Understanding Variability in Space and Time

Analyzing Features and Revisions in Concert

A dissertation submitted towards the degree

Doctor of Engineering (Dr.-Ing.)

of the Faculty of Mathematics and Computer Science of Saarland University

by

Florian Sattler

Saarbrücken 2023

UNIVERSITÄT
DES
SAARLANDES

Dean of Faculty	Prof. Dr. Roland Speicher
Day of Colloquium	15.04.2024
Chair of the Committee	Prof. Dr. Jan Reineke
Reviewers	Prof. Dr. Sven Apel
	Prof. Dr. Sebastian Hack
Academic Assistant	Dr. Norman Peitek

Conquer yourself rather than the world.

— René Descartes

Abstract

The static or dynamic analysis of configurable software systems imposes significant challenges regarding complexity and computation time due to the software systems' large configuration spaces. These are aggravated by the evolution of software systems: developers frequently produce new revisions, adapting and modifying the system. Thereby, analysis results can quickly become out of date or are difficult to interpret. For example, debugging configuration-specific performance bugs is difficult by itself, but it becomes even more challenging in an evolving system. A helpful analysis tool should provide not only accurate configuration-specific performance measurements but also put them into the development context of the project. Through the development context, results are attributed to a more specific scope or developer and, by that, can be easier interpreted. The key problem is that current analyses, even when specialized for configurable software systems, cannot contextualize their findings within the development context of the software project in question. To address this problem, we need to empower program analyses to incorporate variability information. That is, we enable them to integrate information about the configurability of the system as well as the evolutionary context.

To achieve a better integration, we propose a unified abstraction of code regions that provides variability information to existing program analyses. That is, we map information about configuration variability as well as evolutionary variability onto a low-level intermediate program representation on which existing program analyses operate and provide variability information through a uniform interface. This has two main advantages: (1) by decoupling program analysis semantics from variability information, individual program analyses as well as the different types of variability information become reusable; (2) by mapping variability information onto the program representation used by existing program analyses, we make the analyses and their results extendable with variability information without requiring changes to the analysis. This way, we enable existing program analyses to relate and interpret their results in the context of variability. Through a product-line-based design, we make both variability information and analyses reusable and freely combinable, helping researchers to incorporate variability information into their work (e.g., to contextualize their analysis results with evolutionary information).

In this thesis, we demonstrate the applicability of a uniform abstraction of code regions by addressing two novel research problems: First, we combine evolutionary information, mined from software repositories, with an inter-procedural data-flow analysis to determine how evolutionary changes interact within a software project,

revealing new and interesting connections between changes and developers. Furthermore, our work shows that existing program analysis results can be contextualized in the socio-technical context of a software project through the provided evolutionary information. Second, we combine different automated localization approaches that detect configuration-specific code with state-of-practice performance profilers to enable configuration-aware performance profiling. Our results show that this enables performance profilers to attribute performance regressions directly to configuration options without introducing unnecessary overhead.

In summary, this thesis bridges the gap between variability information and precise program analysis. Through the unified abstraction of code regions, we enable the free combination of variability information with state-of-the-art program analyses, laying the foundation for more variability-focused program analyses and, by that, a better understanding of the evolution of configurable software systems.

Zusammenfassung

Die statische oder dynamische Analyse von konfigurierbaren Softwaresystemen stellt durch die großen Konfigurationsräume von Softwaresystemen Herausforderungen im Bezug auf Komplexität und Rechenzeit dar. Durch die Evolution von Softwaresystemen wird dies noch verschärft: Die Entwickler erstellen ständig neue Revisionen, wodurch das System anpassen und modifizieren wird. Aufgrund dessen können Analyseergebnisse schnell veralten oder schwer zu interpretieren sein. Zum Beispiel ist das Debuggen von konfigurationsbezogenen Performanceregressionen an sich schon schwierig, wird aber in einem sich entwickelnden System noch schwieriger. Daher sollte ein hilfreiches Analysewerkzeug nicht nur genaue konfigurationsbezogene Performanceregressionen liefern, sondern diese auch in den Entwicklungskontext des Projekts einordnen. Durch den Entwicklungskontext werden die Ergebnisse einem spezifischeren Bereich oder einer Person zugeordnet und lassen sich so leichter interpretieren. Das Hauptproblem hierbei ist, dass gegenwärtige Analysen, selbst wenn sie auf konfigurierbare Softwaresysteme spezialisiert sind, ihre Ergebnisse nicht in den Entwicklungskontext des Softwareprojekts einordnen können. Um dieses Problem zu lösen, müssen wir Programmanalysen die Integration von Variabilitätsinformationen—das heißt Informationen über die Konfigurierbarkeit des Systems und den evolutionären Kontext—erleichtern.

Um die Integration zu erleichtern, schlagen wir eine einheitliche Abstraktion durch Coderegionen vor, welche die Variabilitätsinformationen für bestehende Programmanalysen bereitstellt. Das heißt, wir bilden Informationen über Konfigurationsvariabilität sowie über evolutionäre Variabilität auf eine Low-Level-Zwischenprogrammrepräsentation ab, auf der bestehende Programmanalysen bereits

arbeiten, und stellen Variabilitätsinformationen über eine einheitliche Schnittstelle bereit. Dadurch haben wir zwei Vorteile: (1) Durch die Entkopplung der Analysesemantik von der Variabilitätsinformation werden sowohl die Analysen als auch die verschiedenen Arten von Variabilitätsinformationen wiederverwendbar; (2) durch die Abbildung von Variabilitätsinformationen auf die Programmrepräsentation, die von bestehenden Programmanalysen verwendet wird, machen wir die Analysen und ihre Ergebnisse mit Variabilitätsinformationen erweiterbar, ohne dabei die Analyse verändern zu müssen. Das heißt, wir ermöglichen es Analyseergebnisse auch im Kontext von Variabilität mit dieser in Beziehung zu setzen und zu interpretieren. Durch ein produktlinienbasiertes Design sorgt unsere Abstraktion dafür, dass sowohl Variabilitätsinformationen als auch Analysen wiederverwendbar und frei kombinierbar werden. Auf diese Weise ermöglichen wir es Forschern, Variabilitätsinformationen in ihre Arbeit einzubeziehen (z. B. um ihre Analyseergebnisse mit evolutionären Informationen zu kontextualisieren).

In dieser Arbeit demonstrieren wir die Anwendbarkeit unserer einheitlichen Coderegionen-Abstraktion, an zwei bisher ungelöste Forschungsproblemen. Erstens verwenden wir evolutionäre Informationen aus Software-Repositories zusammen mit einer inter-prozeduralen Datenflussanalyse, um festzustellen, wie evolutionäre Änderungen innerhalb eines Softwareprojekts interagieren, wodurch neue und interessante Verbindungen zwischen Änderungen und Entwicklern aufgedeckt werden. Unsere Arbeit zeigt außerdem, dass bestehende Analyseergebnisse durch das Bereitgestellen von evolutionären Informationen mithilfe des sozio-technischen Kontexts eines Softwareprojekts kontextualisiert werden können. Zweitens kombinieren wir verschiedene Lokalisierungsansätze, die konfigurationsspezifischen Code erkennen, mit aktuellen Performance-Profilern, um ein konfigurationsbezogenes Performance-Profiling zu ermöglichen. Unsere Ergebnisse zeigen, dass dies Performance-Profiler in die Lage versetzt, Performanceregressionen direkt den Konfigurationsoptionen zuzuordnen, ohne unnötigen Mehraufwand zu verursachen.

Zusammenfassend lässt sich sagen, dass diese Arbeit die Lücke zwischen Variabilitätsinformationen und präziser Programmanalyse schließt. Durch die einheitliche Abstraktion mithilfe von Coderegionen ermöglichen wir die freie Kombination von Variabilitätsinformationen mit modernsten Programmanalysen und legen so den Grundstein für variabilitätsorientierte Programmanalysen und damit auch für ein besseres Verständnis der Entwicklung konfigurierbarer Softwaresysteme.

Publications

The following publications where published during the development of this dissertation and influence the overall work. Hence, some ideas, wordings, and figures have appeared previously in the following publications:

Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Trans. Softw. Eng. Methodol.* (2023).

Florian Sattler, Alexander von Rhein, Thorsten Berger, Niklas Schalck Johansson, Mikael Mark Hardø, and Sven Apel. "Lifting Inter-App Data-Flow Analysis to Large App Sets." In: *Autom. Softw. Eng.* 25.2 (2018), pp. 315–346.

Missing a train is only painful if you run after it!
Likewise, not matching the idea of success others expect from you
is only painful if that's what you are seeking.

— Nassim Nicholas Taleb [225]

Acknowledgments

Writing a dissertation is a long journey one undertakes to further scientific understanding, learn more about the world, and especially about oneself. Luckily, I did not have to make this journey on my own. In what follows, I want to relay a heart-warmed *thank you* to all who helped and walked alongside me in this journey over the last six years.

First and foremost, I want to thank my supervisor Sven Apel. As he guided me through my journey, I don't know if he ever realized that the person I admired most was himself. Sven is my role model of a scientist. Never, have I met someone so dedicated, passionate, and enthusiastic about research and scientific work. The way in which he radiates his love for scientific inquiry and knowledge, far beyond the scope of computer science, is contagious and creates a positive environment in which research and researchers can thrive. Our journey together was incredible and changed me as well as my view of the world for the better. Words cannot describe what impact you had on my life. Thank you, Sven, for everything. I'll be forever grateful.

Second, I would like to thank Norbert Siegmund. Our discussions often clarified important statistical and practical aspects of my work, and helped me to put my work into context. Norbert, I learned a lot from you about scientific thinking, visualizing data, and constructing arguments. Thank you, for all your help.

In general, I would like to thank all my co-authors for many interesting discussions, feedback, and so much more that I could learn from them: Christian Kästner, Sebastian Böhm, Christian Kaltenecker, Max Weber, Stefan Mühlbauer, Philipp Schubert, Fabian Schiebel, Ben Hermann, Eric Bodden, Christian Engwer, Miguel Velez, Alexander von Rhein, Thorsten Berger. I would like also to thank my second examiner Sebastian Hack who agreed to review my thesis, which is an arduous task that is often not valued enough.

In addition, a big thanks to all PHASAR developers, most notably Philipp Schubert, Martin Mory, and Fabian Schiebel who created, through their hard work, a practical static analysis framework that enabled me to conceptualize new and interesting insights.

Furthermore, I would like to thank all my colleagues, with whom I had the pleasure of working with, for supporting me on my way to this thesis: Christian Kaltenecker, Sebastian Böhm, Thomas Bock, Christian Hechtl, Annabelle Bergum, Lukas Abelt, Kallistos Weis, Georg Seibt and Friederike Repplinger. In addition, I would like to thank also my external proofreaders: Andreas Simbürger, Jörg Liebig, and JC van Winkel.

In the end, on a more personal note, I would like to thank my mother and grandparents who always believed in me, never gave up on me, and supported me emotionally throughout this work. I would like to thank all my friends that stood by my side since my early youth and those that I found during my studies, for their support and the good times we had together. Finally, a very special and heartfelt *thank you*, to my future wife Miriam. The path to a dissertation leads through personal hardships and sacrifices. You were the best partner I could wish for on such a journey. I love you.

Contents

1 Introduction 1
1.1 Contributions . . . 4
1.2 Outline . . . 6
2 Background 9
2.1 Configurable Software Systems . . . 9
2.1.1 Configuration Options . . . 10
2.1.2 Configurations . . . 12
2.1.3 Binding Times . . . 13
2.1.4 Implementation Techniques . . . 14
2.1.5 Feature Interactions . . . 18
2.2 Software Evolution . . . 20
2.2.1 Version Control Systems . . . 21
2.2.2 Repository Mining . . . 23
2.2.3 Socio-Technical Context . . . 24
2.2.4 Evolution of Configurable Software Systems . . . 25
2.3 Compiler-Aided Program Analysis . . . 27
2.3.1 Data-Flow Analysis . . . 28
2.3.2 Inter-Procedural Analysis . . . 29
2.3.3 Inter-Procedural Distributive Environments . . . 32
2.3.4 Compiler Infrastructure . . . 33
2.3.5 Whole-Program Analysis . . . 35
2.3.6 Phasar . . . 36
2.3.7 Performance Analysis . . . 36
3 A Uniform Code-Region Abstraction 45
3.1 Introduction . . . 45
3.2 Code-Region Abstraction . . . 48
3.2.1 Program Representation . . . 50
3.2.2 Variability Tagging . . . 51
3.2.3 Aggregation Into Regions . . . 52
3.2.4 Program History . . . 55
3.2.5 Code Regions as a Uniform Analysis Interface . . . 56
3.3 Lifting Program Analysis to Code Regions . . . 58
3.3.1 Static Analysis . . . 60
3.3.2 Dynamic Analysis . . . 62
3.4 Code Regions in LLVM . . . 64
3.4.1 Overview . . . 64

3.4.2 Encoding Variability Information into LLVM-IR . . . 65
3.4.3 A Domain-Specific View on LLVM-IR . . . 66
3.4.4 Detecting Code Regions . . . 67
3.4.5 Static and Dynamic Analysis . . . 69
3.5 Code-Region Analysis Applied to Variability . . . 71
3.6 Related Work . . . 72
3.7 Further Applications . . . 75
3.8 Summary . . . 77
4 Integrating Program Analysis and Repository Mining 79
4.1 Introduction . . . 79
4.2 SEAL at a Glance . . . 82
4.2.1 Code Annotation . . . 82
4.2.2 Commit Interactions . . . 83
4.2.3 Computing Commit Interactions . . . 85
4.3 Implementation . . . 90
4.3.1 Lowering Commit Information to LLVM-IR . . . 90
4.3.2 Creating a Whole-Program Bitcode File . . . 91
4.3.3 Taint Analysis for Commit Interactions . . . 91
4.3.4 Data Overview . . . 92
4.4 Commit Interactions: Applications . . . 93
4.4.1 Commit Interaction Graph Analysis . . . 93
4.4.2 Socio-Technical Data-Flow Analysis . . . 98
4.5 SEAL in Action . . . 99
4.5.1 Commit Interaction Graph Analysis . . . 100
4.5.2 Socio-Technical Data-Flow Analysis . . . 105
4.6 Threats to Validity . . . 106
4.7 Discussion . . . 107
4.7.1 Data-Flow-Based Commit Interactions . . . 107
4.7.2 Socio-Technical Data-Flow Analysis . . . 111
4.7.3 Limitations . . . 113
4.8 Related Work . . . 115
4.9 Summary . . . 118
5 White-Box Performance Analysis of Configurable Software Systems 121
5.1 Introduction . . . 122
5.2 Walrus at a Glance . . . 124
5.2.1 Concept of Configuration-Dependent Regions . . . 126
5.2.2 Load-Time Configuration-Dependent Regions . . . 128
5.2.3 Compile-Time Configuration-Dependent Regions . . . 130
5.3 Making Profilers Configuration-Aware . . . 131
5.3.1 Profiler Interfaces . . . 132
5.3.2 Application Scenarios . . . 134
5.4 Evaluation . . . 136
5.4.1 Research Questions . . . 137
5.4.2 Experiment Setup . . . 137

5.4.3 Synthetic Subject Systems . . . 140
5.4.4 Applicability (RQ_1) . . . 142
5.4.5 Impact of Configuration Awareness (RQ_2) . . . 145
5.5 Threats to Validity . . . 147
5.6 Discussion . . . 148
5.6.1 Configuration-Aware Performance Analysis . . . 148
5.6.2 Limitations . . . 148
5.7 Related Work . . . 149
5.8 Summary . . . 150
6 Concluding Remarks 153
6.1 Summary of Contributions . . . 153
6.2 Future Work . . . 155
6.2.1 Extending Code Regions to Other Domains . . . 155
6.2.2 Enhancing Code-Region Interactions Through Data-Flow Path Information . . . 156
6.2.3 Analyzing Code Regions in a Historical Context . . . 157
6.2.4 Applying Commit-Interaction Data to Socio-Technical Network Analysis . . . 158

Bibliography 159

List of Figures

Figure 1.1 Contribution overview 5
Figure 2.1 Partial configuration model for XZ 11
Figure 2.2 Preprocessor example from XZ 15
Figure 2.3 Template meta-programming example from DUNE 16
Figure 2.4 Configuration variable example from LRZIP 17
Figure 2.5 Overview of important GIT-specific terminology. 22
Figure 3.1 Overview of Chapter 3 46
Figure 3.2 Domain-driven design for the uniform code-region abstraction 50
Figure 3.3 Detecting code regions within a instGraph 54
Figure 3.4 Compiler-based variability analysis pipeline 65
Figure 4.1 SEAL: motivational example 80
Figure 4.2 Running example: commit information at source-code level and its mapping to the IR. 83
Figure 4.3 Distributive flow functions and their bipartite graph representations. 86
Figure 4.4 Excerpt of the exploded super-graph for the taint analysis $\mathbb{T}$. 87
Figure 4.5 *Normal* flow functions of the taint analysis $\mathbb{T}$. 88
Figure 4.6 *Normal* edge functions of the taint analysis $\mathbb{T}$. 88
Figure 4.7 Overview of SEAL's full build pipeline. 90
Figure 4.8 The function `align` in file `src/opus_private.h` from OPUS revision `348e694`. 94
Figure 4.9 Central code interaction examples. 95
Figure 4.10 Shortened version of function `ssh_handle_packets` from LIBSSH. 97
Figure 4.11 Example how code form a commit interacts with different authors. 97
Figure 4.12 Example of a program that is vulnerable to SQL injections. . . 99
Figure 4.13 Comparison between node degree and commit size. 101
Figure 4.14 Commit `348e694` from opus changes only a single line in the function `align`. 101
Figure 4.15 Number of commits vs. number of interacting authors for authors, including kernel density estimations. 103
Figure 4.16 Comparison between CI-based and file-based author interactions. 104
Figure 4.17 Comparison between CI-based and call-graph-based author interactions. 105

Figure 4.18 Number of distinct authors each commit interacts with, normalized by the number of authors per project. Violins show the associated probability densities. 106
Figure 4.19 Conceptual example of a software project with multiple authors and files. 108
Figure 4.20 Coordination requirements between authors as determined by different artifact coupling approaches. 110
Figure 4.21 A program that is vulnerable to SQL injections. 111
Figure 4.22 Commit `5341f7b` in which Leonie wants to prevent SQL injections by adding a call to `sanitize`. 112
Figure 4.23 A socio-technical bug report about a found SQL-injection. . . 112
Figure 4.24 Correlation between the number of commits and blame computation time. 115
Figure 5.1 WALRUS: motivational example 125
Figure 5.2 Configuration-aware profiling setup with WALRUS 126
Figure 5.3 Control-flow graph from the motivational example. 127
Figure 5.4 SDT probe with configuration-specific information 134
Figure 5.5 Interleaved stack trace with XRAY and WALRUS measurements. 136
Figure 5.6 Overview of our evaluation setup. 138
Figure 5.7 Comparison of precision and recall for different configuration-specific performance profiling approaches. 145

List of Tables

Table 4.1 Subject Projects. Metrics for BISON, GREP, and GZIP include submodules. 100
Table 4.2 Blame-annotation overhead data 114
Table 5.1 List of selected subject system for evaluating WALRUS. 140
Table 5.2 Evaluation results for configuration-aware profilers. 144

Acronyms

AST	Abstract Syntax Tree
LTO	Link-Time Optimization
HPC	High-Performance Computing
IDE	Inter-Procedural Distributive Environments
CFG	Control-Flow Graph

1 Introduction

Over the last decades, the complexity of modern software systems has been growing rapidly. On the one hand, software projects become increasingly more configurable to better adapt to different use cases and user needs. On the other hand, due to the growing number of developers and project sizes, understanding the evolution and socio-technical context of a project becomes not only more difficult but also more important, as with a growing number of developers, handling of social interactions in a project can become more complicated and slow down development [108, 128, 157]. Both sides, configuration variability (*space*) and evolutionary variability (*time*) pose substantial challenges when conducting static and dynamic program analyses on configurable software systems over time.

Configuration variability arises from deliberate design and implementation choices of developers, where parts of the system are variable in that they can be enabled, disabled, or configured. Developers have a diverse arsenal of techniques at their disposal to realize this configuration variability, such as preprocessor directives (`#ifdef`), template meta-programming, command-line parameters, or configuration files. In general, configuration options make software systems flexible in that users can adapt and optimize the configurable software system for their use case, for example, by disabling unwanted functionality or tuning the system to specific workloads. However, with ever-increasing numbers of configuration options [252], users and developers get easily confused. Xu et al. [252] report that up to 48.5% of configuration issues arise from users difficulties in finding the right configuration options or setting the right values, advocating for leaner configurable systems (e.g., they show that an exemplary project could remove 51.9% of its configuration options with only a small impact on existing users). The core problem with a rising number of configuration options is the combinatorial explosion of the configuration space [7, 228]. Adding one Boolean option to a configuration space already doubles the number of possible configurations (i.e., the size of the configuration space). This results in costs as code becomes harder to adapt and extend, makes maintenance more expensive and difficult, and puts an additional burden on program analysis tools (as they need to incorporate the variability into their analysis).

