209–220, New York, NY, USA, 2018. ACM.

[FL10] Manuel Fähndrich and Francesco Logozzo. Clousot: Static contract checking with abstract interpretation. In *International Conference on Formal Verification of Object-oriented Software*, FoVeOOS, pages 10–30. Springer, 2010.

[FS19] George Fourtounis and Yannis Smaragdakis. Deep static modeling of invokedynamic. In *33rd European Conference on Object-Oriented Programming*, 2019.

[GAE+17] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: Obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 638–648, New York, NY, USA, 2017. Association for Computing Machinery.

[GC01] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[GJS+18a] J Gosling, B Joy, G Steele, G Bracha, A Buckely, and D Smith. *The Java Language Specification, 2018*. Oracle Ameria, 2018.

[GJS+18b] J Gosling, B Joy, G Steele, G Bracha, A Buckely, and D Smith. *The Java Virtual Machine Specification, 2018*. Oracle Ameria, 2018.

[GMB+20] Leonid Glanz, Patrick Müller, Lars Baumgärtner, Michael Reif, Sven Amann, Pauline Anthonysamy, and Mira Mezini. Hidden in plain sight: Obfuscated strings threatening your privacy. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, pages 694–707, New York, NY, USA, 2020. Association for Computing Machinery.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[HGES16] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. Reactive Async: expressive deterministic concurrency. SCALA, 2016.

[Hic18] Rich Hickey. Clojure language. `https://clojure.org/`, [Online; accessed 24-August-2018].

[HKE+18] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. A unified lattice model and framework for purity analyses. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 340–350, 2018.

[HKK+20] Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. A programming model for semi-implicit parallelization of static analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2020.

[HKR+20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 184–196, 2020.

[HM12] Wei Huang and Ana. Milanova. ReImInfer: Method purity inference for Java. FSE, 2012.

[HMDE12] Wei Huang, Ana Milanova, Werner Dietl, and Michael D Ernst. Reim & reiminfer: Checking and inference of reference immutability and method

purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA, pages 879–896, 2012.

[HP18] Andrew Habib and Michael Pradel. How many of all bugs do we find? a study of static bug detectors. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 317–328. IEEE, 2018.

[HREM15] Ben Hermann, Michael Reif, Michael Eichberg, and Mira Mezini. Getting to know you: Towards a capability model for java. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 758–769, New York, NY, USA, 2015. ACM.

[HVDM06] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming*, ECOOP, pages 2–27. Springer, 2006.

[IBM] IBM. Wala - static analysis framework for java. `http://wala.sourceforge.net/`. [Online; accessed 19-APRIL-2018].

[Inc18] Artima Inc. ScalaTest. `http://www.scalatest.org/`, [Online; accessed 24-August-2018].

[JFB+17] Nick P Johnson, Jordan Fix, Stephen R Beard, Taewook Oh, Thomas B Jablin, and David I August. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO, pages 148–159. IEEE Press, 2017.

[JMT10] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, pages 320–339. Springer, 2010.

[Jon90] Larry G Jones. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):429–462, 1990.

[JSMHB13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, CAV, pages 422–430. Springer, 2016.

[KE19] Sven Keidel and Sebastian Erdweg. Sound and reusable components for abstract interpretation. In *Proceedings of the ACM on Programming Languages*, volume 3 of *OOPSLA*, page 176. ACM, 2019.

[KKD+11] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 217–224, 2011.

[KM05] Thomas Kotzmann and Hanspeter Mössenböck. Escape analysis in the context of dynamic compilation and deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, VEE, 2005.

[KNR+17] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936, 2017.

[Knu68] Donald E Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[kot] Kotlin Language. `https://kotlinlang.org/`. [Online; accessed 24-August-2018].

[KPE18] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. Compositional soundness proofs of abstract interpreters. In *Proceedings of the ACM on Programming Languages*, volume 2 of *ICFP*, page 72. ACM, 2018.

[KPK02] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 359–372, New York, NY, USA, 2002. ACM.

[KS13] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Notices*, volume 48, pages 423–434. ACM, 2013.