Over the last decade, many research projects worked on overcoming these challenges. Most important with regard to software analyses was the conceptual combination of standard program analyses with variability information [232]. Brabrand et al. [28] proposed a way to automatically turn standard data-flow analyses into

feature-sensitive (i.e., configuration-sensitive) ones that are able to analyze all valid configurations at once. With SPL$^{\text{LIFT}}$, Bodden et al. [25] showed a conceptual way to lift a generic IFDS analysis to automatically analyze software product lines by transforming the analysis into an IDE analysis that additionally tracks variability information on the edge domain of the analysis [145, 194]. The key idea behind these approaches—following early ideas of Czarnecki and Pietroszek [45]—is to tackle the combinatorial explosion arising from the configuration space by making the analysis itself variability-aware. *Variability-aware* analyses [232] and *variational* data structures [246] exploit the commonalities between different configurations to reduce analysis time, enriching the specific analysis problem with information about the configuration space to analyze the whole configuration space at once. For example, Kästner et al. [118] developed a variability-aware type checker for `#ifdef`-based C product lines that is able to type-check all variants simultaneously [117, 230]. Rhein et al. [194] demonstrated not only that variability-aware analyses scale to real-world software projects but, more importantly, that they outperform existing non-variability-aware analyses by orders of magnitude. Furthermore, Apel et al. [11] demonstrated that their variability-aware model checker outperformed various sampling-based approaches in terms of detection efficiency. Overall, in previous research, many approaches have tackled configuration variability by making standard software analyses variability-aware [232].

However, variability in software projects does not only arise from the configuration space but also from their development over time (i.e., from the evolution of the project). Software projects get developed in code chunks, where every new chunk adds, removes, or changes code from the project. These code changes, some small others large, adapt the behavior of the software project over time (e.g., by adding new features, fixing a bug, or refactoring). Understanding how these changes are produced, interact, and together give rise to a functioning software project is important to improve software development overall [155], independent of the specific goal (e.g., to analyze the impact of a change [141] or finding out how developers interact [107]). However, to analyze the evolution of a software project, one requires data about the changes themselves, the developers that produced them, and the context of the project. An important technique to gather these project data is *repository mining* [46], where one extracts, models, and analyzes data available in the project's resources [48], such as the version control system [107, 255], mailing lists [23, 161, 177, 211], issue tracking systems [175, 178], or pull requests [236]. For example, the version history of a software project contains detailed information on code changes, including the ordering in which they appeared and who wrote the code. Important to note, the mined data gives us not only access to the code changes but also enables analyses of the socio-technical context of a software project. Understanding this socio-technical context is highly important to understand how projects evolve [23, 24, 31, 93, 94, 106], as primarily the developers working on the project shape its evolution. For example, Joblin et al. [106] use developer networks, mined from software repositories and corresponding code artifacts, to extract the hierarchies and organizational structures of open-source communities.

Obtaining a deep understanding of how software projects evolve and how the people involved interact is vital to steer and improve software development overall, including configurable software systems. However, in configurable software systems, the variability introduced through evolution needs to be understood alongside the variability introduced by configurability [34, 182, 183, 233]. So, for configurable systems, software evolution comes with the additional burden that a change in the *time* dimension (i.e., a change to the source code) can have influences across the whole *space* dimension (i.e., potentially affecting multiple configurations) and on the structure of the space dimension itself (i.e., adapting in what ways the system can be configured). That is, a change can have a potential impact on all, a few, or only one configuration(s). The problem lies in the fact that developers do not know which configurations are affected, or change the structure of the configuration space (e.g., adding a new feature or allowing two features to be enabled in parallel), which invalidates previous assumptions about the configuration space. Conceptually, the evolutionary steps (i.e., code changes) form an additional time dimension to the already complex configuration space of a project.

The discussion so far illustrates that analyzing the evolution and socio-technical context of configurable software systems is difficult but important. For example, debugging configuration-specific performance bugs is difficult by itself [252], but it becomes even more challenging in an evolving system where the changes and goals from another team influence how the bug can be addressed. For this task, a helpful analysis should provide not only accurate configuration-specific performance measurements but also put them into the development context of the project (i.e., relating them to a change set or the involved developers), so that the developer working on the performance bug knows directly whom to consult with and can fix the bug quicker. To achieve that, a program analysis needs to take variability information from both the configuration space and evolutionary context into account and correctly attribute it to the analysis results.

A key problem that we identified in this dissertation project is that, even when retrofitted for configurable software systems, current program analyses cannot contextualize their inner working and findings within the development context of the software project (i.e., relating their results to specific changes and developers). This stems from the fact that additional information about configuration variability and evolutionary variability is not easily accessible by state-of-the-art program analyses. To address this problem, we need an *automated and unified way for program analyses to access variability information about configurable software systems together with information about their development context*. Through the provided variability information, we enable program analyses to incorporate configuration knowledge into their findings and contextualize these findings in the socio-technical context of the software project.

1.1 Contributions

The goal of this thesis is to empower program analysis to incorporate evolutionary as well as configuration-specific information and to contextualize their existing analysis results with this additional information. To achieve this goal, we propose an *abstraction of code regions that uniformly represent different types of variability and make them accessible to the program analysis*. The two central ideas behind the uniform code-region abstraction are: (1) decoupling and automating the extraction of different types of variability information, to make both analyses and variability information freely combinable and reusable; (2) presenting the gathered variability information to state-of-the-art program analyses, so that an analysis can directly use the information and interpret its results in the corresponding context (e.g., evolutionary information from the repository).

For this purpose, we built a reusable extraction procedure that gathers variability information from feature models and software repositories. We then map the gathered information onto a low-level intermediate program representation, on which the analysis operates. That is, we map evolutionary data, mined from software repositories, as well as configuration-specific information, extracted from feature models and code, directly onto the program representation, making them accessible for a wide range of program analyses. The uniform code-region abstraction acts as the glue between variability information and program analyses, by providing the mapping between variability and intermediate program representation. Important to note, this relation is the key to reusability and adaptability. Since existing analyses operate on the intermediate program representation, they can now directly access variability information and utilize it throughout the analysis. Furthermore, even without modification of the analysis, existing analysis results can be interpreted in the context of variability information (e.g., by mapping the detected SQL injection through the abstraction to the developers involved).

The uniform abstraction of code regions enables static and dynamic program analyses either to integrate evolutionary and configuration-specific information into their inner workings directly or to contextualize analysis results with it. We demonstrate the usefulness and applicability of our uniform code-region abstraction by applying it to two unsolved research problems: First, we developed SEAL, the first integrated approach that combines low-level program analysis with high-level repository information. For this purpose, we mine repository information (time dimension) from the development history of a given project and present it through our uniform code-region abstraction to state-of-the-art program analysis. Using this information, the analysis is able to infer how developers interact. We evaluated SEAL on a diverse set of 13 real-world open-source projects and found that data-flow analysis can reveal new links between developers that could not be discovered with existing approaches. In this context, we demonstrated how repository information can be used to put already existing program analysis results into a socio-technical context. Second, we developed Walrus, the first analysis framework

that integrates compile-time and load-time configurability detection approaches with state-of-practice performance profilers. For this purpose, we implemented different approaches to locate configuration-specific code (space dimension), which is provided via our uniform code-region abstraction to a dynamic-analysis interface inside Walrus. Based on the detected code regions, Walrus is able to weave in profiler-specific measurement code around code regions to contextualize the performance profiler's measurements with configuration knowledge. This way, we were able to plug together configuration-specific performance profilers using state-of-practice performance profilers without a significant amount of overhead. We evaluated Walrus on 18 subject systems, demonstrating that configuration-sensitive state-of-practice profilers are able to detect configuration-specific performance regressions.

Through solving the two research problems of putting analysis results into an evolutionary context and making state-of-practice profilers configuration sensitive, we demonstrate the applicability of our code-region abstraction to different types of variability. Both static and dynamic analyses can profit from having a uniform interface to query additional information about variability. Figure 1.1 depicts an overview of our contributions. Through an uniform code-region abstraction, we bridged the gap between high-level variability information and low-level program analyses, making it easier for analyses to integrate variability information. Overall, we see our uniform code-region abstraction as a first step to make variability better analyzable by program analyses, laying the foundation for better studying and understanding the evolution of configurable software systems.

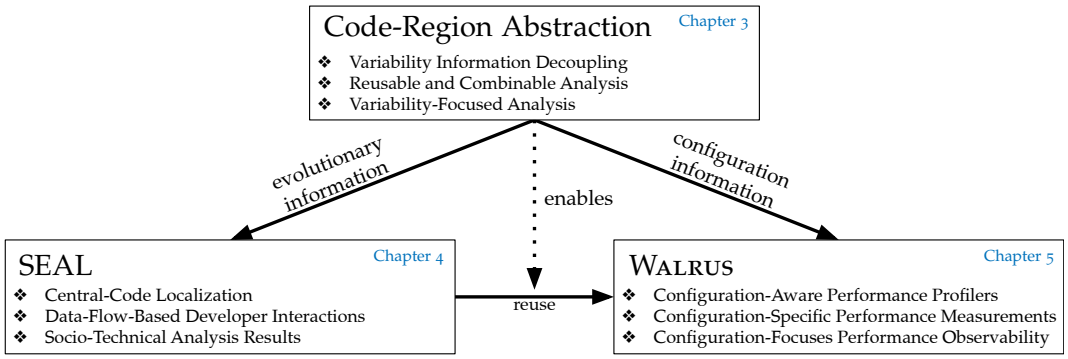


Figure 1.1: Overview of our contributions.

In summary, we have made the following contributions in this thesis:

1. We have developed a uniform code-region abstraction that integrates both time and configuration-space variability into static and dynamic program analyses in a flexible and reusable way. The uniform code-region abstraction bridges the gap between program analyses and variability information by decoupling the concrete analysis semantics from the variability information and by providing a uniform interface through which the additional information can be queried. The key benefit is that program analyses, even already existing

ones, can simply incorporate variability information or contextualize their findings with it. Furthermore, program analyses that use our abstraction can automatically include different types of variability and, by that, be reused in different contexts and settings. We have demonstrated the merits of our uniform code-region abstraction by addressing two distinct research problems that require different types of analyses as well as different types of variability: (1) we integrate evolutionary repository information into an inter-procedural data-flow analysis to infer previously hidden developer links through data-flow dependencies and, (2) we integrate configurability information, using the same data-flow analysis, into state-of-practice performance profiler, to collect configuration-specific profiling data.

2. To analyze and contextualize analysis results with evolutionary data gathered from software repositories, we developed SEAL, an approach that combines high-level program analysis with data-flow analysis. SEAL uses our uniform code-region abstraction to encode change data, mined from a software repository, into LLVM's intermediate representation, enabling a program analysis to directly access and reason about change data. We used SEAL to analyze how commits and their authors interact at a data-flow level, finding new and previously hidden interactions between developers. Furthermore, we have demonstrated how SEAL can be used to further contextualize the results of other analyses with socio-technical information (e.g., to infer authors that are related via data flow to a detected SQL injection). This way, SEAL bridges the gap between high-level repository information and precise program analyses, enabling more detailed socio-technical reasoning.

3. We have developed Walrus, an approach that enables dynamic analyses, such as performance profiling, for configurable software systems. Walrus utilizes the configuration-specific information modeled with our uniform code-region abstraction to instrument a configurable software system during compile-time with dynamic analysis hooks. Walrus is able to weave different types of measurement code from state-of-practice performance profilers into a software system to gather configuration-specific performance measurements. Through Walrus, we have made three state-of-practice profilers configuration sensitive. This way, we have been able to detect 108 configuration-specific performance regressions in 16 synthetic case studies and two real-world open-source projects. Walrus bridges the gap between high-level configurability information and state-of-practice performance profilers, connecting a decade of configurability research with modern performance analysis tools.

1.2 Outline

In Chapter 2 (*Background*), we lay the foundation for this thesis and introduce the two dimensions of configurability (*space*) and software evolution (*time*). For

each dimension, we introduce the respective foundational concepts and present the current state of the art. Furthermore, we introduce core concepts about static and dynamic *program analysis*, with a focus on whole-program inter-procedural static program analysis as well as on performance analysis. In Chapter 3 (*Uniform Code-Region Abstraction*), we introduce the backbone of the thesis, the uniform code-region abstraction that bridges the gap between different types of high-level project information and low-level program analyses. For this purpose, we first give an overview of our design and how it enables reusable program analyses. Second, we lay out a formal description of our abstraction and how it integrates high-level information into program analysis. Third, we provide an overview of the implementation that we built using the compiler framework LLVM. In Chapter 4 (*Commit Interactions*), we apply our uniform code-region abstraction to software evolution and analyze the interactions between code authors at a data-flow level, utilizing a custom code-region-based inter-procedural data-flow analysis. This way, we bridge the gap between state-of-the-art software repository analysis and program analyses, enabling detailed and precise socio-technical analyses. In Chapter 5 (*Feature Performance Analysis*), we apply our uniform code-region abstraction to *variability*, making state-of-practice performance profilers configuration aware, so that they can be used for variability-focuses performance analyses. We not only demonstrate that our uniform code-region abstraction can be applied to another use case but, more specifically, that our abstractions enables reuse, as we profit from the analyses that we developed in Chapter 4. In Chapter 6 (*Conclusion and Future Work*), we conclude this thesis by providing a discussion of our research and highlighting important problems that we identified and overcame during this dissertation. Then we conclude this dissertation and outline future work.

2

Background

In this thesis, we introduce a combined analysis abstraction that enables static and dynamic analyses to process evolutionary data along with information about the configuration space of a software system. With regard to this context, this chapter lays the foundations for our work and introduces key concepts. First, we introduce configurable software systems and their implementation techniques in detail. Second, we explain important building blocks of software evolution and contextualize our work to current research that focuses on understanding software evolution. Next, we give a concise introduction of the important aspects of the compiler infrastructure, which we later use in Chapter 4 and Chapter 5 to realize our analysis abstraction in practice. Last, we introduce static and dynamic analyses in the context of compiler assisted program analysis, focusing on important theoretical foundations as well as practical implementations thereof.

2.1 Configurable Software Systems

The key aspect of a configurable software system is that parts of its functionality are separated out and made selectable with configuration decisions. These separated out parts that encapsulate a certain functionality are referred to as software *features* in the literature [7, 43]. A configurable software system consists of a base system, which builds the core of the software project and is always executed, and variable parts that can be selected or tuned. Through this variability, users of the system can tailor it to their specific needs, for example, by selecting a specific database back-end or by removing functionality that is not needed in their use case, to improve the system's performance.

In what follows, we explain core concepts of configurable software systems in detail. We show how configurable software systems model their variability with features, through abstracting functionality into variable chunks that can be enabled or disabled. Further, we introduce core concepts and definitions about a configurable software system's configuration space, a space that represents all the different configurations. Afterwards, we highlight two important conceptual aspects by which variability in configurable software system differs. Here, we focus on binding time, which defines *when* a configuration decision is made, and implementation techniques, which defines *how* variability can be encoded in configurable software

systems. Both are relevant in the context of program analysis as they influence how and by which means a configurable software system can be analyzed.

2.1.1 Configuration Options

The central idea behind configurable software systems is to enable users to select, adapt, and tune system functionality. This is realized by making certain parts of the system variable or tuneable, and providing the user means to control this variability. Commonly, a configuration interface is exposed to the user through which the user passes information on how the system should be configured, such as command-line options or a configuration file. In general, we refer to the configuration possibilities of a configurable system as *configuration options*. This means, a configurable software system exposes through a configuration interface a set of configuration options for which a user can specify specific values to adapt the system's behavior.

Each configurable software system $\mathcal{S}$ has a set of configuration options $\mathcal{C}$, which control the variable parts of the system, meaning, they allow users to enable, disable, or tune the system's functionality.

Definition 1. $\mathcal{C}$ is the set of configuration options $c \in \mathcal{C}$ for a given configurable software system $\mathcal{S}$.

Configuration options are subdivided into two categories, boolean and numeric options. Each configuration option c has a value, which is enabled/disabled for boolean options or a specific numeric value for numeric options, and a default value that determines the options value in case none is given by the user.

Often, configuration options correspond to specific user-observable functionalities and are also referred to as software features, a term that is also commonly used in software product lines to refer to these functionalities [7]. Simply put, software product lines are configurable software systems that follow specific design approaches, for example, stepwise refinement [250] or program families [181]. However, not every configurable software system is a software product line, as they follow other design processes or are just built in an ad-hoc manner by making previously not variable parts configurable. For our work however, this distinction between software product lines and configurable software systems is not important, so we treat concepts, such as configuration options and features, synonymous.

Configuration model. Configurable systems offer a wide range of different configuration options, which from the outside seem to be freely selectable by a user. However, configuration options often depend on each other, for example, option "A" requires option "B" or two options are mutually exclusive, so only one can be selected. Configuration models, also referred to as feature models, model and document a system's configuration options and the relationships between them [7]. Based on this configuration model, we then can infer which selections of options are valid and which ones are not.

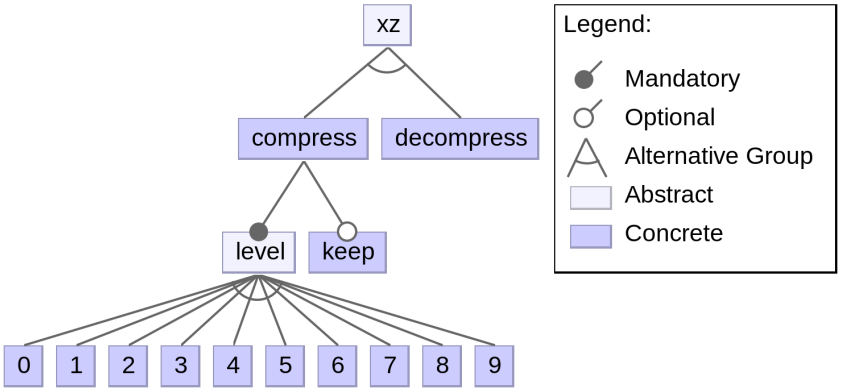


Figure 2.1: Excerpt of the configuration model from the project XZ, depicted as a configuration diagram.

In more detail, conceptually the configuration model consists of a list of configuration options and a set of relationships between them, where the relationships are encoded using propositional logical formulas. However, to ease understandability, configuration models are often encoded as configuration diagrams, also referred to as feature diagrams. A configuration diagram is a tree, where most of the relationships between options can be directly expressed by the edges between them. An edge between two configuration options encodes a parent-child relationship, indicating that `child` only makes sense in the context of option `parent`. In addition, the edges encode whether the child is optional or mandatory, by that encoding whether a `parent` $\Longrightarrow$ (implies) `child`. Abstract configuration options (tree nodes) are used to group options together and encode semantics for groups (OR) and alternative groups (XOR) (i.e., to choose one option from this set of alternatives). The leftover relationships that cannot be directly expressed in the tree are attached as propositional logical formulas, referred to as cross-tree constraints.

The visualization of the configuration diagram depicts configuration options and the relationships between them. Configuration options are represented as boxes, where the color indicates if the configuration option is concrete or abstract. Connections between the configuration options then indicate their relationships (e.g., if it is mandatory or optional). Take for example the partial configuration model from the project XZ[1] shown in Figure 2.1. XZ as the root node offers the user the option to configure either `compress` or `decompress`, indicated by the alternative group arc below XZ. Should the user choose `compress`, it is then mandatory to also select a value for the `level` option, indicating how much effort should be spent compressing the input. Interesting to note here, `level` is actually a numeric option, with a value range of 0–9, which, in this case, is encoded by 10 binary configuration options, where each indicates a value in the value space. This is a common procedure to remove or reduce the number of numeric options in a configuration model, as some approaches have difficulties handling numeric options [113, 152, 212].

1 https://tukaani.org/xz/ (Last accessed: July 26, 2023)

2.1.2 Configurations

Configurable software systems, similar to software product lines, offer a wide range of functionalities to their users that can be combined and tuned. Combining and adjusting the different functionalities are the key to making software systems adaptable and flexible for users and enable them to employ the system in varying use cases. However, the large configuration space that arises from these combination possibilities makes configurable software systems difficult to reason about by users and developers [144, 228, 252]. In what follows, we introduce the core concepts and describe how the configuration space arises from the combinatorics between the different functionalities.

To use a configurable software system, the user needs to provide a *configuration*, that is, the user needs to define which functionalities should be used or how they should be parameterized. This configuration is produced by selecting, directly or indirectly, values for each configuration option. The user either specifies the value directly, for example, by selecting a boolean option or choosing a numeric value, or implicitly selects the default value that was defined by the developers. Through the set of selected values, it is now specified how the variable parts of the configurable software system should behave. We define a configuration as:

Definition 2. A *configuration* is a set of selected values for all configuration options of a given configurable software system.

Based on the previous definition, we define the *configuration space* of a configurable software system as the set of all configurations (i.e., the set of all possible combinations of all configuration options). The configuration space of a configurable software system can quickly become excessively large. For example, a configurable system with only 10 binary configuration options already has 1024 configurations and real-world systems, such as the Linux kernel[2], have exponentially more [144, 228]. In literature this is often referred to as combinatorial explosion [7, 252], indicating that these large configuration spaces are difficult to deal with. Especially, Xu et al. [252] point out that a large part of the configuration options are used only by a very few users or not used at all. In their work, they showcase that exemplary projects could remove up to 51.9% of their configuration options without impacting their users. Considering the exponential growth of the configuration space, this would reduce a project's configuration complexity drastically, and likely lead to fewer bugs.

Configuration spaces, in more detail, are comprised out of two disjoint subsets: the set of valid configurations and the set of invalid configurations. Whether a configuration is valid, is determined by the configuration model, as this is where all relationships between configuration options are encoded.

Next, to conceptualize a configurable software system configured in a specific state, we define a system *variant*. A *variant* is the result of applying a valid configu-

2 https://www.kernel.org/ (Last accessed: September 25, 2023)

ration to the variable code base (i.e., the results of resolving the variability of the system by selecting the parts specified in the configuration). More formally put:

Definition 3. A *variant* denotes the intentional variations in a configurable software system that exist alongside each other in the *space* dimension [7].

How the configuration is applied to the code base to produce a variant depends on the kind of variability and the technique they are implemented with. Important to consider here is the binding time (i.e., when the variability is resolved) and the implementation technique used (e.g., an external generator, C++ template meta-programming, or conditionals that are evaluated at run time).

Important to note, this definition is central to this thesis as it defines one of the two major dimensions we address in our work: the *space* dimension. The space dimension conceptually represents the configurability of a software project and the variability that is introduced by the project's configuration options. In what follows, we highlight different kinds of variability more fine-grained and discuss common techniques used to implement configurability in real-world software projects.

2.1.3 Binding Times

One key aspect where variability encodings differ is their binding time. This means, the binding time of a configuration option defines *when* the variability is resolved and, by that, when a user or developer can decide on it. We divide configuration options into two broad categories: *statically* bound and *dynamically* bound. Where statically bound configuration options are decided upon early at compile time or build time, dynamically bound configuration options are decided upon later, either during startup of the program, called load-time configuration options, or during the execution of the program, called runtime configuration options [198]. As our work focuses on compile-time variability and load-time variability, we introduce these two forms in more detail.

Compile-time variability. Compile-time variability, also referred to as static variability, is decided before or during compile time [7]. A clear advantage of compile-time variability is the optimization potential. By deciding early on configuration decisions, we give the build process and the compiler additional information to optimize the final software product. This includes explicit optimizations, where developers decide to remove unnecessary code, as well as, implicit ones where compiler optimizations can produce more efficient code due to the extra information.

Load-time variability. Load-time variability, also referred to as dynamic variability, is decided after compilation when the program is started [7]. A general advantage of load-time variability is the configuration flexibility of the software product, meaning the product can be reconfigured without recompiling the system, often with little more than toggling a switch in a configuration file. However,

load-time variability can also entail a drawback with regard to performance and missed optimizations. By delaying the decision until after compile time, for example, optimizing the product by removing unnecessary functionality is not possible.

An important distinction we want to highlight between compile-time and load-time variability is that with load-time variability all variable parts are still contained in the final software product, where in the case of compile-time variability deselected parts get excluded from the final product. This distinction is especially important with regard to static and dynamic analyses. Dynamic analyses have drawbacks when analyzing compile-time variability. At the point in time when the dynamic analysis runs, the variability already has been resolved and deselected options are no longer present in the final product, so only one specific variant is analyzed. Often, this problem is resolved by running the analysis multiple times for all variants of interest, which is expensive and limits reuse of analysis results across configurations, or by creating a specialized version of the program where compile-time variability is encoded as load-time variability, often referred to as 150% program [195].

2.1.4 Implementation Techniques

To build configurable software systems, developers use different techniques and mechanisms to encode variability (i.e., they use different implementation strategies to encode choices, so that users of the system and developers can select the functionality they want). So, where binding times specify *when* a configuration choice is made, the implementation technique specifies *how* the variability is encoded into the program.

For example, build-system decisions are a technique where, depending on a choice, different files are selected for compilation or preprocessor options are used to select configuration-dependent code blocks. An often more developer-focused implementation technique uses template meta-programming[3] to select configuration specific code [76]. For example, choosing between different SAT solver implementations based on a configuration option or optimizing an algorithm by the choice of usage-specific tuning knobs (i.e., low-level configuration options for fine-tuning). Another technique to encode high-level user facing configuration options as well as low-level tuning knobs is by using configuration variables. Configuration variables are regular program variables except for the conceptual difference that they carry the configuration choice that is designed to enable or disable certain functionality. In what follows, we give a brief introduction into each technique and explain the mechanisms that are used to implement them.

Preprocessor directives. A common way to implement configurable software systems are C preprocessor directives. Before compilation, the preprocessor processes the source code and does text transformations based on preprocessor directives,

3 In C++, templates are a language specific way of expressing generic types and functions. The template itself is the generic implementation and gets instantiated by the compiler for a concrete type, generating the concrete implementation.

```
1 static void uncompress(lzma_stream *strm, FILE *file, const char *filename) {
2         lzma_ret ret;
3
4         // Initialize the decoder
5 #ifdef LZMADEC
6         ret = lzma_alone_decoder(strm, UINT64_MAX);
7 #else
8         ret = lzma_stream_decoder(strm, UINT64_MAX, LZMA_CONCATENATED);
9 #endif
```

Figure 2.2: Preprocessor example from the project XZ, where an `#ifdef` directive is used at compile time to select between two different decoder implementations based on the configuration option `LZMADEC`.

such as `#ifdef` or `#define`. Important to note, the C preprocessor is only a text processor that does text replacement without any understanding of the language syntax that it's manipulating, which makes it difficult and error-prone to use. In Figure 2.2, we show an example of a preprocessor directive (`#ifdef LZMADEC`) used in the compression tool XZ to select the correct decoder implementation. Based on the preprocessor variable `LZMADEC` that encodes if the LZMA stream decoder is available, the preprocessor will weave in either Line 6 or Line 8.

The C preprocessor is a common tool to build compile-time variability into software systems, since preprocessor directives enable developers to do complicated source code rewriting to weave in configuration specific code snippets. These `#ifdef`-based configurable software systems are regarded as cumbersome and problematic, often referred to as "`#ifdef` hell" [136, 222]. Due to the complicated rewriting patterns, code becomes difficult to understand and error-prone, as it is not clear for developers what final rewritten source code is produced in the different cases. Overall, `#ifdef`-based configurable software systems incur high maintenance costs [63]. Still they are a common and widespread way of encoding variability into a software project, even though approaches that aimed at improving these shortcomings could not catch on [64, 118, 146].

To tackle the challenges posed by the preprocessor and analyze `#ifdef`-based configurable software systems, many tools have been developed over the years. For example, SuperC [71] a variability-aware parser, TypeChef [118] a variability-aware type checker, cppstats [144, 146] a tool for extracting compile-time variability information, or Morpheus [145] a variability-aware refactoring engine.

Template meta-programming. In C++, templates are a language mechanism to write generic code. The template class or function defines template parameters, which can be types or compile-time values, that act as placeholders in the generic implementation. At the usage point of the template, the compiler instantiates a concrete implementation of the generic template code, meaning, the compiler generates a concrete implementation by replacing the generic parameters with concrete types or values determined at the usage point. From a design point of view, template parameters are often used to specify extension points in the generic

```
1  /** \brief Implementation of an hp-adaptive discrete
2   *  function space using product Legendre polynomials
3   *
4   *  \tparam FunctionSpace  a Dune::Fem::FunctionSpace
5   *  \tparam GridPart       a Dune::Fem::GridPart
6   *  \tparam order          maximum polynomial order per coordinate
7   *  \tparam Storage        for certain caching features
8   *
9   *  \ingroup DiscreteFunctionSpace_Implementation_Legendre
10  */
11 template< class FunctionSpace, class GridPart, int order, class Storage >
12 class LegendreDiscontinuousGalerkinSpace;
```

Figure 2.3: Template meta-programming example from the project DUNE. The project's documentation above the class declaration describes what can be configured, for example, with the template parameter `Storage` a class can be provided that is used internally for caching.

code where functionality should be injected, for example, with policy classes or type traits. Therefore, templates are a common way to encode compile-time variability in C++ projects.

There are many large software projects that make heavy use of template meta-programming to write generic code that can be configured for different use cases. For example, DUNE[4] [16], a framework for solving partial differential equations. Figure 2.3 depicts a **`class`** from DUNE that can be configured with different function spaces or specific storage classes. As described by the documentation, the specified storage class is used internally for caching. Using template parameters to weave in additional functionality or provide optimization opportunities is a common implementation strategy [43, 238]. In our example, the configuration parameter `Storage` enables the user of the DUNE framework to optimize the `LegendreDiscontinuousGalerkinSpace` with a caching implementation that is optimized with regard to a specific use case.

Compared to a C preprocessor based encoding, templates have the advantage to be part of the C++ programming language. This means, they not only adhere to more fine-grained language rules with regard to syntax and semantics but, most importantly, they are part of the type system. By being included in the type system, the compiler cannot only type check the code but also produce more accurate error messages. However, they still have drawbacks. Similar to **`#ifdef`**-based configurable software systems, template-based ones can become very complicated to understand and reason about.

Nonetheless, templates are highly used in C++ to build configurable software systems. Foundational work in utilizing template meta-programming for the design of configurable software systems—software product lines to be specific—has been done by Czarnecki and Eisenecker [43, 44].

4 https://www.dune-project.org/ (Last accessed: June 26, 2023)

```
1 case 'L':
2   if (compat) {
3     license();
4     exit(0);
5   }
6   control->compression_level = strtol(optarg, &endptr, 10);
7   if (control->compression_level < 1  control->compression_level > 9)
8     failure("Invalid compression level (must be 1-9)\n");
```

Figure 2.4: Example from the project LRZIP, where in Line Line 6 the configuration variable `compression_level` is set to the value parsed from command line argument `'L'`.

Larger template-based configurable systems, such as DUNE, that have many different configuration possibilities are complex to understand for users and developers. To address this, Grebhahn et al. [76] introduced a lightweight semi-automatic approach that extracts variability information from the project and presents the discovered variability to the user as a configuration diagram. Furthermore, through the extracted information it becomes easier to locate the part in a code base that implement template variability.

Configuration variables. Another very common implementation technique that is often used to encode load-time configuration options are configuration variables (i.e., program variables that carry the information about the configuration choice during the execution of the program). Initially, the configuration value is loaded at program start, by parsing it from command line flags or loading it from a configuration file. Then, the configuration value is stored in the configuration variable which represents the configuration option throughout the program and is referenced whenever the value is needed elsewhere in the program. Configuration variables are often grouped together in a configuration context object, which is passed to all parts of the code base or is accessible as a global variable. One key drawback is that it can be difficult to reason about where configuration options have impact in a large code base, as all configuration variables are often accessible with the configuration context object from anywhere.

Figure 2.4 depicts the configuration variable `compression_level` from the compression tool LRZIP[5], which is used to control how much effort is spent to compress the input. At the start of LRZIP, the configuration value is parsed from the command line and stored in the configuration variable `compression_level` (Line 6). In LRZIP, configuration variables are grouped together by a `struct lrzip_control`, which is accessible in the whole code base through the global variable `control`. So, as noted earlier, determining all places that are directly or indirectly influenced by a configuration variable is difficult for a developer.

To address this issue, Lillack et al. [147] developed LOTRACK, an approach that uses a taint analysis to determine which parts of a program are directly or indirectly

5 https://github.com/ckolivas/lrzip (Last accessed: December 26, 2022)

controlled by a configuration variable. Through LOTRACK, one can locate load-time configuration-specific code.

2.1.5 Feature Interactions

As previously mentioned, large configuration spaces make analyzing configurable software systems difficult, as standard analyses often need to analyze each variant for itself. A standard technique to mitigate this issue is sampling [112, 152, 202, 212]. The sampling technique selects a representative subset of the configuration space and only then is the subset analyzed. The idea is to reduce the number of variants to analyze, while still covering the configuration space with regard to the effect the analysis wants to observe. However, even specialized sampling techniques, such as feature-wise sampling [213] or distance-based sampling [113], that where devised for configurable software systems, cannot guarantee that the sample set contains all effects the analysis wants to observe. The reason that makes selecting a subset difficult are potential interactions between different features, that is, enabling a feature might alter or influence the behavior of another feature. So, an effect or analysis finding may only be visible if a specific set of configuration options are selected or deselected, activating the respective features. Apel et al. [7] define a feature interaction and the problem to detect and manage them as follows:

Definition 4. A *feature interaction* between two or more features is an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved.

An *inadvertent feature interaction* occurs when a feature influences the behavior of another feature in an unexpected way (...).

The *feature-interaction problem* is to detect, manage, and resolve (inadvertent) feature interactions among features. [7]

Following this definition, it becomes clear that a feature interaction can potentially be between any number of features of a configurable system, where higher interactions that affect multiple features at the same time, are less probable but still cannot be excluded [125]. Previous research has shown that feature interactions cause a wide range of problems and affect different fields [5, 6, 8]. For example, the null-pointer dereference in the LINUX Kernel that was fixed by commit `6252547b8a7`[6] only occurred when TWL4030_CORE was enabled but OF_IRQ was disabled [1]. Kolesnikov et al. [124] give another example in MBEDTLS, where the choices of block cipher mode and hash function interact and can have a negative impact on performance. Hence, finding and analyzing feature interactions is important to detect such problems.

Depending on the way how features interact, there is often no way of determining a feature interaction up front without running the actual analysis on the specific

6 https://github.com/torvalds/linux/commit/6252547b8a7acced581b649af4ebf6d65f63a34b (Last accessed: October 5, 2023)

configuration that includes the interacting features. From previous work, we know that features can interact with each other in a multitude of ways and that feature interactions appear in many domains [6]. Furthermore, it is not always clear how to categorize them: effect (good vs bad), intention (intended vs unintended), or context (design vs implementation) [6]. With regard to program analysis, we group feature interactions into three categories: *structural* interactions, *program-flow* interactions, or *non-functional-property* (NFP) interactions. This way, we differentiate feature interactions by the type of analysis that is used to detect them.

Structural feature interactions. The commonality of *structural* feature interactions [9] is that the same code is shared between multiple features. Feature interactions that emerge through a structural overlap of feature code are often the simplest to detect, as they are encoded directly. For example, nested `#ifdef` blocks, where one feature block is nested or overlaps with another block.

Structural feature interactions arise from the fact that features are often crosscutting concerns [10, 43, 144] (i.e., the implementation of the feature is spread over the code base). In general, the design principle of *separation of concerns* aims to program code into separated parts [55, 180, 190]. The goal is to produce modular code, where each modular unit implements a specific concern, for example, a specific task or algorithmic approach, but is separated from the rest of the program by a well-defined API. This way, the details that are needed to implement the modular unit are hidden from the rest of the program to encapsulate complexities. However, as Kiczales et al. [121] note, some concerns cannot easily be represented through procedural or object-oriented programming techniques, leading to "tangled" code that is scattered all around the code base. These concerns are referred to as *crosscutting concerns*, where the term crosscutting describes the structural relationship between the representation of two or multiple concerns [7]. The shortcoming of not allowing a clear separation between concerns of classical procedural or object-oriented programming languages is referred to as the *tyranny of the dominant decomposition* [227]. The consequence of this is that many feature implementations cannot be implemented in separation and, by that, are scattered throughout the code base. In the presence of multiple features this can lead to crosscutting features, which leads to structural feature interactions.

Program-flow feature interactions. Compared to structural feature interactions, program-flow based ones are non-local, meaning that the execution of one feature modifies the program state in a way that later influences other features. These interactions happen through control or data-flow connections between the features. For example, the execution of one feature might permutate previously sorted data that is later passed as input to another feature which then behaves differently because it expected the data to be sorted. From a source code perspective, there might be no overlap or syntactic relation between the two features, but still enabling one causes a bug in the second one because of data dependencies between them. Detecting program-flow interactions requires more elaborate program analysis

compared to structural ones, as they need to model and analyze the control or data flow of a program. For example, Kolesnikov et al. [124] demonstrated that control-flow dependencies between features can be used as a predictor for feature performance interactions. However, their findings also indicate that control flow alone is not enough to reliably predict feature performance interactions and they hypothesize that data-flow dependencies could improve prediction accuracy.

NFP feature interactions. Non-functional properties of software systems, such as performance, binary size, or memory consumption [216], are important to users in various application scenarios. Non-functional properties are especially important in the domain of embedded or real-time systems where the application is constrained by limited amounts of memory or hard time requirements [91, 92]. Similar to functionality, the NFPs of a configurable software system are also influenced by the feature selection and are impacted by feature interactions. Detecting NFP feature interactions can be challenging, as there might not be any direct connection in the program between the involved features. For example, enabling a feature might lead to worse performance of another feature because the data that was loaded by the first feature displaced data from the cache that is later needed by another feature, resulting in more cache misses and longer latencies for the second feature. This means, detecting NFP feature interactions typically requires a dynamic analysis that measures the NFP for all variants. In general, performance is an important NFP for software systems, including configurable ones. We highlight these performance-specific feature interactions in Section 2.3.7.3 in detail, where we focus on the performance analysis of configurable software systems.

There exist various forms of inadvertent feature interactions, resulting from different implementation techniques, properties, or objectives. Overall, these inadvertent feature interactions pose a challenge to developers and users that encounter them, as well as, to the analyses that aim to identify them. In our work, we do not focus on a specific technique or objective but treat configuration variability uniformly, represented by the *space* dimension, and aim to integrate this type variability information into program analysis.

2.2 Software Evolution

With an ever-changing environment and new user demands, modern software projects are forced to constantly evolve and adapt. To facilitate that, modern development practices have changed, from the traditional waterfall model [200] to agile software development practices [36, 150], such as extreme programming [18]. The new agile processes center around iterative and incremental development, which makes it easier to incorporate changing needs of the customer, even late in the development cycle. However, these new processes also pose new challenges, for example, how do developers from large and distributed teams coordinate in agile develop-

ment processes [224]. Solving these challenges to improve development practices and furthering the understanding of software evolution is important. As Mens and Demeyer [155] put it: "The ability to evolve software rapidly and reliably is a major challenge for software engineering.". However, to understand and improve these processes it is essential to analyze the evolution of existing software projects and find new methods that allow us to better understand how software is developed (e.g., we need to better understand how new changes interact with existing code). Another key aspect that needs to be incorporated into the big picture is the socio-technical context around software projects. Modern software projects are developed in a highly collaborative manner [155], meaning they are developed by many individual developers. So, incorporating the human factor into the understanding of a software project evolution is key to get a better understanding.

In what follows, we give an overview of modern version control systems and how they enable the development of constantly evolving software systems. We explain the crucial role of repository mining in the research process to extract and reason about these evolving software systems. Next, we highlight key aspects to understand the socio-technical context around a software project, which is important to consider when aiming to understand the evolution of a software system, as the developers of the system are the drivers of software evolution. Afterwards, we contextualize software evolution with regard to configurable software systems, as analyzing them poses additional challenges because the evolutionary changes impact the systems' configurations differently and can also influence the configuration space itself.

2.2.1 Version Control Systems

The goal of version control systems is to manage, store, and orchestrate different states of a software project, to enable developers to work simultaneously on a software project. As software projects are developed incrementally [155] by multiple developers, version control systems need to orchestrate the different project states that arise through the many changes. Furthermore, version control systems need to preserve older project states, so that developers can look back and reason about the changes made. Hence, a central concept of version control systems is the *revision*, which specifies a software project's state at a specific point in time, with regard to one or multiple previous states. Thereby, revisions create an order or partial order between different project states. Note, for version control system that are only partially ordered, we can create an order by linearizing the revisions through their timestamps [103].

Definition 5. A *revision* is the ordered variation of a software system along the *time* dimension [7].

Important to note, this definition is central to this thesis as it defines the second major dimension we address in our work: the *time* dimension. In general, version control systems manage the revisions of a software project over time and provide

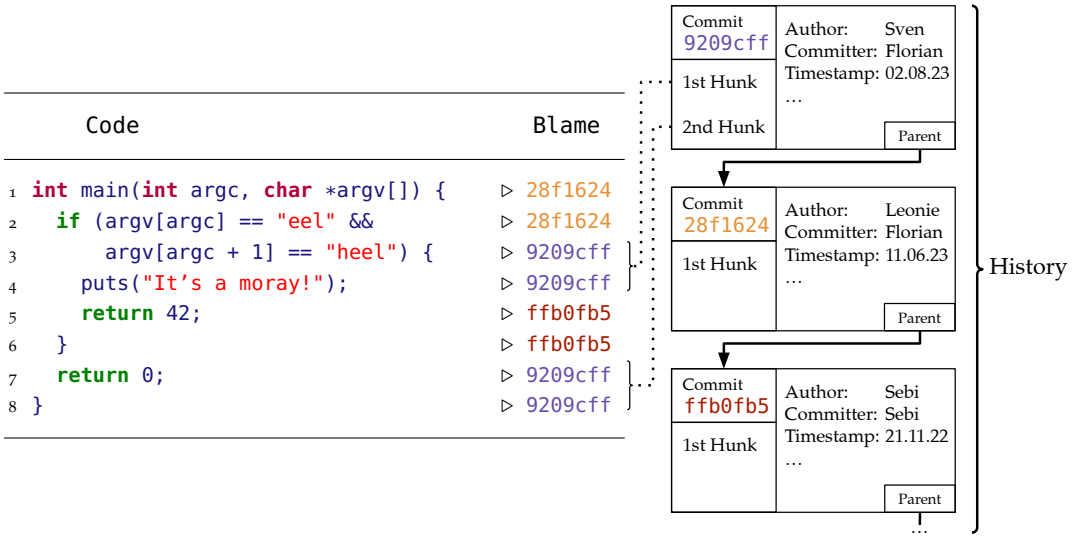


Figure 2.5: Overview of important GIT-specific terminology.

functionalities to work with them. For example, they can compute the textual difference between two revisions or provide a way to bring the software project into the specific state that is described by a revision.