[Lau18] École Polytechnique Fédérale Lausanne. Scala Language. `https://www.scala-lang.org/`, [Online; accessed 24-August-2018].

[LBLH11] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[LDDC+95] Hongyi Li, Rudi Deklerck, Bernard De Cuyper, A Hermanus, Edgard Nyssen, and Jan Cornelis. Object recognition in brain ct-scans: knowledge-based fusion of data from multiple feature extractors. *IEEE Transactions on medical imaging (TMI)*, 14(2):212–229, 1995.

[LGC02] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. POPL, 2002.

[LH03] Ondřej Lhoták and Laurie Hendren. Scaling java points-to analysis using s park. In *International Conference on Compiler Construction*, pages 153–169. Springer, 2003.

[Lho07] Ond∨rej Lhoták. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*, pages 37–42, New York, New York, USA, June 2007. ACM Press.

[Liv05] Benjamin Livshits. Defining a set of common benchmarks for web application security. 2005.

[Liv18] B. Livshits. SecuriBench Micro. `https://suif.stanford.edu/~livshits/work/securibench-micro/`, [Online; accessed -August-2018].

[LL05] 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.

[LLA+15] Xiaoni Lai, Zhaoyi Luo, Karim Ali, Ondřej Lhoták, Julian Dolby, and Frank Tip. Evaluating call graph construction for jvm-hosted language implementations. Technical Report CS-2015-03, University of Waterloo, David R. Cheriton School of Computer Science, 2015.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[LSBM15] Johannes Lerch, Johannes Spath, Eric Bodden, and Mira Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbounded Access Paths (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 619–629. IEEE, November 2015.

[LSS+15] 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.

[LTSX14] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for java. In *European Conference on Object-Oriented Programming*, pages 27–53. Springer, 2014.

[LTX15] Yue Li, Tian Tan, and Jingling Xue. Effective soundness-guided reflection analysis. In *International On Static Analysis*, pages 162–180. Springer, 2015.

[LTX19] Yue Li, Tian Tan, and Jingling Xue. Understanding and analyzing Java reflection. *ACM Transactions on Software Engineering and Methodology*, 28(2), 2019.

[LWL+05a] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12. ACM, 2005.

[LWL05b] Benjamin Livshits, John Whaley, and Monica S Lam. Reflection analysis for java. In *Asian Symposium on Programming Languages and Systems*, pages 139–160. Springer, 2005.

[LWR20] Yi Lu, Daniel Wainwright, and Michael Reif. Probabilistic call-graph construction, May 2020. US Patent App. 16/200,045.

[MH07] Eva Magnusson and Görel Hedin. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[MNGL98] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):158–191, 1998.

[MPM+15] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 50 of *OOPSLA 2015*, pages 695–710, New York, NY, USA, 2015. ACM.

[Muc97] Steven Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[Mvn18] MvnRepository. Maven - Popular Projects.a. `https://mvnrepository.com/popular`, [Online; accessed 15-July-2018].

[MYL16] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. In *ACM SIGPLAN Notices*, volume 51, pages 194–208. ACM, 2016.

[NAW06] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[New62] Allen Newell. Some problems of basic organization in problem-solving programs. Technical report, Rand Corp Santa Monica CA, 1962.

[NFA82] H Penny Nii, Edward A Feigenbaum, and John J Anton. Signal-to-symbol transformation: Hasp/siap case study. *AI magazine*, 3(2):23–23, 1982.

[Nii86] H Penny Nii. The blackboard model of problem solving and the evolution of blackboard architectures. *AI magazine*, 7(2):38–38, 1986.

[NNH05] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. 2005.

[ÖH17] Jesper Öqvist and Görel Hedin. Concurrent circular reference attribute grammars. In *10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE, pages 151–162. ACM, 2017.

[PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, CASCON, page 10. IBM Press, 2000.

[PFL16] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[PL18] Jens Palsberg and Cristina V Lopes. Njr: a normalized java resource. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 100–106, 2018.

[PM14] Edgar Pek and P Madhusudan. Explicit and symbolic techniques for fast and scalable points-to analysis. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP, pages 1–6. ACM, 2014.