GIT[7] is a version control system that has gained wide acceptance over the last decade[8] [53]. In what follows, we introduce GIT-specific terminology that we later use and relate them to central theoretical concepts, as our practical implementation is closely tied to GIT. Figure 2.5 depicts the GIT-specific terms and visualizes their relation to each other.

Commit. In GIT, a *commit* represents a revision of the software system. Conceptually, a commit is a set of changes. The changes encoded in the commit specify how a revision can be produced based on the previous commits (i.e., they specify how the previous states of the system need to be transformed). In detail, each commit is comprised of one or multiple *hunks*, where each hunk represents the changes made to a separate part of a file (i.e., it describes the differences to the previous state of that specific part). Applied together, all hunks describe how all files need to be transformed on a textual level. In addition to the changes, commits also encode meta-data, such as author (i.e., the person who wrote the change), authoring timestamp, or committer (i.e., the person who committed the change). These meta-data are important for analyses as they associate a change set with the developers that work on it and can, therefore, be used to embed changes into the socio-technical context of a software project.

7 https://git-scm.com/ (Last accessed: July 3, 2023)
8 https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems (Last accessed: May 16, 2023)

Commit history. All commits together form the history of a software project. Each commit describes the changes with regard to the previous commit, referred to as parents. This way, the parent relationship explicitly encodes the ordering between the different revisions, that is, it specifies in which order the changes need to be applied to produce a specific revision. This explicitly captured history is valuable for understanding the evolution of a software project, as we can post-hoc retrace the different revisions of the software project and see how it evolved over time.

Blame. Technically, *git-blame* is a GIT command that annotates at a given revision each line in a file with the commit that last modified it. Conceptually, the blame view collapses the creation history of a file and visualizes for a revision what changes in the history of the software project produce a specific line in a file. From a software evolution standpoint, this shows us for a revision which parts of a commit did survive over the evolution of the software project and are still relevant.

Overall, version control systems are a rich data source for the analysis for software projects, as they preserve important details about the evolutionary steps of a software project (i.e., they track all textual code changes, the order in which they were made as well as when and by whom the changes were made).

2.2.2 Repository Mining

Software repository mining focuses on gathering, modeling, and studying the data and software artifacts produced by developers during the software development process [46, 48]. Conceptually, the mined data enables us to connect the technical changes made over time to a project's developers, enabling us to reason about how the software project evolved.

A project's revisions (i.e., the source code changes) are especially important for evolutionary and socio-technical analyses because they encode the history of the software project and closely tie developers to the produced code. Kagdi et al. [110] state that source code changes are the fundamental unit of software evolution. Bird et al. [22] also describe that GIT offers a wide range of useful data but also point out potential pitfalls that can be encountered when mining data from GIT repositories, such as that histories can be edited by developers afterwards. However, by incorporating the historical data encoded in the revisions, one can analyze how a software project evolved over time. For example, Xie et al. [251] investigated several evolutionary patterns (law's) in seven long-lived software projects and analyzed the distribution of software changes and growth rate of development and maintenance branches. They could confirm many proposed patterns and, furthermore found that a high percentage of changes are concentrated on a small percentage of code, describing these regions of code as "change hot spots". Kagdi et al. [110] further mention that current version control systems do not provide information about code

semantics, so most of the analyses only use syntactical information when reasoning about changes. A shortcoming that we address with SEAL (see Chapter 4).

An important application where the mined data can also help to better understand how software evolves is social-network analysis. The data extracted from software projects provides a way to connect source-code changes with the developers that produced them and, by that, enables us to analyze social interactions between developers that arise from the technical work. These connections, based on the mined data, let us then reason about the socio-technical context in which the software project evolves and how this social-technical context evolves over time.

2.2.3 Socio-Technical Context

Modern software systems are developed in a highly collaborative manner [155] by many individual developers, which all need to organize and work together to realize the system. To further improve development processes, we need to understand the social factors of how developers interact and how they work together to build successful software systems. However, we should not consider the social factors in isolation but together with the technical artifacts of the software system, as both together form the software project. Studying social factors together with technical artifacts is difficult, especially for open-source projects where hierarchies and organizational structures are often not explicit, but also necessary to get a better understanding how successful software systems are developed [226].

The overall goal is to reach socio-technical congruence, a state where the social and the technical sides interact properly [32]. According to Herbsleb and Grinter [95], socio-technical congruence is essential for project success. However, to build socio-technical congruent development processes, we first need to study and analyze the technical and social evolution software projects.

A key problem for socio-technical congruence is to determine when developers should coordinate between each other, meaning, when do technical dependencies and requirements need social interplay between developers [32]. Code changes of one developer can impact the code of other developers, for example, adding a new method to an interface that now needs to be fulfilled by all classes implementing it requires other developers to adapt their code. In many real-world scenarios determining these dependencies and interactions is difficult, so overall the question arises: ”When should two developers coordinate?”.

Answering such questions is difficult and requires detailed data about the evolution of the technical and social side of a software system. To gather the data, repository mining is an important building block for socio-technical research, as it extracts valuable data from the development history of the project.

For example, to better understand how developers coordinate, work by Joblin et al. [105] investigated the communication structures between developers in 18 large open-source projects. They analyzed how the communication structure changes when projects evolve and grow over time. An important building block in their

analysis is the function-level semantic coupling data. They computed this coupling data by first extracting source code and comments from the repositories' code base and then processing it further with text mining techniques to extract the semantic connections between developers. Their work showed that organizational structures of large projects adapt to balance the costs and benefits of developer coordination.

Another example, where data mined from software repositories is utilized by socio-technical research, is to reverse engineer organizational structures for open-source projects. Determining and analyzing the organizational structure of a software system is important because, following conway's law [39, 132] and the mirroring hypothesis [37], a projects organizational structure influences the technical structure of the software project. However, open-source projects often lack an explicit organizational structure. So, to further study these, one has to first reverse engineer them from available data sources. To infer organizational structures of open-source projects, recent work by Joblin et al. [106] uses repository mining approaches to discover communication links between developers and, based on these, identify the organizational structures. A key data source for their work is e-mail communication extracted from mailing lists and issue data mined from GitHub. Based on the mined data, their network approach helped to uncover that open-source projects have hybrid organizational structures with hierarchical and non-hierarchical parts.

Factoring in the socio-technical context is necessary to obtain a holistic understanding of the evolution of a software system, especially for configurable software systems. However, socio-technical reasoning about configurable software systems is currently still difficult. Current approaches often analyze the evolution and socio-technical context of configurable software systems without incorporating the variability that arises from the configuration space.

2.2.4 Evolution of Configurable Software Systems

As previously discussed, understanding the evolution of software systems and the socio-technical context they are developed in is important to improve development processes (e.g., to improve coordination requirements [97]). However, this becomes more difficult for configurable software systems, as alongside the evolutionary dimension, which adds variability over time, there is also the spacial dimension (i.e., the variability introduced by the configurability of the system). As both of these dimensions interact [233], investigating one without incorporating the other cannot lead to a complete picture. So, only a joint approach that incorporates both evolutionary variability and configuration variability is suited to analyze how configurable software systems evolve.

In what follows, we relate software evolution to configurable software systems and describe the peculiarities that arise through this in detail. We introduce the central concept of a software *version* that combines the configuration space of a configurable software system with a specific revision from the system's development history. We explain why the influence of the evolution on the configuration space makes

analyzing the evolution of configurable software systems particularly challenging. Furthermore, we highlight recent research that aims at understanding the evolution of configurable software systems.

We define a *version* as the combination of a variant together with a revision.

Definition 6. A *version* of a configurable software system is comprised of a revision (*time* dimension) and a variant (*space* dimension) [7].

This way, we introduce a term that unifies both *time* and *space* dimension. Put differently, a *version* is the result of applying a valid configuration on a variable code base at a specific point in time. The combination of both dimensions is necessary as only together we can make precise assessments. Analzying a revision of a configurable software system, without incorporating configurability, can give a distorted picture of the project's state. On the other side, analyzing a configurable software system without specifying at which point in time (revision) the project was analyzed, makes the results difficult to interpret, as they might only be valid in a specific time range. Hence, we unify both dimensions into a *version* of the configuration software system.

Analzying the entire version space of a configurable software system is challenging and extremely computationally expensive. To conceptually get a full picture, the already potentially very large configuration space would need to be analyzed at every revision. Meaning, we would get a worst case estimate of $\mathsf{numberOf}(\text{revisions}) * \mathsf{largest}(\text{configuration space})$ configurations to evaluate. That is why analyses need to carefully evaluate which kinds of information to incorporate and how to incorporate it, as for many interesting questions, fully exploring the version space is not necessary.

Combining the space dimension with the revisions gives us more than just multiplying existing analyses along the time dimension. Adding revisions builds a bridge to the evolutionary information that is present in software repositories. Research on software evolution already has demonstrated that incorporating evolutionary information, such as change histories and socio-technical information, can provide benefits for understanding software projects. So, by focusing our analyses on *versions* we enable the incorporation of evolutionary information into existing analyses that target configurable software systems.

However, accessing and incorporating all this additional information into analyses is a further burden for analysis developers, consuming time and adding analysis complexity. Among other things, we try to address this problem of incorporating both time and space variability into existing analyses with our unified region abstraction, which we introduce in Chapter 3.

Work in the recent years suggests there is an interest in combining both dimensions to analyze configurable software systems. For example, Hunsen et al. [97] investigate socio-technical congruence in open-source software projects, by assessing the alignment of collaboration and communication through coordination requirements. To determine coordination requirements, they use the coupling of different abstraction levels of source-code artifacts. Among other things, they analyze coordi-

nation requirements arising from features. Important to note, their study focuses only on compile-time variability implemented through preprocessor directives. Having a more uniform access to variability information can enable such studies to consider all kinds of binding times and implementation techniques. Furthermore, having a combined access to the socio-technical context of the code base and feature information would probably have made it easier to relate the detected feature code to the developer information.

In addition, recent work by Mühlbauer et al. [163, 164] started to jointly analyze the version space of configurable systems. Initially, they developed an active sampling approach to detect performance changes, especially regression, in the performance history of configurable software systems [163]. Their approach analyzes the performance of specific revisions, sampled with an adaptive sampling approach to minimize the amount of measurements needed, to build up the performance history of a configurable software system in retrospect. With their work they only need a small set of measurements to determine important performance change points along the time dimension of a configurable software system. In follow-up work, Mühlbauer et al. [164] extended their approach to also include the space dimension, enabling them to detect *version* specific performance changes. However, currently their approach is black-box based and does not enable developers to analyze the detected performance changes alongside the changing code. Again, a uniform interface to access variant and revision specific information can help here by determining the affected configuration specific code and putting the measured results into the socio-technical context by connecting them to the involved developers.

These examples not only demonstrate the potential and interest in *version* focused analyses, but also show that a uniform abstraction to access variability information from both dimensions could ease the development of such analyses. However, in the end we need a unified way of connecting all these different types of data.

2.3 Compiler-Aided Program Analysis

Program analyses are vital tools to better understand and reason about programs. The goal is to develop a program—the analysis—that reasons about the behavior of given input programs, for example, to optimize the input program, find flaws, or reason about a specific property. Program analyses are divided up into two broad categories: static analyses and dynamic analyses. Where *static* analyses reason about the behavior of a program without actually running it, *dynamic* analyses are executed at run time together with the target program, either woven into the program as instrumentation code or alongside it.

Static analyses take a program in an abstract representation as input and try to determine properties about the program behavior. Important to note, all non-trivial behavior properties are mathematically undecidable, following from Rice's theorem [196] and the halting problem [237]. This means, sound static analyses need to overapproximate, for example, at points where runtime specific information

is needed, computing a solution would take long, or where potentially no solution could be computed. Compared to static analyses, dynamic analyses are not constrained by undecidability, as they follow actual execution traces, but they are limited by the number of traces that can be produced. Dynamic analyses cannot reason about the unexecuted parts of a program, so the number of traces and the information contained within them, limits what they can prove. This can be especially problematic for configurable software systems, where currently disabled parts are not visible to the dynamic analysis. Furthermore, dynamic analysis can alter the behavior of the target program by observing it—similar to the observer effect in physics. For example, performance measurements perturb the analyzed target program by competing for or influencing system resources, such as CPU or cache memory. We discuss a more detailed example of such effect in Section 2.3.7.

In what follows, we give a brief introduction into the large area of program analyses. Initially, we focus on important theoretical concepts and frameworks on which our later work builds. Next, we introduce the LLVM compiler infrastructure, a versatile compiler framework that offers building blocks to assemble program analyses. Afterwards, we highlight PhASAR, a LLVM-based analysis framework, through which we implement advanced inter-procedural program analyses. In the end, we give a brief overview of performance analysis, an area of program analysis that is concerned with measuring and understanding the performance of programs.

2.3.1 Data-Flow Analysis

The goal of a static data-flow analysis is to compute dynamic properties of a program by only analyzing its representation (e.g., source code) and without actually executing it. Therefore, the data-flow analysis utilizes data-dependencies and flow information to compute the properties for each program component that are guaranteed to hold at a point on all possible program executions. Common applications that use information computed by data-flow analyses are compiler optimizations, for example, to compute which variables could still be read (liveness analysis), or security-focused analyses that, for example, want to guarantee that a specific data value cannot be read by the user.

Data-flow analyses take as an input a representation of a program, usually a set of functions where each function is represented as an intra-procedural control-flow graph, and analyze each function. An intra-procedural Control-Flow Graph (CFG) is a representation that depicts a function of a program as a graph, where each node is either an instruction or a block of instructions (called basic block) that are executed in sequence, and the directed edges between the nodes indicate where the execution can continue after the execution of the current node. To analyze a program, a data-flow analysis defines the analysis *lattice*, an abstract structure of the information that the analysis wants to track, as well as, a set of data-flow equations that specify the analysis semantics. In short, a complete analysis *lattice* is a partial order $(S, \sqsubseteq)$ where a least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) exist

for each subset of the lattice [162]. So, the lattice describes the different states the analysis properties can be in. Commonly, the largest element of the lattice is referred to as the *top* ($\top$) and the smallest one as the *bottom* ($\bot$). The *transfer* function specifies how the desired properties for a given node in the CFG are computed and the join function describe how analysis properties from multiple nodes should be combined. A data-flow solving algorithm then computes the data-flow properties that hold at each node in the graph, by iteratively applying the data-flow equations until a fix point is reached. Note that, to guarantee termination for a given data-flow analysis, the transfer function needs to be monotonic and the height of the lattice finite.

Taint analysis. A taint analysis is a specific form of data-flow analysis that aims to determine if and which parts of a program depend on specified data sources. Conceptually, the analysis defines data *sources* and *taints* every part of the program that uses data produced by a source (i.e., the analysis marks every part of the program that is dependent on the data source). This way, the analysis determines if specified parts of the program, called *sinks*, that should not be able to access data from a *source* could do so. Taint analyses are often employed in the security context, for example, to detect if sensitive information could be leaked to the user [13, 27, 142, 204]. In such an analysis, the variables containing the sensitive information are defined as taint *sources* and the functions that can give information to the user are defined as *sinks*. Then, if no taint can reach a sink, it is not possible to leak the sensitive information, within the bounds of the analysis specification. For example, to check that a server's private key cannot be leaked, the analysis would define the variable containing the key data as a *source* and all functions that send data to the user as *sinks*. The taint analysis then taints all values that stem from a source variable. If one of these tainted values is passed to a function marked as sink, the analysis would report this as a potential way to leak sensitive information.

Another application of a taint analysis in the context of variability is to determine which parts of a program are related to configuration options. Developed by Lillack et al. [147], the tool LOTRACK uses a taint analysis to infer which statements of a configurable software system are controlled by which configuration options (i.e., which statements can only be executed if at least these configurations options are set). To compute this information, Lillack et al. [147] define read operations that load configuration options as taint *sources* and then use the taint analysis to taint all control-flow decisions (*sinks*) in the program that depend on the loaded options. This way, they can compute which parts of a configurable software system can only be executed given a set of configuration options. We later use such a taint analysis in Section 5.2.2 to infer load-time dependent code.

2.3.2 Inter-Procedural Analysis

We previously simplified that a data-flow analysis represents a program as a set of functions and analyzes each function separately, by that describing how *intra-*

procedural analyses work. However, only analyzing each function for itself limits the precision of the analysis and what claims it can make, as the analysis does not consider the interactions between functions and, therefore is required to over-approximate at call-sites. *Inter*-procedural analyses overcome this shortcoming by additionally modeling call relations between functions, creating and then analyzing a more holistic representation of the program.

Compared to intra-procedural analyses, inter-procedural analyses analyze the program represented as an *inter-procedural control-flow graph*. The inter-procedural control-flow graph is created by introducing additional graph edges between call sites and the called functions (i.e., the different functions in the CFG get *interconnected*). Each graph node representing a call site is split into two artifical nodes, one representing the analysis state before the call the other after the call. We then introduce an edge that maps the local analysis context before the call to the called function and another edge that maps back the analysis context from the return of the called function to the node that represents the state after the call. By these transformations, the resulting inter-procedural control-flow graph models inter-function relationships and enables our analysis to analyze the program as a whole. Important to note, these transformations only encode static call relationships precisely without overapproximation. With static, compared to dynamic, call relationships we refer to function calls that can be determined at compile-time, either because they are static in the first place or because they can be devirtualized (i.e., we do not need to determine the callee dynamically). Dynamic call relationships, introduced by virtual calls or function pointers, need to be modeled with a set of functions that could be potentially called (i.e., we need to add graph edges to each function that could be called and merge the resulting analysis states of all these functions afterwards). The resulting overapproximation can be reduced by helper analyses that provide more accurate call graphs. However, in general inter-procedural analyses enable the reasoning about properties that span over multiple functions and, by that, can generate results that intra-procedural ones cannot infer. For example, a taint analysis needs to be inter-procedural to detect a flow if source and sink are in different functions.

Inter-procedural data-flow analyses can have different properties that impact their precision, most importantly: *context sensitivity*, *field sensitivity*, and *flow sensitivity*.

Context sensitivity. Up to now, even with inter-procedural control-flow information, the analysis analyzes each function only in a generic sense, that is, the analysis aggregates the analysis states of all incoming call edges, analyzes the function, and then propagates the information back to all call sites. This introduces an additional amount of overapproximation as generalized analysis results of a function represent the analysis state that is valid for all call sites. However, given a specific analysis context more precise results could be possible.

The idea behind *context sensitivity* is to produce call-site specific analysis results for function calls, which only need to be valid for the given calling context (i.e., they only need to be valid for the specific call instruction that is analyzed). By

contextualizing the analysis of a function with a specific call site, the analysis can then analyze the function for the specific analysis context that holds at the call site, which reduces the overapproximation introduced by spuriously propagating the functions' analysis results to all potential call sites. Conceptually, this can be thought of as cloning the function for each call site and analyzing each clone separately. However, as cloning and analyzing each function multiple times would introduce a considerable amount of analysis overhead, modern approaches use techniques, such as call strings or the functional approach [210], to implement context sensitivity more efficiently by computing reusable function summaries.

Field sensitivity. Another source of overapproximation are record types, such as structs or classes, or arrays. A record type is a collection of fields, where each field for itself has a name, its own type, and location in the memory of the record type [184]. Hence, when a variable of a record type is created in memory, different parts of the allocated memory belong to different fields. Which raises the question: How should an analysis model such a variable? Field-insensitive analyses merge the different fields of each record and treat all fields together as one variable. This means that all fields of a record share a possibly overapproximated common value of the lattice. However, by that the analysis cannot differentiate between different fields, which leads to overapproximation.

To address this issue, field-sensitive analysis instead uses specialized lattices that keep different abstract values for the different field names [162] or encode the additional information into the data-flow domain. The later is common, for example, in the IDE framework, which we explain afterwards in Section 2.3.3. An approach to implement field-sensitivity that works well in practice is to distinguish fields by their access path and propagate lattice values for each of these field access paths [116]. This means, the analysis tracks for records, which potentially have other record types as fields, the path through which fields and subfields are accessed and persists this information in the analysis state. However, as loops or recursion can introduce an unlimited amount of field indirections and analyses need a finite field access path in order to terminate, modelling the complete field access path is not always feasible. Hence, techniques, such as k-limiting, bound the depth of the field access path to a predefined constant k (i.e., they only provide field-sensitivity up to k indirections and afterwards overapproximate). However, practical evaluations of k-limiting show that this is often not a problem, as even $k = 3$ enables analyses to model most of the field accesses [51, 205]. Furthermore, recent research has shown that synchronized pushdown systems[9] are an efficient replacement for k-limiting, providing the same precision as $k = \infty$ while being as efficient to compute as $k = 1$ [220, 221].

Flow sensitivity. Another important property of data-flow analysis is flow sensitivity. Flow-insensitive analyses ignore the order of statements in a program and

9 Synchronized pushdown systems are a context-, flow-, and field-sensitive approach to solve data-flow problems. That is, synchronized pushdown systems solve two CFL-reachability problems synchronously: one for context-sensitivity, and one for field-sensitivity, where both are flow sensitive.

compute an overall solution for the program. This means, they only compute information that holds for the entire program and cannot reason about specific points. Flow-sensitive analyses incorporate the information contained in the program's CFG into the analysis, meaning, they take the statement order of a program into account and differentiate analysis states at different program points. For example, where a flow-insensitive taint analysis determines that a program variable is tainted, a flow-sensitive analysis can differentiate for a given location in the program whether the variable is tainted at this point.

Context, field, and flow sensitivity are important properties for our inter-procedural analysis that we introduce later in Section 4.2.3, as they reduce overapproximation and thereby produce more accurate results.

2.3.3 Inter-Procedural Distributive Environments

Up to now, we have discussed general program-analysis concepts. In what follows, we give a short introduction to the conceptual Inter-Procedural Distributive Environments (IDE) framework [192, 201], which we later use to implement our inter-procedural data-flow analysis (see Section 4.2.3).

IDE is an algorithmic framework that pursues the functional approach [210] to implement fully context- and flow-sensitive, inter-procedural data-flow analyses. This is done by transforming the data-flow problem into an efficiently solvable graph reachability problem, by constructing a so called *exploded super-graph* (ESG) that describes which data-flow facts hold at which statement. So, to check whether a property of interest holds at a certain point in a given program, we need to determine if the corresponding ESG node is reachable from a special tautological node Λ. We construct such an ESG by replacing each node in the program's inter-procedural control-flow graph with a bipartite graph representation of the corresponding flow function. A flow function in IDE encodes the effect an instruction has on a data-flow fact, based on the instruction's semantics. Then, if a node (I, d) in the ESG is reachable from Λ, the data-flow fact d holds at instruction I, meaning there is a potential connection (e.g., a data flow) from d to I.

In addition, to the data-flow semantics that are encoded in the flow functions, the ESG's edges are annotated with lambda functions to specify additional computations. These so-called *edge functions* allow to encode a value computation problem that is solved when performing the reachability check from above. This means, IDE determines a value of the edge domain by successively executing the composition of the edge functions alongside the data-flow path, encoded through the flow functions. It is crucial to understand that the data flow and the value computation problem are decoupled in the sense that the flow functions encode the semantics of how data is passed between instructions, the edge functions encode the additional value computation problem that we want to solve (e.g., the lambda functions could

encode which feature taints are propagated along the ESG edges in an analysis similar to LOTRACK [147]).

Important to note, the runtime complexity of IDE is $\mathcal{O}(|N| \cdot |D|^3)$, where $|N|$ is the number of nodes in the inter-procedural control-flow graph and $|D|$ is the size of the data-flow domain D [192, 201]. Thus, analysis efficiency highly depends on the size of the underlying data-flow domain. The computations that can be specified along an ESG's edges operate on a separate value domain V. The value domain V can even be infinite and does not affect the algorithm's complexity, as long as all operations on edge functions, such as composition and join, can be performed in constant time.

2.3.4 Compiler Infrastructure

After conceptually introducing inter-procedural data-flow analysis and IDE, in the following we focus on the technology stack that we use to implement such analyses, which are then able to analyze real-world software projects.

Analzying real-world software projects is a difficult but necessary task for understanding the evolution of configurable software systems: It is difficult because commonly used high-level programming languages introduce a multitude of complexities into the analysis process (e.g., parsing the context-sensitive language grammar of C++). But it is also necessary, as understanding the evolution of configurable software system is only possible by observing how these configurable software systems evolve in the real world. To address these problems, we build on existing compiler technology and research that strips away many of the complexities by introducing a general language-independent abstraction that is easier to model in our analyses. Specifically, we selected the LLVM [134] compiler framework, as it not only offers such an abstraction but also offers a variety of helper analyses and tooling to implement our analyses.

In what follows, we give a brief overview of LLVM and its core components that we later build upon. We highlight the C/C++ front end CLANG and LLVM's intermediate representation (LLVM-IR), as the general abstraction that LLVM offers. Furthermore, we give a short introduction to LLVM's analysis infrastructure that offers a variety of helper analyses.

LLVM offers a wide range of reusable tools, data structures, and utilities, for building compilers. Conceptually, a compiler built with LLVM is decoupled into three parts: a language-specific front end, a shared optimizer, and architecture specific back-ends. For example, the C/C++ compiler CLANG focuses primarily on parsing and mapping the high-level language to LLVM's internal representation. For optimization and code generation, CLANG simply reuses the analyses, optimizations, and back-ends provided by LLVM and composes them together into a complete compiler. This composability is enabled by LLVM's intermediate representation.

Intermediate representation. LLVM-IR is the central abstraction for representing a program in LLVM. It separates out language and architecture specific parts, allowing the core parts of LLVM to be independent and, therefore reusable. Thereby, LLVM-IR decouples the core part of the compiler infrastructure from the language-specific front ends and architecture-specific back-ends. Through this design, LLVM-IR enables reusable analyses, optimizations, and even back-ends, as all of these only depend on a common intermediate representation.

On the top level, a LLVM-IR module is comprised out of functions, global variables, and meta data. Each function is represented as a sequence of basic blocks, which themselves are linear sequences of instructions, implicitly defining the first basic block as the entry node to the function. This way, LLVM-IR represents a function as the linearized form of the corresponding CFG. Important to emphasize about LLVM-IR, is its meta data design. LLVM-IR allows one to attach meta-data references to functions and instructions, which enables developers to attach and persist custom information (e.g., additional data needed for an analysis). From a design perspective, LLVM's meta data format offers free extensibility but also allows developers to break the language independence. However, as described later in detail, it enables us to encode variability information into LLVM-IR.

Intermediate-code generation. To generate LLVM-IR, Clang first preprocesses and parses an input file into an Abstract Syntax Tree (AST). An AST is a tree-structured representation of the source code, where the tree nodes represent specific parts of the program, such as types, declarations, statements, or literals. Important to note, Clang's AST nodes can be extended with additional information, for example, by custom C/C++ attributes that allow us to annotate the source code with custom information. The information attached to AST nodes, can then later be used during the generation of LLVM-IR (e.g., to pass on additional information as meta data to LLVM-IR-based analyses).

The next step after parsing the source code into an AST is lowering it into a LLVM-IR module. This code generation process translates each AST node into a set of equivalent functions, basic blocks, and instructions, representing the translation unit. The transition from AST to LLVM-IR marks an important extension point for our work, as it enables us to extend the generated LLVM-IR with custom information that is only accessible at the source code level.

After the generation process finishes, we have a language-independent representation of our program in LLVM-IR that can now be further processed (e.g., by a static analysis). Language independence is important here, as it not only allows us to target multiple languages but, more importantly, because it allows us to analyze the program in a simpler representation, exonerating the analysis from dealing with the complicated language rules.

Analysis infrastructure. The analysis infrastructure that is part of the LLVM framework is a vital building block in assembling larger analyses and extracting additional information. From a design perspective, each analysis or optimization is

implemented in a separate compiler pass, that is, a modular component that takes LLVM-IR as input and either analyzes it to provide information about the IR or transforms it (e.g., a pass can optimize specific code patterns or add instrumentation code). Through the analysis infrastructure, each pass can requests other passes to gain access to the information they provide. This design enables us to reuse and compose different helper analyses into larger and more complex analyses. Understanding this design concept is important, as we utilize it later on to implement our own analyses, so they can be queried and reused.

2.3.5 Whole-Program Analysis

In Section 2.3.2, we explained inter-procedural analyses and the benefits that can be gained from analyzing the interactions between functions. Ideally, we want to run such analyses over the program as a whole to get the most accurate picture. However, this entails additional difficulties in practice, as in languages, such as C or C++, each source file is compiled on its own into an object file and only later, are these linked together into the final program. Hence, to run LLVM-based whole-program analyses, we need to first produce a combined LLVM-IR file.

One-way to build such a whole-program IR file, is to hook into the build process with Whole Program LLVM (WLLVM[10]) as a compiler wrapper. During build, WLLVM invokes Clang and, in addition to the normal object file, also calls a bitcode compiler that generates an LLVM-IR bitcode file and embeds the location of the bitcode file in the original object file. After the project is compiled, we use WLLVM to generate a whole-program bitcode file by linking the referenced bitcode files together. This way, the generated whole-program contains all LLVM-IR code of the complete program and enables us to run a whole-program analysis.

Another way through which we can automatically merge all LLVM-IR files together and run a whole-program analysis is by running our analyses during the *link-time optimization* stage. Link-Time Optimizations (LTOs), put more generally, are optimizations and analyses that the compiler performs at link-time (i.e., when all processed source files have been linked together). For this purpose, the compiler needs to preserve its internal representation until link-time. To do that, the compiler produces IR files instead of object files and implements linking for its internal representation to merge the IR files together before running LTO passes. So, through LLVM's pass-based analysis infrastructure, we can run our inter-procedural analyses over the whole program, by putting our analyses passes into Clang's LTO stage.

In general, both forms of enabling whole-program analysis are useful. WLLVM for cases where we want to extract analysis information from the program without adapting the compilation process. LTO for cases where want to utilize the results of our analysis during compilation, for example, to feed information to a transformation pass that inserts instrumentations to measure the performance of a program.

10 https://github.com/travitch/whole-program-llvm (Last accessed: May 13, 2023)

Important to note, from a practical perspective neither of the approaches can guarantee that we analyze the whole program as dynamic components are not accessible during link time (e.g., the implementation of functions from dynamic libraries are determined at the start of program). Nonetheless, merging all statically available parts together and running an inter-procedural analysis gives us quite accurate results in practice.

2.3.6 Phasar

In the previous sections, we explained how inter-procedural analyses can analyze whole programs and how LLVM's analysis pipeline works to build reusable analyses. However, in practice LLVM is not well-equipped to run inter-procedural analyses. Due to the runtime constraints an industrial compiler needs to fulfill, most of the provided analyses and data structures, such as points-to analyses and CFGs, are intra procedural only and, therefore, they are not well suited to build inter-procedural analyses. However, the abstraction provided by LLVM, especially, LLVM-IR, are very suitable for program analysis and could enable inter-procedural analysis. To fill this gap, Schubert et al. [208] developed PhASAR[11], a modern static-analysis framework built on top of LLVM that enables users to build inter-procedural analyses that work on LLVM-IR.

PhASAR extends LLVM in multiple ways: It provides abstractions for common analysis problem formats, such as monotonic or IDE (see Section 2.3.3), and corresponding solvers that can solve these general analysis problems. PhASAR provides data structures and helper analyses, such as inter-procedural control-flow graphs, type hierarchy, call graphs, and points-to information. Furthermore, PhASAR also provides additional capabilities for debugging and reasoning about inter-procedural analyses, such as path tracing and graph representations of analysis results.

Most importantly, PhASAR enables us to express IDE problems and solve them automatically through the integrated `IDESolver`. This means, through PhASAR's problem abstraction, we encode the data-flow semantics into flow functions and encode a value computation problem as edge functions. The `IDESolver` then takes the problem description and computes all potential flows and, in addition, solves the value computation problem. How we encode our generic data-flow analysis with PhASAR is presented in Section 3.3 in detail, and the concrete applications for *time* and *space* variability in Chapter 4 and Chapter 5 respectively.

2.3.7 Performance Analysis

With ever more increasing computing needs and the decline of Moore's law [59]—i.e., putting more transistors on a computer chip becomes more and more difficult—hardware-driven performance improvements slow down as semiconductor minia-

11 https://github.com/secure-software-engineering/phasar (Last accessed: April 11, 2023)

turization becomes harder. Leiserson et al. [138] argue, if hardware improvements do not give us the performance we need, software improvements need to drive the performance gains in the future. However, to be able to make software faster, we need to understand its performance characteristics. This means, we need to find *where* in the program time is spent, locate hot-spots, and optimize the code to improve the overall performance. Finding and focusing on performance hot-spots is important, as optimizing program code is time-consuming and only yields worthwhile payoffs when parts in a program are optimized that make up most of the computing time [218, 219].

To detect these hot-spots and infer performance bottlenecks, researchers and practitioners have developed a wide range of methodologies, techniques, and tools [15, 77, 78], summarized under the term performance analysis. Hence, *performance analysis* is the process of evaluating the performance of a software system at different levels of abstraction, utilizing various tools and techniques.

Overall, performance analysis is a difficult and crosscutting endeavor, as it depends on various factors throughout the hardware and software stack, such as CPU type [15, 78], instruction set [15, 78], operating system and kernel [78], support libraries [191, 197], and configurability [83]. Furthermore, even if a software system could be analyzed and optimized, performance is not a constant non-functional property of the system. Over time, as the system evolves, its performance characteristics change and potentially degrade, introducing *performance regressions*. This means, the evolution of the software system can worsen the performance of the software system, for example, by nullifying previous performance optimizations or introducing new bottlenecks. So, performance analysis should be a continuous process that tracks the performance of the system over time, ensuring that it stays within its performance goals [49, 100, 127].

In what follows, we give a brief introduction to performance measurements and profiling in general and, afterwards we contextualize performance profiling to both evolutionary variability (*time*) and configuration variability (*space*).

2.3.7.1 *Performance Measurements*

To extract the necessary information for performance analysis, we need to measure how the software system performs (i.e., we need to observe how the system is executed). Performance measurements are a form of dynamic program analysis that measures various non-functional properties of a software system. Typically, performance measurements are done through a tool (i.e., a performance profiler) that observes the system under measurement (i.e., the program) and the environment it is running in, to extract information about the system's execution behavior. Performance profilers work either with or without modification of the system under measurement. For example, instrumentation-based profilers inject specific measurement code into the system under measurement, where sampling profilers only query the current execution state of the program through a kernel API.

When measuring a software system, we differentiate *what* is measured (i.e., the metric) and *how* it is measured (i.e., the performance measurement technique used to derive the metric). Furthermore, we differentiate performance measurement techniques between *black* box and *white* box and divide them into different strategies [84, 114, 125, 212, 240, 247].

Performance metrics. To understand the performance of a software system, we need to collect data about the execution behavior of the system, aggregate this data into an interpretable form, and then reason about it.

So, in the process, we first need to select what kind of data we want to gather. For example, cache misses, to determine memory bottlenecks, IO operations, to infer network bound issues, or time measurements, to reason about where time is spent in a program. To collect this data, the operating system kernel provides a wide range of different internal counters and inspection capabilities through kernel APIs [78, 81]. For example, cache misses stem from a hardware source (CPU), and IO operations are implemented as a software counter in the kernel, which counts how many operations took place. Furthermore, measurement tools and libraries, such as XRay[12] or likwid[13], provide additional capabilities for analyzing where time is spent inside a program.

After deciding on *what* kind of data should be collected, we need to specify how we retrieve it. For this purpose, we define a *measurement* and *metric*, following the definition from Kounev et al. [126].

Definition 7. A *measurement* is the assignment of values to objects or events by applying a given set of rules or a procedure referred to as a measurement process. [126]

Definition 8. A *metric* is a value derived from some fundamental measurement comprising one or more measures. In the context of benchmarking, metrics are used to characterize different properties of the system under test (SUT), such as performance, reliability, or security. [126]

In more general terms, a measurement is the reading and storing of a value of interest, for example, reading the cache-miss counter (i.e., the generation of a data point from the system under measurement). Then a metric is either the direct value that was captured through the measurement or an aggregated form of one or multiple values. For example, the raw value of the cache miss counter, cache misses per second, or cache misses per user. More high-level metrics are usually derived using some form of statistical process.

Black box vs. white box. We differentiate performance measurement techniques by the information that is available to the analysis and whether the system under measurement needs to be modified. A black-box analysis treats the system under

12 https://llvm.org/docs/XRay.html (Last accessed: June 27, 2023)
13 https://github.com/RRZE-HPC/likwid (Last accessed: June 27, 2023)

measurement as a "black box", meaning, it does not incorporate system-specific information into the analysis or modify the program. Black-box analyses have the advantage that they are often easy to set up. Furthermore, black-box analyses are suitable to be implemented with low-overhead [29, 77, 78], therefore most black-box analyses do not perturb the system under measurement particularly strong. White-box analyses on the other side, incorporate information about the software system, such as its source code, into the analysis and are allowed to also modify the system. For example, a white-box performance profiler can insert custom measurement code during the compilation of the software system, which later gets executed as part of the system under measurement. Important to note, the difference between black-box and white-box profilers is not strict. Many techniques fit into the middle between the two extremes, often referred to as gray-box profilers, where they incorporate only a bit of information about the system or only do slight modifications.

Measurement strategies. We divide performance measurement techniques by *when* and *how* they collect their measurement data into three fundamental strategies [126]. Each strategy collects measurement data differently, however, all strategies are defined around *events* (i.e., they are defined around specific triggers that notify the strategy to do a measurement). For example, the call of a specific function, the reaching of a predefined location (tracepoint) during execution, or a specific amount of time passed. There exists a wide range of different event sources that can be used by all measurement strategies. For example, the operating system offers access to hardware events (e.g., branch-misses), as well as, software events (e.g., page faults). Furthermore, there exist libraries and tools to create static and dynamic tracepoints (i.e., a specified point in the program where) if the execution reaches the tracepoint, an event is triggered. Next, we highlight the three fundamental performance measurement strategies.

Event-driven strategies record their measurements every time one of the specified events occurs (e.g., they count the number of branch-misses). In general, event-driven strategies do not require modifications to count the number of events that occurred. For example, a black-box profiler can count the number of cache misses during a program run without instrumenting the program. However, sometimes a white-box profiler is necessary to instrument the system under measurement to record specific events, such as the amount of requests to an API.

Tracing strategies, compared to event-driven ones, collect more data when an event occurs to provide context about the system. For example, a tracing strategy might additionally store the username for each API request—the API has predefined static tracepoints for profiling—to analyze the load produced by different users.

Sampling strategies are driven by an internal timer that initiates a measurement every x-seconds, often specified in Hz (e.g., common sampling frequencies are 97 Hz or 997 Hz [78]). Compared to the previous two strategies, sampling has the advantage that the produced measurement overhead is based on the sampling frequency instead of the event frequency. However, this also incurs the drawback that some events/program states might be missed. So, in the end, measurement

data derived through a sampling strategy is only a statistical approximation, where overhead and approximation precision form a trade-off [126].

Important to note, the choice between different strategies and tools for performance measurements is always a trade-off between precision, cost, and applicability [78]. As a general rule of thumb, performance profilers that enable precise measurements, such as measuring the execution time of a function in microseconds, require specialized instrumentation code and a more fine-grained control of the system under measurement and, by that, produce more overhead. For comparison, low-overhead strategies, such as sampling, often induce relatively little overhead but also give rough estimates about the time spent within a specific function. Furthermore, as with any observational technique, performance profilers perturb the system under measurement. For example, instrumentation code woven into the system influences the processor cache and, by that, influences the performance of subsequent code. So, depending on which metrics should be measured a measurement technique should be selected that reduces the influence on the system under measurement. However, how exactly the measurement technique influences the system under measurement must always be examined for the specific application scenario and needs to be controlled [77, 78, 126].

Profiling vs. tracing. The term *performance profiling* is often used loosely referring to the general process of measuring and analyzing the performance of a software system. However, a more precise definition of the term differentiates performance profiling from tracing [78, 81, 126]. Performance *profiling* collects the relative amount of time the system under measurement was in a specific state, by storing the number of times an event occurred into the performance profile [126]. For example, the relative amount of time spent executing each function, collected through sampling the call stack at 97 Hz and storing the function name that currently is on top of the call stack, to infer proportionally in which function execution time was spent. Compared to profiling, *tracing* produces an ordered list of the events that occurred during the execution of the system under measurement [126]. This way, we get more information about the system under measurement and enable a more precise evaluation of the measurement data. For example, where in the previous example we only knew the relative amount of time that was spent within a function, tracing can give us the exact amount of time in case we collect timestamps at each function entry and exit event.

However, as differentiating between profiling and tracing is often not important for our work, we use the term performance profiling mostly in the more general sense. In the cases where it is important, we highlight the differences and make them clear to the reader.

2.3.7.2 *Performance Histories*

Similar to functional properties, software evolution also has an impact on non-functional properties, such as performance. Every code change to a software system might also influence its performance characteristics, sometimes in unanticipated ways [34]. For example, an addition, such as a small extra check, to a function that is called very frequently can have drastic impacts on performance. Hence, to prevent undesired side effects, we want to detect performance degradations as early as possible. However, not all changes create an immediate effect, sometimes performance degrades over time through many changes that all have a slight performance impact [50]. Therefore, understanding the performance characteristics of a software system should always require a historical component, where the current performance is analyzed with regard to previous observations. This means, constant performance monitoring of performance critical software systems is essential to detect *performance regressions*.

After an overall performance regression is discovered, we need to track it down to either a specific set of changes that introduced it, or to a specific part of the software system that has become a new hot-spot. To detect the evolved changes, we detect points in the performance history of the system where the performance degrades, referred to as change points (i.e., we search for changes in the performance history where a specified set of performance metrics becomes worse than a given threshold). However, even when such historical performance data does not exist, change points can still be found afterwards. For example, Mühlbauer et al. [163] presented a sampling approach that uses a bisection technique based on gaussian process models, to retroactively determine performance change points. Daly [49] reports that improvements to the performance testing environment of MongoDB allowed them to detect performance regressions earlier in the development process and with more precision, which increased the productivity of developers and lead to a more performant product.