[pro18a] The Apache Groovy project. Groovy Language. `http://groovy-lang.org/`, [Online; accessed 24-August-2018].

[Pro18b] The Opal Project. The opal project. `http://opal-project.de/`, [Online; accessed 23-August-2018].

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspot tm server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, volume 1, 2001.

[RC00] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM.

[REH+16] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. Call graph construction for Java libraries. In *Proceedings of the 2016 24^{th} ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 474–486, 2016.

[REHM17] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. Hermes: assessment and creation of effective test corpora. In *Proceedings of the 6^{th} ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 43–48, 2017.

[Rep95] Thomas W Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, pages 163–196. Springer, 1995.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22^{Nd} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[RKE+19] 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 28^{th} ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 251–261, 2019.

[RKEM18] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Proceeding ISSTA '18 Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, SOAP 2018, pages 107–112, 2018.

[RKH+20] Michael Reif, Florian Kübler, Dominik Helm, Ben Hermann, Michael Eichberg, and Mira Mezini. Tacai: An intermediate representation based on abstract interpretation. In *Proceedings of the 9^{th} ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2020, pages 2–7, New York, NY, USA, 2020. Association for Computing Machinery.

[RL11] Jonathan Rodriguez and Ondřej Lhoták. Actor-based parallel dataflow analysis. CC, 2011.

[RMR04] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Trans. Softw. Eng.*, 30(6):372–387, June 2004.

[Ros09] John R Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, pages 1–11, 2009.

[RR01] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Proceedings of the 10th International Conference on Compiler Construction*, volume 2027 of *CC '01*, pages 20–36. Springer Berlin Heidelberg, 2001.

[Ryd79] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[SAE+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66, 2018.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[SB10] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*, pages 245–251. Springer, 2010.

[SBEV18] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[SBKB15] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *Asian Symposium on Programming Languages and Systems*, pages 485–503. Springer, 2015.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, volume 46, pages 17–30. ACM, 2011.

[SBMH17] Justin Smith, Chris Brown, and Emerson Murphy-Hill. Flower: Navigating program flow in the ide. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 19–23. IEEE, 2017.

[SDE+18] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation. In *Asian Symposium on Programming Languages and Systems*, pages 69–88. Springer, 2018.

[SDTF20] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. On the recall of static call graph construction in practice. ICSE, 2020.

[SGSB05] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[Shi88] Olin Shivers. Control flow analysis in scheme. In *ACM SIGPLAN Notices*, volume 23, pages 164–174. ACM, 1988.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 264–280, New York, NY, USA, 2000. ACM.

[SHR01] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. Interprocedural control dependence. *ACM Trans. Softw. Eng. Methodol.*, 10(2):209–254, April 2001.

[Sie16] Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20. IEEE, 2016.

[Sma] Yannis Smaragdakis. DOOP Benchmarks. `https://bitbucket.org/yanniss/doop-benchmarks/`. [Online; accessed 23-August-2018].

[Sma18] Yannis Smaragdakis. DOOP - Framework for Java Pointer and Taint Analysis. `https://bitbucket.org/yanniss/doop/`, [Online; accessed 23-August-2018].

[SMS+12] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z Guyer. new scala () instance of java: a comparison of the memory behaviour of java and scala programs. In *ACM Sigplan Notices*, volume 47, pages 97–108. ACM, 2012.

[SMSB11] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. In *ACM SIGPLAN Notices*, volume 46, pages 657–676. ACM, 2011.

[SNAB16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. *DROPS-IDN/6116*, 56, 2016.

[SRH95] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 651–665, London, UK, UK, 1995. Springer-Verlag.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[TAD+10] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[TLR20] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, pages 81–93, 2020.

[TLX16] Tian Tan, Yue Li, and Jingling Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*, pages 489–510. Springer, 2016.

[TLX17] Tian Tan, Yue Li, and Jingling Xue. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. *ACM SIGPLAN Notices*, 52(6):278–291, 2017.

[TP00] Frank Tip and Jens Palsberg. *Scalable propagation-based call graph construction algorithms*, volume 35. ACM, 2000.