After noticing a large change or detecting a new emerging hot-spot, we can use standard performance profiling techniques to locate hot code parts and improve them. In summary, we notice the importance of detecting performance regressions early, which is made easier through the collection of performance history.

2.3.7.3 *Performance Profiling of Configurable Software Systems*

Each variant of a configurable software system can potentially have different performance characteristics. For example, Jamshidi and Casale [99] report that the throughput of Apache Storm's worst configuration is *480 times slower* than the best configuration [99, 130]. That is, setting Apache Storm's configuration options to the wrong values can lead a significant performance loss. A recent empirical study by Kaltenecker et al. [114] found that the majority of performance changes affects only a few variants. So, to get an overall picture that encompasses the performance of all configurations, we would need to profile all variants, or at least a representative

sample. Another peculiarity about the performance analysis of configurable software system is that from a usual performance trace or sample (previously explained in Section 2.3.7.1), we do not know how the measured performance data is related to the different features. This means, we have no direct way of mapping the measured performance metric to a feature's implementation and, by that, reason about the influence a feature has on performance characteristics of the measured variant. Summed up, the performance analysis of configurable software systems poses two additional problems (1) "How do we describe the performance of the overall system?" (overall performance) and (2) "How can we relate performance characteristics to individual features and interactions thereof?" (feature relation).

Overall performance. To better represent the overall performance of a configurable software system and address problem (1), Siegmund et al. [212] introduced *performance influence models*. Performance-influence models are useful and a establish way to describe the performance influences of configurable software system, which is why, we later also employ them in the evaluation of our configuration-aware profilers in Chapter 5. Performance-influence model represent a possible solution for detecting performance issues of configurable software systems [212, 213]. In addition to quantify the influence of configuration options on performance, they can be used also to find unexpected interactions among options [216].

In essence, performance-influence models are typically a prediction function whose terms are the configuration options and interactions thereof together with a weight that depicts the option's performance influence. Take, for example, configurable software system with two configuration options: *compression* (📦) and *encryption* (🔒). After measuring all configurations, we create a performance model (Π) of the example system, where c refers to a configuration (i.e., a set of selected values for all configuration options).

$$\Pi(📦, 🔒) = 10s + 3s \cdot 📦 + 5s \cdot 🔒 + 1.3s \cdot 📦 \cdot 🔒$$

Hence, the configuration where both configuration options are enabled would take 19.3s, the one where 🔒 is disabled only 13s.

Similar to other program analyses, feature interactions make performance prediction and analysis difficult because even a small and alleged change in the set of selected features can have a profound effect on the performance characteristics of the whole software system. Remember the example from before, where enabling a feature can influence the cache state and, by that, alter the performance of another feature. The central point is that cross-configuration performance predictions might be wrong. A model might predict performance quite well on average but still produce wrong values of some configurations that contain unseen performance feature interactions. Hence, similar to general feature interactions, profiling the NFPs of a configurable software system is subject to the same trade-offs as regular program analysis where analyzing more variants produces more precise results but also incurs more cost. However, in practice sampling strategies, such as distance-based

sampling [113], work well in selecting a reasonably small set of variants for building performance influence models.

Feature relation. The second problem (2) one encounters when analyzing the performance of a configurable software system is that off-the-shelf performance profilers cannot relate their measurements to features. However, performance influence models alone do not enable developers to attribute the learned performance influences to source code. As a first step, Weber et al. [247] propose a joined approach that combines white-box performance measurements, taken at function-level granularity, with the learning of performance influence models. They demonstrate that performance influence models at a method level can accurately pinpoint the influences different configuration options have on a function's execution time. Furthermore, to address the problem of attributing performance measurements to the feature code, Velez et al. [240] developed ConfigCrusher, a white-box analysis tools for collecting feature-specific time measurements. They later extended their work to Comprex, by adding a dynamic-analysis to locate feature-specific code with more precision [241]. Later in Chapter 5, we generalize this work by providing a flexible way to make performance profilers feature aware.

Important to note, challenges introduced by evolution (*time*) (see Section 2.3.7.2) and configurability (*space*) are not independent [233]. An empirical study exploring 190 releases from 12 real-world configurable software system, done by Kaltenecker et al. [114], found performance changes in nearly all releases. Furthermore, their study shows that the majority of performance changes often affects only a small subset of the configuration space, which means they could easily be missed, even by a good sampling strategy. This indicates that performance analyses should always incorporate the evolution and the configurability of the system under measurement. In their work, Mühlbauer et al. [164] demonstrate the feasibility of such a combined approach that can detect configuration-dependent performance changes.

In summary, analyzing the performance characteristics of configurable software systems poses additional challenges, due to the large amount of variants and potential interactions between features. Hence, special modeling approaches, such as performance-influence models [212], or adapted profiling techniques, such as ConfigCrusher [240] or Comprex [241], are necessary to reason about the overall performance characteristics of a configurable software system.

3

A Uniform Code-Region Abstraction

This chapter presents the overarching idea behind this thesis and introduces its centerpiece, the uniform abstraction of code regions. The goal of the uniform abstraction of code regions is to bridge the gap between different types of variability information and state-of-the-art program analysis, making both variability information and analyses reusable and freely combinable. To achieve this, we decouple variability through the abstraction from the concrete program analysis and its semantics, and separate the different parts into combinable components. The design of variability information and program analysis components follows the product-line idea, where through a configuration process analyses can be automatically combined with different types of variability information.

In what follows, we first lay out how building more reusable variability-focused analyses could improve variability research. Then, we give an in-depth explanation how our code-region abstraction is comprised and how it can be used to make variability-focused analyses more general and reusable. Therefore, we demonstrate how we lift a static as well as a dynamic program analysis to code regions, to compute multiple kinds of interactions between different types of variability. After the theoretical foundations, we explain how we built our code-region abstraction into the compiler framework LLVM to apply variability-focused program analyses to real-world configurable software systems. Then, we describe two research problems that we could address through our analyses and how we benefitted from the reusable analysis structure that we built through our code-region abstraction. Last, we give an outlook how our abstraction could be generalized to more use cases, making the abstraction even more uniform.

3.1 Introduction

Both, *time* and *space* variability should be analyzed in unison to better reason about the evolution of real-world configurable software systems [233]. Up to now, research has mostly focused either on one of the two types of variability or only considered both types of variability together in an ad-hoc way through black-box analyses. Initial work in this direction using a black-box analysis already produced interesting

results. For example, the work from Mühlbauer et al. [164] showed how to identify configuration-specific performance changes in the history of configurable software systems. Furthermore, works from Velez et al. [240] and Weber et al. [247], already show that applying white-box analyses to specific types of variability, *space* in their case, leads to interesting new insights (e.g., that models built through white-box performance analyses can identify performance-relevant methods whose performance depends on configuration options). From other domains we also know that employing white-box analyses can lead to interesting insights. For example, detecting SQL injections vulnerabilities through white-box analysis, in the security domain [69]. Nonetheless, developing such analyses is time-consuming and seldomly reusable, which incurs higher research cost. What is currently missing in the variability domain, is the widespread use of white-box analysis that target the evolution of configurable software system. Hence, one would expect that similar insights could be gained by applying white-box analysis on *time* and *space* variability. However, this research direction is currently not often explored due to the challenges that white-box analysis entails [208]. We identified two major challenges that currently hinder a wider use of white-box analyses: (1) It is difficult to access variability information in white-box analyses, as it requires a precise mapping of the variability information onto the source code; (2) Building and employing precise white-box analyses that are capable of analyzing real-world software projects is difficult and time-consuming. To address these two challenges, we devised a uniform abstraction that maps variability information into an analysis framework, and built a universal and reusable white-box analysis that can analyze different types of variability.

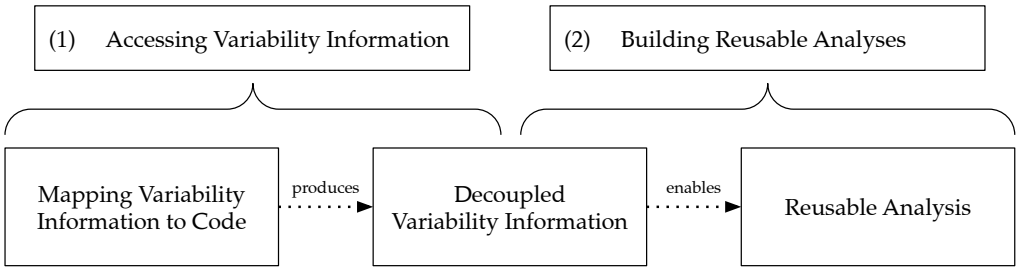


Figure 3.1: Overview of the two major challenges and how they are divided up into three concrete problems that we address in this chapter.

Our goal is to simplify the use of precise white-box analyses to analyze both *time* and *space* variability, each for themselves or even together. However, to enable precise and reusable white-box analyses, we need to address three concrete problems (depicted in Figure 3.1). First, we need to decouple variability information and its representation from the concrete analysis, making it easier to develop new analyses or reuse existing ones to investigate variability. In addition, the decoupling also needs to ensure that both types of variability can be represented simultaneously alongside each other to enable analyses to analyze both in unison. Second, we need a precise and automated way to map the externally provided variability information

into a representation that is usable by an analysis. Third, we need to enable the building of reusable analyses that can reason about different types of variability only through a simple but uniform API. By providing a solution to all of these three problems, we facilitate the easier use of precise white-box analysis in the domain of variability. In what follows, we explore these problems in more detail and explain how we address them.

Uniformly representing variability. To enable the building of reusable and flexible analyses, we need a way to decouple the *what*, the specific variability information we want to analyze, from the *how*, the concrete analysis semantics. This decoupling is necessary because, as soon as the analysis semantics are intermixed with the specific information, the analysis is no longer reusable. This means, we need to introduce an abstraction that presents the variability information uniformly to the analysis. In Section 3.2, we present such a uniform abstraction by introducing the concept of *code regions* and explain how domain-specific commonalities can be used to unify both variability dimensions. Furthermore, in Section 3.4 we explain the technical details that were necessary to implement the abstraction within LLVM, enabling us to target real-world software projects with our analyses.

Tracing variability information. To analyze variability through a uniform abstraction, we need concrete ways to extract and unify the different types of variability information. This requires two important steps: First, having a precise way of mapping the variability information into the intermediate representation that is used by the analysis. Conceptually, this step is simple. We need to attach the additional variability information to the intermediate representation. However, from a practical point of view, this step is difficult. It requires that we modify the transformation process from source code to intermediate representation and precisely track which parts of the intermediate representation are related to variability information. Second, as variability information can have structural properties (e.g., this block of code is controlled by a specific configuration option, that can be relevant for an analysis, we want to preserve them). For this purpose, we model structurally connected parts of the code that carry the same information as code regions (see Section 3.2). How both of these steps can be realized in practice is shown in Section 3.4.4.

Reusable analysis. To get the precise analyses results, a white-box analysis should often be some or all of the following: context sensitive, field sensitive, inter-procedural, alias aware, and flow sensitive. However, building such a precise inter-procedural static analysis that can analyze real-world software projects is a difficult and time-consuming task, often too expensive to build for answering a single research question. The extensive amount of work becomes only viable if the analysis itself is so generic that it can be employed in multiple research scenarios. To achieve this for variability-focused analyses, we decouple the concrete variability information for the actual analysis through our uniform abstraction and implement a reusable data-flow analysis based on it. More details about the practical analysis

interface to the uniform abstraction are presented in Section 3.4.5, and how we use this interface to lift data-flow analysis to our uniform abstraction in Section 3.3.

Our goal is to devise the foundation for reusable variability-focused white-box program analysis. We achieve this by our uniform *code region* abstraction that integrates both types of variability into an intermediate representation suited for inter-procedural white-box program analysis but, at the same time, decouples the concrete variability information from the program analysis to enable reusability.

3.2 Code-Region Abstraction

The first key challenge is to introduce an abstraction to unify different types of variability. What we currently observe is that approaches analyzing variability do this in an ad-hoc manner, where they build custom variability mappings for their type of variability together with a custom program analysis. This way, they spend a lot of time and effort in developing their approach and often cannot employ state-of-the-art program analysis techniques to tune and optimize the precision of the analysis. What was missed up to now is that treating, modeling, and analyzing different types of variability as a single domain would enable us to build more reusable program analyses that target different types of variability. The idea is that, by exploiting domain commonalities, we can build more flexible and reusable analyses, and overall, reduce the burden of analyzing different types of variability.

Going back to the ideas behind domain-driven design [62], we notice that what is currently missing in the design of many analysis tools is a *domain layer* that decouples analyses (*infrastructure*) from the concrete application scenario (e.g., building an analysis to detect feature interactions [12, 125]). The idea behind domain-driven design is to isolate the different responsibilities by separating them into different layers, where each layer only interacts with the one above or below it. The infrastructure layer encapsulates all the technical capabilities that are needed; In our case it should contain the existing program analyses. The domain layer models the business logic (i.e., the domain specific part). This is the layer that is currently missing in analyses and which we want to add, to enable more reusable program analyses. The application layer then only defines what the software should do, which means in our case *what* variabilities should be analyzed and *how* should they be analyzed (with which analysis). The original definition of a domain layer does at first not apply cleanly to our use case, as we do not want to treat the domain layer as just business logic. However, if we combine the idea of a domain-specific separation layer, between the infrastructure and the application, together with domain engineering [7], we get a product line for variability-focused program analysis. So, in our case, the domain layer provides a uniform variability abstraction, that is, it provides uniform access to different types of variability, and domain-specific adapters to program analyses from the infrastructure layer. This way, through the application layer—which can be practically seen as application engineering

from product-line engineering [7]—we can analyze different types of variability by selecting domain-specific components from the domain layer.

So, what we need is a domain layer with a uniform abstraction, fulfilling these two complementary roles: (1) Presenting different types of variability information in a uniform way; (2) Decoupling the variability information from the concrete analyses.

We devise such an abstraction with *code regions*. In a nutshell, code regions provide a uniform way of representing domain-specific variability information, together with structural information, as an abstract interface to program analyses.

In our design (depicted in Figure 3.2), we treat variability information as domain specific and isolate this information into the domain layer. This way, we conceptually gain a separation from program analyses (the infrastructure layer) and the concrete application scenario (the application layer) (e.g., detecting feature interactions [12, 125]). To uniformly separate the information in the domain layer from the other two layers we use code regions (i.e., connected pieces of code that carry the same domain-specific information). On the one side, code regions unify different kinds of variability by aggregating domain-specific variability information. On the other side, code regions define an interface for program analyses through which we can connect existing program analyses to the domain layer. This way, an existing program analysis that is lifted to code regions can automatically be applied to different types of variability. To visualize this, we depict code regions in Figure 3.2 as a connecting piece between variability as well as program analyses. Therefore, code regions act as link between a concrete analysis and domain-specific variability information. So that in the end, using a program analysis on variability (i.e., asking a question about variability that is solved by a program analysis) translates to, selecting the right components from the domain layer (code-region-based analysis + variability information).

Figure 3.2 depicts two example applications that we built using the domain-driven layered design around code regions. Application scenario 1 and 2 can both select to their needs specific program analyses and variabilities from the domain layer and piece them together to solve their problem, since code regions make analyses and variability freely combinable. The analysis framework then automatically integrates them to solve our concrete problems. Details to both of these applications that we built are later described in Section 3.5.

To build up the information in the domain layer, we introduce domain-specific variability information, on a fine-grained level, as tags to the instructions of a program. Based on this tagging information, we then employ a detection approach that groups instructions with the same tags into code regions (i.e., a code region is a connected part of the program's code that represent the same domain-specific information). Now, to decouple program analysis from the concrete variability information, we introduce an analysis interface for code regions, so that an analysis can access only information offered by a code region and does not have direct access to the specific variability information. The crucial part here is to decouple the analysis semantics from the specific variability information. How we split the

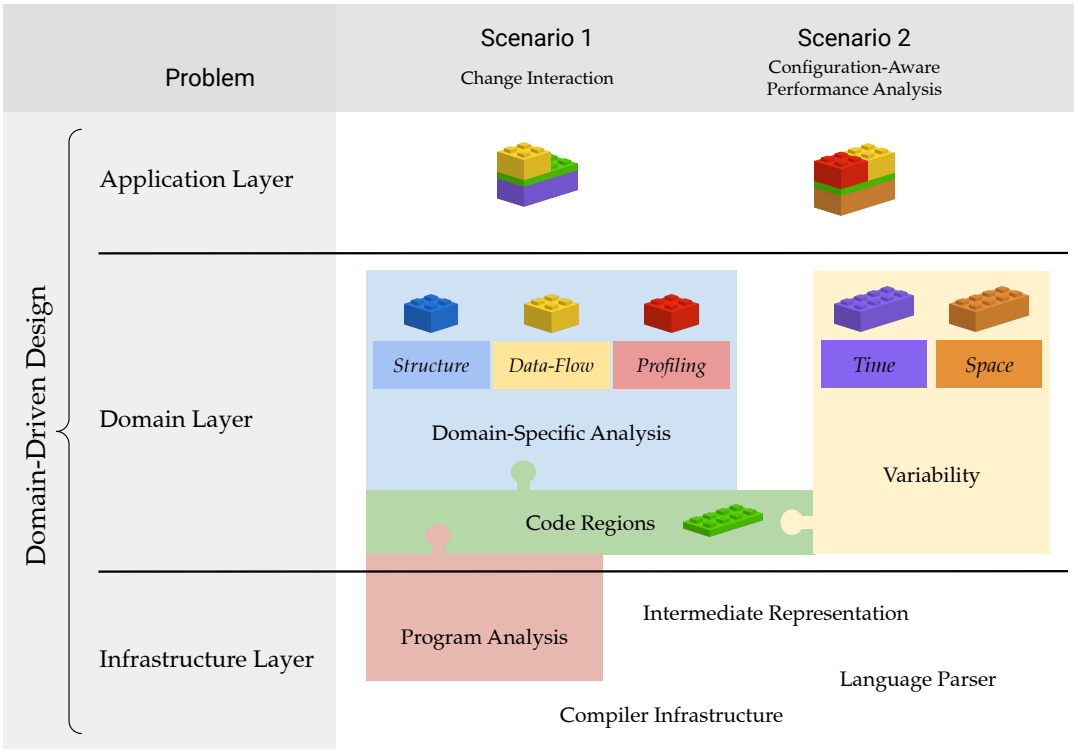


Figure 3.2: Design overview of the uniform code-region abstraction. The layered architecture, following domain-driven design principles, isolates the domain-specific variability information through the code regions into the domain layer. This separates variability from technical details, such as program analyses or language parsers, and from the concrete application scenario that should be addressed.

analysis semantics into analysis specific and variability specific, and by that, lift program analysis to code regions is presented in detail in Section 3.3.

In what follows, we define our uniform code-region abstraction, to which we refer to as just code regions from now on. First, we introduce how we represent a program and how the core code-region abstraction is connected to the program representation. Then, we present a uniform technique that computes code regions from domain-specific tags, adding structural information to the code regions in the process. Afterwards, we extend our view on code regions by the program history and describe the analysis interface, which we later use for developing code-region-based static and dynamic analyses.

3.2.1 Program Representation

As introduced in Section 2.3.1, a program p is a collection of functions $f_1, \ldots, f_n \in p$, where each function is represented as a CFG. To model code regions more precisely than at the level of basic blocks, we now transform this basic-block-based graph representation into an instruction-based graph representation. This way, we can

later attribute variability at instruction granularity, which can be important for very fine-grained types of variability, such as revisions, which can impact only a few or even a single instruction.

So, we now represent each function f as a triple that consists of each a single entry and exit instruction (referred to as EntryI_f and ExitI_f) as well as a set of instructions that make up the function, which can be accessed through $\mathsf{insts}(f)$. We model the ordering of instructions with the function $\mathsf{next(i)}$, which maps an instruction i to the set of successor instructions, and $\mathsf{pred(i)}$, which maps an instruction i to the set of predecessor instructions. This allows us to abstract from the conceptual program representation and execution order as well as from the concrete program representation. For example, our implementation of next for LLVM-IR maps the control-flow changing instruction at the end of a basic block to the first instruction of the basic block's successors. For all other instructions, next returns the following instruction in the basic block. Important to note, we are not interested in any specific instruction sequence as long as the ordering is correct and, therefore, our next function does not model these relationships and only assume a sequential execution order. However, one could adapt next to precisely model ordering semantics in cases where it is important and still use our abstraction. By defining next, we can then use a standard way to represent a function f as a graph composed out of instructions.

Definition 9. An $\mathsf{instGraph}(f)$ of a function f is a tuple of vertices and edges, where each vertex is an instruction of f, and where we draw an edge (i, i') between two instructions i and i' if i' can be executed next after i.

$$\mathsf{instGraph}(f) = \Big(\mathsf{insts}(f), \{\ (i,\, i')\ \mid\ i,\, i' \in \mathsf{insts}(f)\ \ \wedge\ \ i' \in \mathsf{next}(i)\ \} \Big)$$

Based on the program representation of $\mathsf{instGraphs}$, we define in the following our uniform code-region abstraction that combines the program representation with variability information.

3.2.2 Variability Tagging

In addition to the program representation, we need to define the mapping function that introduces the variability-specific information. For this purpose, we require one commonality from all supported types of variability: to be analyzable, all types of variability in the end need be attributable to a valid code fragment during compilation. This is a restriction that is technically necessary and fulfilled by many variabilities, at least partially. For example, the only limit we discovered up until now is for compile-time variability, where alternatives cannot be present in the program at the same time as the analyses require a valid program (e.g., an `#ifdef` with mutually exclusive then and else parts). However, this can theoretical be overcome by compiling and running the analysis multiple times and did not cause any issues for us in practice. To use our uniform abstraction, there are two specific concretization points that need to be defined to apply the uniform code-region

abstraction to a specific instance of variability: (1) a set of domain-specific tags that represent the variability information (2) a tagging function that relates the concrete variability information to instructions.

So, for each program p, we define a set of tags $\mathcal{T}$ that models the variability specific information and a mapping function $\mathsf{tags(i)}$ that tags each instruction i with the variability specific information that relates to it. For this purpose, we define $\mathcal{T}$ as follows:

Definition 10. Let $\mathcal{T}$ be a generic set of domain-specific tags $t \in \mathcal{T}$ that carry variability information.

Important to note, from the point of view of the uniform code-region abstraction, $\mathcal{T}$ is only seen as a set of tags. In a concrete instance where the abstraction is applied, the tags are then mapped to variability information. For example, in the case of *time* variability, we can define the set of tags to be the set of revisions and then tag each instruction with the revision that introduced the instruction. However, the uniform abstraction does not consider the specific meaning or semantics of the tags, just whether they are present or not. Next, we define the abstract mapping function $\mathsf{tags(i)}$ that relates instructions to the specific set of tags. It is important to note that the issues of determining how a specific tag is related to (@) an instruction i is defined by the concrete instance.

Definition 11. Let $\mathsf{tags}(i)$ be a function that maps an instruction i to the set of domain-specific variability tags that relate (@) to the instruction i.

$$\mathsf{tags}(i) = \{\, t \mid t \in \mathcal{T} \ \wedge \ t \mathbin{@} i \,\}$$

After conceptually defining the two concretization points $\mathcal{T}$ and tags, we can now define our uniform code-region abstraction around them. This way, a concrete instance only needs to supply the two concretization points to automatically reuse everything built around the uniform abstraction of code regions.

3.2.3 Aggregation Into Regions

Mapping variability information into an analyzable representation, by tagging the instructions of a program with the various kinds of variability information, is an important first step. However, this is not enough to analyze all facets of variability, as it does not explicitly represent the structural information of specific types of variability. For example, some parts of a program are only executable if a specific configuration option is turned on, that is, the configuration option controls whether a group of adjacent instructions is executed. To automatically expose this information and enable program analyses to capture and reason about it, we need to compute structural information by grouping together instructions that are adjacent and carry the same variability information.

We achieve this grouping of instructions that are associated with the same variability tags through graph clustering. From the graph-based program representation

presented in Section 3.2.1, we see that groups of instructions that have the same variability tags form clusters in the graph (i.e., adjacent instructions with the same variability tags induce a sub-graph in a function's instGraph). We exploit this fact and define the core of our uniform abstraction, the code region, as a graph together with a set of tags that apply to all instructions in the graph. Through this,

Definition 12. A *code region* is a tuple consisting of a graph, which is comprised out of instructions as vertices and edges between instructions, and a set of tags (Tags) that are attached to all instructions in the code region. Where $\mathcal{V}$ is the set of all instructions, $\mathcal{E}$ is the set of all edges between instructions, and $\mathbb{P}$ denotes the powerset. We can therefore define the set of all code regions as follows.

$$\mathcal{CR} \subseteq \big(\mathbb{P}(\mathcal{V}), \mathbb{P}(\mathcal{E})\big) \times \mathbb{P}(\mathcal{T})$$

This way, the problem of detecting a code region is equal to finding sub-graphs in the instGraph of a function, where all instructions in the graph carry the same tags. In our case, the problem is simplified by the fact that we can remove all vertices that do not carry these tags and related edges from the graph and the leftover components are the code regions. To access the information in a code region, we introduce mapping functions for each item of the code region.

Definition 13. Let vertices, edges, and tags_{CR} be functions that map a code region to the set of elements (instructions, edges, tags), that are part of the region.

$$\begin{aligned} \text{vertices} &: \mathcal{CR} \to \{\, \mathcal{I} \,\} \\ \text{edges} &: \mathcal{CR} \to \{\, (\mathcal{I}, \mathcal{I}) \,\} \\ \text{tags}_{\text{CR}} &: \mathcal{CR} \to \{\, \mathcal{T} \,\} \end{aligned}$$

With $\text{computeCR}_{\text{Tag}}(f, t)$, we compute all code regions for tag t in function f, by removing all instructions (and related edges) that are not tagged by t from f's instGraph. The leftover components of the graph now correspond to the code regions, as they are connected groups of instructions that are all tagged with t. Computing the graph's components is a standard graph problem, which we solve by the well-known standard algorithm by Hopcroft and Tarjan [96].

Definition 14. The function $\text{computeCR}_{\text{Tag}}(f, t)$ computes the set of code regions associated with the tag $t \in \mathcal{T}$ by removing ($\ominus$) all vertices from the instGraph of $f \in p$ that are not tagged by t as well as the related edges.

$$\begin{aligned} \text{computeCR}_{\text{Tag}}(f, t) = \\ \left\{ (g, \{t\}) \;\middle|\; g \in \text{components}\Big(\text{instGraph}(f) \ominus \right. \\ \left. \Big\{ i \mid i \in \text{insts}(f) \;\wedge\; t \notin \text{tags}(i) \Big\} \Big) \right\} \end{aligned}$$

Important to note, this is only the theoretical approach to detect code regions, in practice we implemented an algorithm that does not need to modify the graph. We present how this algorithm works in detail in Section 3.4.4.

Now, to compute all code regions for a function f, we first compute all code regions for each tag and then merge the resulting sets together.

Definition 15. The function $\mathsf{computeCR}_{\mathsf{Function}}(f)$ computes all code regions for a given function f, by computing the code regions for each tag $t \in \mathcal{T}$.

$$\mathsf{computeCR}_{\mathsf{Function}}(f) = \bigcup_{t \in \mathcal{T}} \mathsf{computeCR}_{\mathsf{Tag}}(f, t)$$

Figure 3.3 shows an example for the code region detection process, where we first tag each instruction of a function with its corresponding tags. Then, for each tag, we compute sub-graphs that carry the same tag and construct a code region for each sub-graph by attaching the tag information. In the end, as depicted in Figure 3.3, we know which parts of a function are related to the variability information represented in the tags.

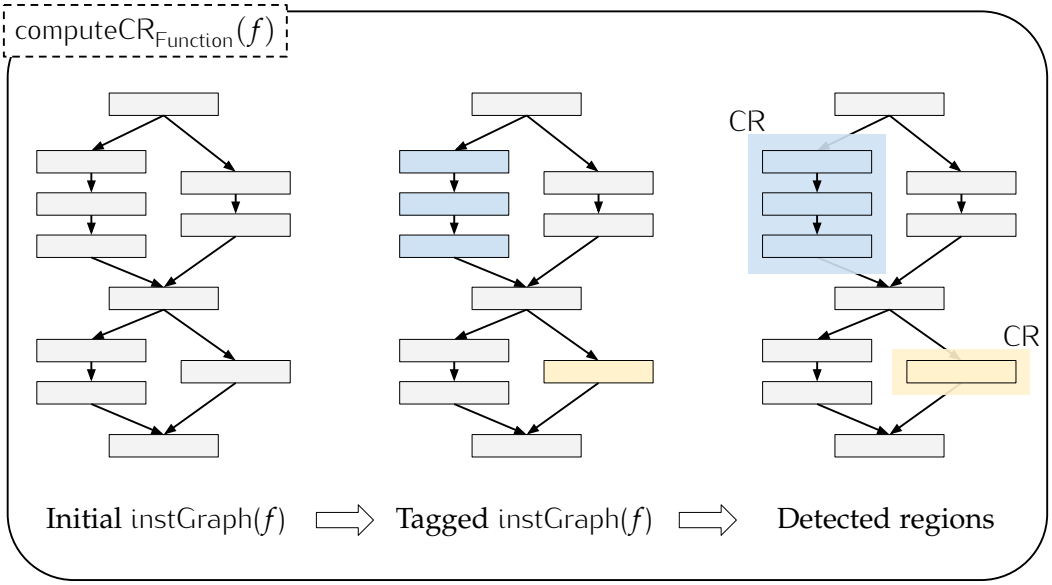


Figure 3.3: To compute the code regions for a function, first we annotate each instruction (depicted as blocks) with its corresponding tags through tags (indicated by color). Next, $\mathsf{computeCR}_{\mathsf{Function}}(f)$ computes all connected components that carry the same tags, one tag at a time, and groups them into code regions.

Furthermore, as an optimization step to reduce the amount of code regions, we can merge together all regions that span the same sub-graph by merging their tag sets together. Important to note, finding code regions covering the same sub-graph and merging them is not equal to the $\mathcal{NP}$-complete problem of finding isomorphic graphs [40], as in our case we do not need to match vertices. We know the vertices involved in both regions and as the edges are code region independently defined by

next, determining if two regions span the same sub graph can be done by comparing their vertices (i.e., *iff* two regions have exactly the same set of vertices) they describe the same sub graph.

Based on the definitions above, we can now compute all code regions for a given program $\mathcal{P}$ with computeCR, by first computing the code regions for each function and then merging them together.

Definition 16. With $\mathsf{computeCR}_{\mathsf{Program}}(p)$, we compute all code regions for a given program p, by computing the code regions for each function $f \in p$.

$$\mathsf{computeCR}_{\mathsf{Program}}(p) = \bigcup_{f \in p} \mathsf{computeCR}_{\mathsf{Function}}(f)$$

Through the generic detection approach presented in this section, we can automatically compute code regions that also represent structural information, in addition to the variability information that is tagged onto the instructions. So, code regions present a uniform abstraction that enables us to model different types of variability uniformly on a given intermediate representation (e.g., LLVM-IR), by attaching variability in a uniform and structured way to the program representation.

3.2.4 Program History

Up to now, we treated a program p as a single and fixed entity, however, this view leaves out the history through which the program was developed. As we mention in Section 2.2, programs evolve and are produced over time through a sequence of changes, where each change produces a new *revision* of the program (see Section 2.2.1). So, referring to a program as just p is imprecise, as we do not specify which revision of the program we are analyzing.

To address this issue, we define the relation $\mathcal{R}_{\mathcal{P}}$ that associates a revision identifier to each program revision.

Definition 17. Let $\mathcal{R}_{\mathcal{P}}$ be the relation that defines all program revisions of a software project where $p_1, p_2, \ldots, p_n \in \mathcal{P}$ is the set of programs and $r_1, r_2, \ldots, r_n \in \mathcal{R}$ is the set of revisions (e.g., the commits contained in the software project's repository).

$$\mathcal{R}_{\mathcal{P}} : \mathcal{R} \to \mathcal{P}$$
$$\mathcal{R}_{\mathcal{P}} = (r_1, p_1), (r_2, p_2), \ \ldots \ , (r_n, p_n)$$

As a shorthand, we introduce a revision specifier p^{rev} for a program p that specifies the revision of the program.

Definition 18. Let p^{rev} refer to the program at revision `rev`.

$$p^{\mathsf{rev}} \equiv \mathcal{R}_{\mathcal{P}}(\mathtt{rev})$$

Based on the revision relation, we can compute the code regions for multiple program revisions through $\mathsf{computeCR}(\mathtt{revs})$.

Definition 19. With computeCR(revs), we compute all code regions for the specified revisions in revs $\subseteq \mathcal{R}$, by computing the revision-specific code regions for each program revision $p^{\texttt{rev}}$ with rev $\in$ revs.

$$\text{computeCR}(\texttt{revs}) = \bigcup_{\texttt{rev} \in \texttt{revs}} \text{computeCR}_{\text{Program}}(p^{\texttt{rev}})$$

Thereby, a code region $c \in \mathcal{CR}$ detected in program $p^{\texttt{f4}}$ is revision specific to program p at revision f4.

Important to note, the revision we attach to a program p to represent it at a specific point in its development time comes from the *time* dimension. However, it is not the same as modeling revisions directly as variability with code regions. Whereas the revisions attached through the revision specifier states what "revision" of a program we analyse, revisions modeled through code regions would indicate what change introduced a specific piece of code (i.e., which parts of the code base were added in which revision).

By explicitly stating the exact revision of the program we analyze through the revision specifier, we can identify code regions and also analyze results more precisely. This precision then enables us to analyze multiple different revisions of a program and relate the results to each other. For clarity purposes, we explicitly state revision for programs or code regions only in the cases where they are needed and leave them off otherwise.

3.2.5 Code Regions as a Uniform Analysis Interface

To build static and dynamic program analyses that are reusable and, thereby, capable of analyzing different types of variability, we need to connect variability information with the analysis conceptually. In our case, this means we need to connect our uniform abstraction of code regions with the analysis (i.e., we need to enable the analysis to access variability information in a uniform way). Depending on the analysis, we identified two key information requirements: (1) uniform access to the variability information and (2) uniform access to the structural information.

Encapsulating variability. To enable uniform access to the variability information contained in the code regions, we introduce *code-region taints* as an additional separation that connects code regions and the analysis' semantics. Code-region taints decouple the variability focused part of the analysis' semantics and allow an analysis implementer to specify the rest in a generic way.

Simply put, a code-region taint can be seen as a unique identifier for a code region that is used directly in the analysis state to refer to the code region. The code-region taint encapsulates the variability-specific part of the analysis by providing additional operations that are needed to express the analysis semantics (e.g., expressing equality between two code regions). These operations are implemented through the code region interface. Important to note, this does not break the abstraction between

analysis and variability information, as the code-region taint implementation can only make use of the information provided by the uniform code region and cannot access the concrete variability information. This means, the implementation can, for example, check if the code region referred to by one code-region taint has the same tags as the one referred to by the other code-region taint, but not what specific tags are attached.

For example, sets of code-region taints can be used as values in the analysis lattice to represent the variability information that holds at a specific point in the analysis. The analysis implementer expresses the semantics in a generic way, for example, that two lattices should be merged by the join function through a set union. To compute this set union now requires a definition for equality on the lattice values. Comparing generic lattices values, such as *bottom*, *top*, or sets can be handled in a generic way. The equality check between two code-region taints is specified by the code-region taint implementation and not by the analysis itself. This way, the variability-specific part is fully encapsulated in the code-region taint equality. Understanding this separation is crucial: where the *analysis-specific how* (i.e., the join-semantics are expressed by the set union) the *variability-specific how* is defined through the code-region taint equality.

Structural boundary. With regards to structural information, we identified two important information requirements. First, analyses need a way to determine if parts of the analysis representation are contained within a given code region. For that, we defined a query function contains that checks if a given instruction is part of a code region.

Definition 20. The function $\mathsf{contains}(i, r)$ checks if a given instruction i is part of the code region r.

$$\mathsf{contains}(i, r) = \; i \in \mathsf{vertices}(r)$$

Second, we found that analyses, especially dynamic ones that need to insert code around a code region, require access to the entry and exit instructions of a code region. We query this information with the functions entries and exits.

Definition 21. The function $\mathsf{entries}(r)$ computes the set of instructions through which the region r can be entered.

$$\mathsf{entries}(r) = \left\{ i \mid \neg\, \mathsf{contains}\big(\, \mathsf{pred}(i),\, r \,\big) \;\wedge\; i \in \mathsf{vertices}(r) \right\}$$

Definition 22. The function $\mathsf{exits}(r)$ computes the set of instructions through which the region r can be left.

$$\mathsf{exits}(r) = \left\{ i \mid \neg\, \mathsf{contains}\big(\, \mathsf{next}(i),\, r \,\big) \;\wedge\; i \in \mathsf{vertices}(r) \right\}$$

Through the code-region analysis interface presented above, we are now able to conceptualize static and dynamic program analyses that, from the perspective of

the analysis, are only defined against our uniform abstraction of code regions. In the following Section 3.3, we describe in detail how we lift a data-flow analysis through this interface to code regions, to make the analysis uniformly applicable to different types of variability. Furthermore, we highlight important technical parts of the code-region analysis interface in Section 3.4.5.

3.3 Lifting Program Analysis to Code Regions

Our overall goal is to better connect variability information with program analysis and ease the process of building and analyzing different kinds of variability. In the previous section, we showed how our uniform abstraction of code regions enables us to generalize over different types of variability. Building upon our code-region abstraction and the analysis interface introduce in Section 3.2.5, we demonstrate in the following how we make program analyses domain-specific by lifting them to code regions, so that the in the end, an analysis can be applied automatically to different types of variability.

As the general field of program analysis is quite large and diverse, we focus our demonstration on the family of interaction analyses. From previous research we know that determining and analyzing different types of interactions between features—analyzing interactions in the *space* dimension—lead to valuable insights [6, 12, 125]. However, this has not been applied in the same granularity to other types for variability, such as *time* variability. Therefore, we expect that lifting interaction analyses to code regions and then applying it to various kinds of variability will lead to new insights.

Another reason why we select the family of interaction analyses is that they offer a diverse set of application possibilities that can all be generalized under the concept of an interaction. This generality offers us, additionally to the reuse between different types of variability, the applicability to a diverse set of application scenarios. Take for example, feature interactions, as introduced in Section 2.1.5. Feature interactions can be caused through multiple means, for example, through control or data-flow interactions, or indirectly influencing the non-functional-properties of other features.

From an abstract point of view, interactions are composed out of two components: *what* is interacting and *how* is it interacting. In our context, we are interested in how different types of variability interact, so *what* can refer to: configuration options, variants, revisions, or versions. To simplify the handling of all these different variability concepts, we condense them into code regions. This way, from an analysis point of view, interactions are defined between code regions. What is left now, is to define the *how*, which is specified and computed by a program analysis.

In general, we identified three important groups of interaction mechanisms *how* code regions can interact that have been investigated in the past: (1) *structural* interactions (i.e., interactions where variability related code structurally overlaps); (2) *control-flow* or *data-flow* interactions (i.e., interactions where variability related code parts affect each other through control-flow or data-flow); (3) *non-functional-*

property (NFP) interactions (i.e., indirect interactions between variability that are observed through their effect on non-functional properties). Finding these different interaction types requires different program analyses. Where *structural* and *data-flow* interactions can be analyzed through static analyses, *NFP* interactions often require dynamic analyses.

However, finding the same type of interaction for different types of variability does not require a different analysis, because the underlying analysis problem is still the same. By design, if we implement an analysis against our uniform code-region analysis interface, we separate the analysis semantics—interaction semantics—from the concrete type of variability, and can automatically apply the analysis to various kinds of variability. This means that, put together, code regions enable a uniform way to compute different types of interactions between different types of variability by providing a link between the *what* and the *how*.

By separating the specific interaction semantics into an interaction relation ($\blacklozenge$), we generalize interactions to code regions through the concept of a *code-region interaction*. So, where the code region unifies and decouples the variability from the analysis, the interaction relation stands in for the concrete interaction semantics.

Definition 23. A *code-region interaction* is a tuple consisting of an interaction relation $\blacklozenge \in \diamondsuit$, where $\diamondsuit = \mathbb{P}(\mathcal{CR} \times \mathcal{CR})$ is the set of all interaction relations, a code region $r \in \mathcal{CR}$, and the set of code regions which interact with r through $\blacklozenge$.

$$\mathsf{CRI} : \left(\diamondsuit,\ \mathcal{CR},\ \mathbb{P}(\mathcal{CR}) \right)$$

Similar to code regions, we define the following accessor functions for each value in a code-region interaction.

Definition 24. Let relation, baseRegion, and interactingRegions be functions that map a code-region interaction to the elements that are part of the interaction, and let $\mathcal{CRI}$ be the set of all code-region interactions.

$$\begin{aligned} \mathsf{relation} &: \mathcal{CRI} \to \diamondsuit \\ \mathsf{baseRegion} &: \mathcal{CRI} \to \mathcal{CR} \\ \mathsf{interactingRegions} &: \mathcal{CRI} \to \mathbb{P}(\mathcal{CR}) \end{aligned}$$

Furthermore, using interaction relations, we define the function interactingCR to compute all regions that interact with a given code region through a concrete interaction relation $\blacklozenge$.

Definition 25. The function interactingCR($\blacklozenge$, r_b, revs) computes the set of all regions that interact with $r_b \in \mathcal{CR}$ with regard to a concrete interaction relation $\blacklozenge \in \diamondsuit$.

$$\begin{aligned} \mathsf{interactingCR}(\blacklozenge, r_b, \mathtt{revs}) = \{\, r \mid\ & r \in \mathsf{computeCR}(\mathtt{revs}) \\ & \wedge\ r_b \neq r \\ & \wedge\ r \blacklozenge r_b \,\} \end{aligned}$$

For a given software project $\mathcal{P}$, we compute all CRIs that are present in one of the specified revisions $\texttt{revs} \subseteq \mathcal{R}$ through the function computeAllCRInteractions. This means, we compute through an analysis, which implements $\blacklozenge$, for each revision in revs all interactions between code regions in $p^{\texttt{rev}}$.