[VPP+20] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, 2020.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[VRH98] Raja Vallee-Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. 1998.

[Wha07] John Whaley. *Context-sensitive pointer analysis using binary decision diagrams*. PhD thesis, Stanford University, 2007.

[WL04] John Whaley and Monica S Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, 39(6):131–144, 2004.

[XR08] Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 225–236, New York, NY, USA, 2008. ACM.

[XRS09] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling cfl-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 98–122, Berlin, Heidelberg, 2009. Springer-Verlag.

[ZLTX18] Yifei Zhang, Yue Li, Tian Tan, and Jingling Xue. Ripple: Reflection analysis for android apps in incomplete information environments. *Software: Practice and Experience*, 48(8):1419–1437, 2018.

[ZTLX17] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. Ripple: Reflection analysis for android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 281–288, 2017.

[ZXZ+14] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for c. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 829–845, New York, NY, USA, 2014. ACM.

Appendix

A. Hermes: Example API Query

Listing A.1 shows an example of a Hermes query that computes the occurrences of API calls. The query shown in the listing captures whether the given input program performs calls to Java's *JDBC* API. For example, it counts the callsites that call *java.sql.Connection*'s *createStatement* method (cf. Line 34).

```
1  /* BSD 2-Clause License - see OPAL/LICENSE for details. */
2  package org.opalj.hermes
3  package queries
4
5  import org.opalj.br.ObjectType
6  import org.opalj.collection.immutable.Chain
7  import org.opalj.hermes.queries.util.APIFeature
8  import org.opalj.hermes.queries.util.APIFeatureGroup
9  import org.opalj.hermes.queries.util.APIFeatureQuery
10 import org.opalj.hermes.queries.util.InstanceAPIMethod
11 import org.opalj.hermes.queries.util.StaticAPIMethod
12
13 /**
14  * Counts the amount of calls to certain JDBC api methods
15  *
16  * @author Michael Reif
17  */
18 class JDBCAPIUsage(implicit hermes: HermesConfig) extends APIFeatureQuery {
19
20     override val apiFeatures: Chain[APIFeature] = {
21         val DriverManager = ObjectType("java/sql/DriverManager")
22         val Connection = ObjectType("java/sql/Connection")
23         val Statement = ObjectType("java/sql/Statement")
24         val PreparedStatement = ObjectType("java/sql/PreparedStatement")
25         val CallableStatement = ObjectType("java/sql/CallableStatement")
26
27         Chain(
28
29             StaticAPIMethod(DriverManager, "getConnection"),
30             InstanceAPIMethod(Connection, "rollback"),
31
32             APIFeatureGroup(
33                 Chain(
34                     InstanceAPIMethod(Connection, "createStatement"),
35                     InstanceAPIMethod(Statement, "execute"),
36                     InstanceAPIMethod(Statement, "executeQuery"),
37                     InstanceAPIMethod(Statement, "executeUpdate")
```

```
38                  ),
39                  "creation and execution of\njava.sql.Statement"
40              ),
41
42              APIFeatureGroup(
43                  Chain(
44                      InstanceAPIMethod(Connection, "prepareStatement"),
45                      InstanceAPIMethod(PreparedStatement, "execute"),
46                      InstanceAPIMethod(PreparedStatement, "executeQuery"),
47                      InstanceAPIMethod(PreparedStatement, "executeUpdate")
48                  ),
49                  "creation and execution of\njava.sql.PreparedStatement"
50              ),
51
52              APIFeatureGroup(
53                  Chain(
54                      InstanceAPIMethod(Connection, "prepareCall"),
55                      InstanceAPIMethod(CallableStatement, "execute"),
56                      InstanceAPIMethod(CallableStatement, "executeQuery"),
57                      InstanceAPIMethod(CallableStatement, "executeUpdate")
58                  ),
59                  "creation and execution of\njava.sql.CallableStatement"
60              )
61          )
62      }
63  }
```

Listing A.1: The code of the *JDBCAPIUsage* HERMES query that checks a given program for the occurence of call to the Java's JDBC API.

B. Hermes: Example Metric Query

Listing B.1 shows the code of *Metrics* Hermes query. The query computes several metrics, namely fields per class (FPC), methods per class (MPC), classes per package (CPP), number of children (NOC), and McCabe. Furthermore, the query computes for every feature—a metric is considered a feature—four feature extensions.

Whereas the first three metrics are self-explanatory, we will next shortly introduce McCabe and NOC. McCabe computes the cyclic complexity of a method's control-flow. The lower the metric, the less complicated the method is, e.g., a method without any branches has linear complexity.

NOC computes the number of direct descendants (subclasses) for each class. Classes with a large number of children are considered to be challenging to modify and usually require more testing because of the effects on changes on all the children. They are also considered more complex and fault-prone because a class with numerous children may have to provide services in a larger number of contexts and, therefore, must be more flexible [BBM96].

```
1  /* BSD 2-Clause License - see OPAL/LICENSE for details. */
2  package org.opalj
3  package hermes
4  package queries
5
6  import scala.collection.mutable
7  import org.opalj.br.analyses.Project
8  import org.opalj.br.cfg.CFGFactory
9
10 /**
11  * Extracts basic metric information (Fields/Methods per Class; Classes per Package; etc.).
12  *
13  * @author Michael Reif
14  */
15 class Metrics(implicit hermes: HermesConfig) extends FeatureQuery {
16
17   /**
18   * The unique ids of the extracted features.
19   */
20   override val featureIDs: Seq[String] = {
21     Seq(
22       "0 FPC", "1-3 FPC", "4-10 FPC", ">10 FPC", // 0, 1, 2, 3
23       "0 MPC", "1-3 MPC", "4-10 MPC", ">10 MPC", // 4, 5, 6, 7
24       "1-3 CPP", "4-10 CPP", ">10 CPP", // 8, 9, 10
25       "0 NOC", "1-3 NOC", "4-10 NOC", ">10 NOC", // 11, 12, 13, 14
```

```
26          "linear methods (McCabe)", "2–3 McCabe", "4–10 McCabe", ">10 McCabe" //
                15, 16, 17 ,18
27        )
28      }
29
30      override def apply[S](
31        projectConfiguration: ProjectConfiguration,
32        project: Project[S],
33        rawClassFiles: Traversable[(da.ClassFile, S)]
34      ): TraversableOnce[Feature[S]] = {
35
36        val classLocations = Array.fill(featureIDs.size)(new LocationsContainer[S])
37
38        class PackageInfo(var classesCount: Int = 0, val location: PackageLocation[S])
39        val packagesInfo = mutable.Map.empty[String, PackageInfo]
40
41        val classHierarchy = project.classHierarchy
42
43        for {
44          (classFile, source) <- project.projectClassFilesWithSources
45          classLocation = ClassFileLocation(source, classFile)
46        } {
47        // FPC: fields per class
48
49        classFile.fields.size match {
50        case 0 => classLocations(0) += classLocation
51        case x if x <= 3 => classLocations(1) += classLocation
52        case x if x <= 10 => classLocations(2) += classLocation
53        case x => classLocations(3) += classLocation
54        }
55
56        // MPC: methods per class
57
58        classFile.methods.size match {
59          case 0 => classLocations(4) += classLocation
60          case x if x <= 3 => classLocations(5) += classLocation
61          case x if x <= 10 => classLocations(6) += classLocation
62          case x => classLocations(7) += classLocation
63        }
64
65        // noc
66
67        classHierarchy.directSubtypesOf(classFile.thisType).size match {
68          case 0 => classLocations(11) += classLocation
69          case x if x <= 3 => classLocations(12) += classLocation
70          case x if x <= 10 => classLocations(13) += classLocation
71          case x => classLocations(14) += classLocation
72        }
73
```

```
74      // count the classes per package
75      val packageName = classFile.thisType.packageName
76      val packageInfo = packagesInfo.getOrElseUpdate(
77        packageName,
78        new PackageInfo(location = PackageLocation(packageName))
79      )
80      packageInfo.classesCount += 1
81
82      // McCabe
83      classFile.methods foreach { method =>
84        CFGFactory(method, project.classHierarchy) foreach { cfg =>
85            val methodLocation = MethodLocation(classLocation, method)
86            val bbs = cfg.reachableBBs
87            val edges = bbs.foldLeft(0) { (res, node) =>
88              res + node.successors.size
89            }
90            val mcCabe = edges - bbs.size + 2
91            mcCabe match {
92              case 1 => classLocations(15) += methodLocation
93              case x if x <= 3 => classLocations(16) += methodLocation
94              case x if x <= 10 => classLocations(17) += methodLocation
95              case x => classLocations(18) += methodLocation
96            }
97          }
98        }
99      }
100
101     packagesInfo.values foreach { pi =>
102       pi.classesCount match {
103         case x if x <= 3 => classLocations(8) += pi.location
104         case x if x <= 10 => classLocations(9) += pi.location
105         case x => classLocations(10) += pi.location
106       }
107     }
108
109     for { (featureID, featureIDIndex) <- featureIDs.iterator.zipWithIndex } yield {
110       Feature[S](featureID, classLocations(featureIDIndex))
111     }
112   }
113 }
```

Listing B.1: The code of the Metrics Hermes query that computes several metrics.

C. Hermes: Example Custom Query

Listing C.1 shows an example of a Hermes query that computes the number of recursive data structures, which can significantly limit the scalability of analyses [LSBM15].

```
1  /* BSD 2-Clause License - see OPAL/LICENSE for details. */
2  package org.opalj
3  package hermes
4  package queries
5
6  import org.opalj.graphs.UnidirectionalGraph
7  import org.opalj.br.ObjectType
8  import org.opalj.br.analyses.Project
9
10 /**
11  * Identifies recursive data structures. Such data-structure can often significantly limit
12  * the scalability of analyses.
13  *
14  * @author Michael Eichberg
15  */
16 class RecursiveDataStructures(implicit hermes: HermesConfig) extends FeatureQuery {
17
18     override def featureIDs: IndexedSeq[String] = {
19         IndexedSeq(
20             /*0*/ "Self-recursive Data Structure",
21             /*1*/ "Mutually-recursive Data Structure\n2 Types",
22             /*2*/ "Mutually-recursive Data Structure\n3 Types",
23             /*3*/ "Mutually-recursive Data Structure\n4 Types",
24             /*4*/ "Mutually-recursive Data Structure\nmore than 4 Types"
25         )
26     }
27
28     override def apply[S](
29         projectConfiguration: ProjectConfiguration,
30         project: Project[S],
31         rawClassFiles: Traversable[(da.ClassFile, S)]
32     ): TraversableOnce[Feature[S]] = {
33
34         import project.classHierarchy.getObjectType
35
36         val g = new UnidirectionalGraph(ObjectType.objectTypesCount)()
37
38         val locations = Array.fill(featureIDs.size)(new LocationsContainer[S])
39
```

```
40          // 1. create graph
41          for {
42              classFile <— project.allProjectClassFiles
43              if !isInterrupted()
44              classType = classFile.thisType
45              field <— classFile.fields
46              fieldType = field.fieldType
47          } {
48              if (fieldType.isObjectType) {
49                  g += (classType.id, fieldType.asObjectType.id)
50              } else if (fieldType.isArrayType) {
51                  val elementType = fieldType.asArrayType.elementType
52                  if (elementType.isObjectType) {
53                      g += (classType.id, elementType.asObjectType.id)
54                  }
55              }
56          }
57
58          // 2. search for strongly connected components
59          for {
60              scc <— g.sccs(filterSingletons = true)
61              if !isInterrupted()
62              /* An scc is never empty! */
63              sccCategory = Math.min(scc.size, 5) — 1
64              objectTypeID <— scc
65              objectType = getObjectType(objectTypeID)
66          } {
67              locations(sccCategory) += ClassFileLocation(project, objectType)
68          }
69
70          for { (locations, index) <— locations.iterator.zipWithIndex } yield {
71              Feature[S](featureIDs(index), locations)
72          }
73      }
74  }
```

Listing C.1: The code of the *JDBCAPIUsage* HERMES query that checks a given program for the occurence recursive data structures.