Definition 26. The function computeAllCRInteractions computes the set of all CRIs that are present in the software program at one of the specified revisions $\texttt{revs} \subseteq \mathcal{R}$, regarding the concrete interaction relation $\blacklozenge \in \diamondsuit$.

$$\begin{aligned}\text{computeAllCRInteractions}(\blacklozenge, \texttt{revs}) = \\ \Big\{ (\blacklozenge, r_b, \text{interactingCR}(\blacklozenge, r_b, \texttt{revs})) \;\Big|\; \\ r_b \in \text{computeCR}(\texttt{revs}) \\ \wedge \; \big|\text{interactingCR}(\blacklozenge, r_b, \texttt{revs})\big| > 0 \Big\}\end{aligned}$$

Practically, this means running a potentially expensive static or dynamic program analysis multiple times that computes the necessary information to determine whether code regions interact.

In what follows, we introduce three concrete interaction relations, corresponding to the three interaction types described above. We then conceptually highlight how we can lift a static and a dynamic program analysis to code regions, to find these interactions. Later in Section 3.5, we describe two concrete research problems that we could solve through code-region interactions and our code-region-based static and dynamic analyses, demonstrating the applicability of our code region abstractions.

3.3.1 Static Analysis

Statically determining interactions between different types of variability can be beneficial in many ways. For example, structural interactions of feature code indicate that the implementation of a feature needs to be adapted if another feature is present, which implies that these features should also be tested together. Another example can be made for *time* variability. Even a small change to a code base can have wide-reaching impact on correctness or performance [83, 203]. For example, a performance bug[1] affecting FIREFOX's search-bar autocompletion could be fixed with a two line patch that altered the chunksize and timeout of the search bar [83]. So, to better grasp where a change might have impact on a code base, we can use a data-flow analysis to determine data-flow interactions that indicate which parts of a code base are connected to the changed code via data flow.

To address such problems, we model structural and data-flow interactions through our code region abstraction and describe how static analyses are able to determine these interactions.

1 https://bugzilla.mozilla.org/show_bug.cgi?id=415489 (Last accessed: June 30, 2023)

Structural interactions. We introduced structural interactions as interactions that arise from the code structure itself (e.g., through nested feature code). At the core of such a structural interaction is that code from two different implementation overlaps (i.e., is shared between the two implementations). For example, code implementing one feature overlaps with the code from another feature, that is, the two features share common parts that are attributed to both features. With regard to code regions, this means that code that is contained in one code region, is also part of at least another (i.e., there is an overlap of the instructions contained in one code region with the instructions contained in another). Therefore, we define the interaction relation $\circledcirc$ for structural code-region interactions based on the structural overlap between code regions.

Definition 27. $r_1 \circledcirc r_2$ is the structural interaction relation that holds when at least one instruction that is part of r_1 is also part of r_2.

$$r_1 \circledcirc r_2 = \exists_{i \in r_1} \exists_{i' \in r_2} i = i'$$

In practice, finding overlappings between code regions is straightforward and computationally inexpensive. With a simple intra-procedural analysis that does a linear scan over the instructions of r_1 and r_2, we can determine if they have at least one in common. Therefore, even a naive implementation that compares all code regions against each other has a quadratic worst case complexity $\mathcal{O}(n^2)$, where n is the number of code regions.

Even if structural code-region interactions are only local and easy to detect, they still can give us valuable information about crosscutting features, which occur often in software systems [10, 43, 144]. For example, similar to Fischer et al. [68] that predict structural feature interactions, the computed interaction information could be used to guide analysis approaches that want to cover probable feature interactions only, as they do not want to pay the computation cost to cover all potential ones. Furthermore, they could be used in combination with other interactions types to define more specific interaction cases.

Data-flow interactions. Compared to structural interactions, data-flow interactions have two benefits: (1) Data-flow interactions explicitly encode data relationships, that is, they only infer relationships in cases where data, produced by one part of the program, is actually used in another. This helps to reduce spurious interactions. (2) Data-flow interactions allow us to identify non-local relationships, or even program-wide relationships if an inter-procedural analysis is used.

Data-flow analysis has been used for a long time (e.g., in compilers or for security analysis) to determine if specific parts of a program can pass information to another. For example, polyhedral compiler optimizations use data-flow information to extract data-dependencies (i.e., read/write orderings) [79, 244], or in security, where data-flow analysis determines whether secrets or private information could be leaked to an attacker [67, 204].

However, applying precise data-flow analysis to reason about different types of variability has long been burdensome. To ease the application of data-flow analysis

to variability, we demonstrate how to conceptually lift a data-flow analysis to code regions, and by that to various kinds of variability.

We define interaction relation $\rightsquigarrow$ for data-flow code-region interactions based on the data-flow relationships (DF) between instructions of one region with the instructions of another.

Definition 28. $r_1 \rightsquigarrow r_2$ is the data-flow interaction relation that holds when the data produced by at least one instruction i that is part of r_1 flows as input to an instruction i' that is part of r_2.

$$r_1 \rightsquigarrow r_2 = \exists_{i \in r_1} \exists_{i' \in r_2} \mathsf{DF}(i, i')$$

It is crucial to understand that this definition utilizes the underlying program representation—in the form of instructions—to link variability information to data-flow analysis results. This means, we can now use any data-flow analysis to compute data-flow interactions between different types of variability. Initially, we compute the data-flow relationships between instructions through the data-flow analysis. Then, we use the analysis results, the interaction relation $\rightsquigarrow$ (see Definition 28), and the function computeAllCRInteractions (see Definition 26), to compute all data-flow interactions between code regions. This way, choosing or optimizing the precision with which data-flow-based variability interactions are computed becomes solely a program analysis optimization problem.

3.3.2 Dynamic Analysis

Dynamically determining interactions between different types of variability can become necessary in cases where static analyses introduce too much overapproximation or when we want to reason about runtime properties of a configurable software system that can be observed only during execution.

To demonstrate the applicability of our code-region interactions to dynamic analyses, we focus in our work on the analysis of non-functional properties, which is a research area that has gained a lot of traction over the last 10 years in the area of variability research. The analysis of non-functional properties, such as, performance, memory consumption, or binary size, of configurable software systems and software product lines focuses on determining and understand the effect variability can have on the NFPs [215, 216]. Even after many years, there are still unsolved problems, especially in the area of white-box program analysis. Notable exceptions are ConfigCrusher [240], Comprex [241], and the work of Weber et al. [247]. However, the authors indicate open problems, such as scalability or interpretability issues, which shows that there are existing difficulties that should be addressed. This stems from the fact that combining dynamic white-box analyses with variability is inherently difficult. We aim to ease further research in this direction by unifying these dynamic analyses to measure NFPs with our code regions. Therefore, we introduce NFP interactions to model interactions of non-

functional-properties not only for software product lines but generally for different types of variability.

NFP interactions. Configurable software system have many NFPs, such as performance, that are of interest in a variety of use cases. Furthermore, there are many approaches, techniques, tools to determine those NFPs. For our work, all those can be conceptually grouped together into either a direct dynamic analysis or a dynamic analysis coupled with a statistical evaluation, as our interaction relation only connects our interactions to an effect determined by the approach. Hence, it is only important in the end that the approach can infer a relationship between different code regions, that is, infer a relation between different variabilities. Take for example, feature-related performance measurements, where we want to identify the features involved in a performance regression. In this case, it is not important for our interaction abstraction which tool is used for performance measurements or which statistical test is used to determine whether the measured change is significant, just that there *is* a way of determining an effect between different versions.

However, apart from the inessential differences between the approaches, analyzing NFP interactions introduces two difficulties compared to structural or data-flow interactions: (1) It depends on the specific analysis whether the NFP interaction has a direction or not (e.g., many statistical evaluations of performance data provide a correlation but have no information about who influences whom). (2) NFP interactions might only be observable between different revisions of a program (e.g., a performance regression is usually a decrease in performance compared to a previous revision of the same program).

Difficulty (1) means that it depends on the particular analysis and evaluation whether a direction can be inferred. Therefore, to model cases with and without direction, we define the NFP interaction relation without a specific direction and defer the interpretation of directionality to the concrete analysis semantics. This means, in cases where the analysis cannot infer a direction the relation is commutative, and by that, a code-region interaction between r_1 and r_2 is equivalent to an interaction between r_2 and r_1. We address difficulty (2) by using the revision specifier (see Section 3.2.4) to contextualize the code regions that are involved in an interaction relation. This way, we explicitly model the revision and can infer interactions between different program revisions.

We define interaction relation $\Rightarrow$ for NFP code-region interactions based on the effect that is observed on the non-functional property by the analysis or evaluation.

Definition 29. $r_1 \Rightarrow r_2$ is the NFP interaction relation that holds when the underlying analysis measured an effect between r_1 and r_2.

$$r_1 \Rightarrow r_2 = \mathsf{Effect}(r_1, r_2)$$

It is important to note that we define $\Rightarrow$ quite loosely on purposes to cover a wide range of dynamic analysis and evaluation methods. Nevertheless, a specific dynamic analysis can always use a more strict definition of $\Rightarrow$ to be able to make

more precise statements. For example, determining if there is an Effect between two regions, could be done measuring the runtime of each region with a dynamic analyses and determining if more runtime is spent in region r_1 as in region r_2. But also more elaborate, like using causal analysis to determine if executing region r_1 causes a performance regression of region r_2, similar to the work on feature causality by Dubslaff et al. [56], however, with the extension to code region granularity instead of features. Important to define Effect is that the dynamic analysis and evaluation can attribute their results to regions (e.g., through a unique ID in practice).

3.4 Code Regions in LLVM

In the previous sections: We defined our code-region abstraction around domain-specific tags to represent variability; We conceptualized how to detect these code regions in a program representation; We showed how various static and dynamic analyses can be used to find interactions between different types of variability. Now, we describe how these concepts and ideas can be implemented and used in practice.

Based on our definitions, we implemented code regions as an extension to LLVM. In what follows, we give a brief overview of how we integrated our code regions into LLVM and highlight important practical details on how we use them to encode variability information. Next, we explain how code regions can be automatically detected within the compiler and highlight peculiarities that arise when detecting code regions at later stages in the compiler pipeline. Last, we depict how domain-specific static and dynamic analyses that are based on code regions fit into the LLVM framework.

3.4.1 Overview

On a high-level, the implementation of our domain-specific analysis framework is divided into two parts: the language-dependent front-end extensions and a language-independent analysis layer. The front-end extensions to Clang load, process, and encode the variability-specific information into LLVM's intermediate representation. Afterwards, region-detection passes convert the variability-specific information into code regions, which then can be freely used by code-region-specific analysis passes. Figure 3.4 gives an overview of the complete compilation setup and highlights the variability specific parts.

Front-end extensions. For each type of variability, we implemented a thin front-end extension to map the variability-information into LLVM-IR. We kept the language-specific front-end extension small, so that the core part of our framework is language independent and can be used with other languages. We map *space* variability information by annotating which variables encode configuration

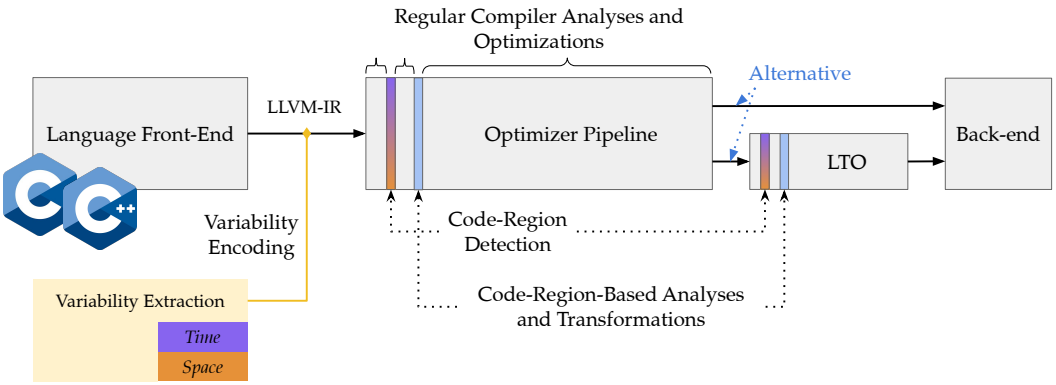


Figure 3.4: Overview of our compiler-based variability analysis pipeline.

options into the LLVM-IR (see Section 5.2.2). For *time* variability, we map repository information in the form of revisions into the IR (see Section 4.3.1).

Analysis pipeline. Following LLVM's design, we implemented region detections and all our analyses as LLVM passes, so that they can be freely applied in different stages of the analysis pipeline. This is especially useful to enable whole-program analysis, as both the region detection and analysis can be used during LTO for whole-program analysis. Furthermore, this design automatically decouples analyses from the specific code region type that is used. The detection pass that processes the variability information added to LLVM-IR is the only component that is variability specific. The analysis passes that are executed afterwards are implemented only based on the code region interface and process the code regions that were detected before. In Section 3.4.5, we describe in more detail how static and dynamic analyses are implemented in LLVM's pass infrastructure.

3.4.2 Encoding Variability Information into LLVM-IR

The first step, to make variability accessible in the compiler, is extracting it from an external variability source and attaching it to the compiler's internal representation. Conceptually, this represents the tagging function introduced in Definition 11. In order to achieve that, we need to modify the compiler front-end to take in the external variability source, extract its information, and encode it, by attaching it to the correct instructions during the lowering from the source language to LLVM-IR. We explain how this process works in detail for the different types of variability later in Section 4.3.1 and Section 5.2.2.

To ease the development of different variability types, we split the variability-specific implementation into a variability provider and added an annotation factory to annotate instructions with variability-specific meta-data. The annotation factory removes the complexities that arise from the LLVM-IR construction process by

removing the details of how meta-data is attached. This means, the annotation factory separates the *where* and *how* meta-data is attached in the compiler from the *when* and *what*, provided by the concrete variability provider. The variability provider tracks during lowering from AST to LLVM-IR the variability information and provides this information in the form of meta-data when queried by the annotation factory. Through that design, the variability provider can focus only on the two important parts: *variability extraction* and *variability encoding*.

After implementing this variability-specific lowering step, we now have LLVM-IR that is enriched with variability information. However, this information is currently unstructured and not easily accessible by program analysis. We solve both these issues in the next step of the pipeline, where we extract code regions from the annotated variability information.

3.4.3 A Domain-Specific View on LLVM-IR

We introduced code regions in Section 3.2 to provide a domain-specific view on a program, by abstracting away variability information behind an uniform interface. In Definition 12, we define code regions on a conceptual program representation. However, to apply our uniform abstraction of code regions to actual program analyses and use these analyses on real-world programs, we need a concrete implementation of our abstraction.

We implemented our code-region abstraction in the compiler framework LLVM as a lightweight overlay to LLVM-IR. The code-region overlay adds the interface type `llvm::IRegion` for a code region and handles generic implementation details, such as indicating where in the LLVM-IR the region is (i.e., which instructions belong to the region). All variability-specific details, are implemented in separated classes under the `llvm::IRegion` interface type. We chose a lightweight overlay compared to adding special instructions or other constructs, as modifying LLVM-IR would introduce many unnecessary complexities. Furthermore, conceptually we want to present variability information alongside the program representation, without influencing the LLVM-IR semantics, as this would affect existing program analyses and by that, make them less reusable. Important to note, the code regions introduced in this section should not be confused with a `llvm::Region`, which represents a part of a CFG with one entry and exit node. Our code regions offer a much more fine-grained and detailed view on the code, as they are not restricted to basic blocks in the CFG but work on instruction granularity and, besides that, they are not limited by the amount of entry and exit nodes, allowing them to have more precise borders.

From a design perspective, code regions can be seen like a view of a database. The view presents the same information that is already present in database tables but adds additional contextual information by aggregating or filtering out unnecessary information. Conceptually, the same applies to code regions as they present the same LLVM-IR but add contextual variability information, by aggregating variability-

related code and present it to analyses, hiding all the unnecessary non-variability-related parts.

At the level of the code region interface type `llvm::IRegion`, we implemented accessor methods to LLVM-IR, such as iterators, and the interface methods presented in Section 3.2.5. From both our practical implementations, we learned that the original designed interface is enough to implement our analyses presented in Chapter 4 and Chapter 5, indicating the abstraction works quite well. However, further extending it would also be an option, as long as the extensions keep the theoretical separation of program representation and variability information.

Code regions provide a central uniform interface for representing variability information to analyses passes. Next, we explain how we extract code regions during program compilation in LLVM's optimizer through special analysis passes.

3.4.4 Detecting Code Regions

As described in Section 3.2.3, detecting code regions is a straightforward process where we group together adjacent instructions that carry the same variability information. This works well in theory and most often also in practice. However, in some whole-program analysis scenarios, we encounter problems where compiler optimizations can delete variability information before we could analyze it. In order to solve this, we need to introduce an additional information-preservation step for whole-program analyses.

We implement the region detection, as introduced in Section 3.2.3, with a reusable generic function that implements the core of the detection algorithm. Algorithm 1 depicts the implementation of the $\mathsf{computeCR}_{\mathsf{Tag}}(f, t)$ function (see Definition 14), which takes as an input a function f and the tag t, for which code regions should be detected. In short, we walk over each basic block in a function's CFG and create potentially incomplete code regions for all connected instructions that carry the tag t. Afterwards, we join code regions that are connected, meaning, we join code regions where the end of one region is connected through a control-flow edge of the corresponding basic block to another region. We then apply this generic function in multiple LLVM analysis passes, creating a separate pass for each type of code region we want to support (i.e., for each type of variability). By separating the detection logic into a generic function, we share the commonalities of detecting code regions, and by, building a separate LLVM analysis pass for each code region type we enable users to individually select the variability they need on the application layer.

Other analysis passes can access code regions by adding a code-region detection pass into the pass pipeline. As depicted in Figure 3.4, a code region detection pass can be inserted into the compiler pipeline at different stages: either during the normal optimization stage, or during LTO. We found that inserting code-region detection passes before compiler optimizations and directly before that analysis works best, as currently detected code regions do not dynamically update when the underlying LLVM-IR changes.

Algorithm 1: In practice, the code-region detection algorithm is comprised out of two steps. First, we compute partial code regions for each basic block by iterating over the block and forming a code region for each connected block of instructions that carry the specified tag $t \in \mathcal{T}$. This step is necessary, as LLVM-IR is structured around basic blocks compared to our instGraph. Second, we combine partial code regions, should they be adjacent. This means, should the underlying basic blocks fulfill a predecessor-successor relationship and should the predecessor/successor code regions start/end with the first/last instruction of the underlying basic blocks.

Parameters: $f \in p$, $t \in \mathcal{T}$
Result: detectedRegions

```
1  Function computeCR_Tag(f, t):
       ▷ First, detect partial regions per basic block.
2      partialBBMap ← ∅;
3      for bb ∈ getBasicBlocks(f) do
4          taggedInsts ← ∅;
5          for i ∈ getInstructionsInSequence(bb) do
6              if t ∈ tags(i) then                          ▷ expand current region
7                  append (i, taggedInsts);
8                  continue;
9              else                                         ▷ complete region
10                 newRegion = CodeRegion(taggedInsts, t);
11                 append (newRegion, partialBBMap[bb]);
12                 taggedInsts ← ∅;
13             end
14         end
15         newRegion = CodeRegion(taggedInsts, t);
16         append (newRegion, partialBBMap[bb]);
17     end
       ▷ Second, merge partial regions, when adjacent to other region.
18     for bb ∈ getBasicBlocks(f) do
19         partialCRs ← partialBBMap[bb];
20         for partialCR ∈ partialCRs do
21             if startsAtFirstInst(partialCR, bb) then
22                 for pred ∈ pred(bb) do
23                     for predPCR ∈ partialBBMap[pred] do
24                         if endsAtLastInst(predPCR, pred) then
25                             combinedCR = combine(partialCR, predPCR);
26                             replace (partialCR, combinedCR, partialBBMap[bb]);
27                             replace (predPCR, combinedCR, partialBBMap[pred]);
28                         end
29                     end
30                 end
31             end
32         end
33     end
34     detectedRegions ← removeDuplicates(concat(values(partialBBMap)));
35     return detectedRegions
```

As mentioned above, this setup does not work by default for some whole-program analysis scenarios. To be precise, it does not work in scenarios that: require compiler optimizations, want to run code transformations based on the analysis results, and need to produce an executable binary after the analysis. Unfortunately, this is exactly the case for dynamic analysis, such as the instrumentation-based performance profilers presented later in Chapter 5. Usually, the problem of compiler optimizations can be circumvented by running the analysis before any optimizations are applied to the IR. However, this is not possible for whole-program analyses that need to produce an executable binary. As described in Section 2.3.5, there are two ways to build whole-program analyses: either by using tools such as WLLVM that produce a whole program bitcode file, or by running the analysis during LTO. In the specific case described above, we cannot use WLLVM and need to run our analysis and the code transformation during LTO. The problem of loosing variability information that is tagged to LLVM-IR, now arises when optimizations transform the IR before every module is linked into a whole-program module and processed during LTO. We solve this problem in two ways. First, we patch a few central LLVM-IR rewriting functions in LLVM to preserve variability information encoded into the meta-data by attaching the meta-data to the newly generated replacement instructions. Second, we run a lightweight intra-procedural pre-analysis of the actual whole-program analysis. We persist the intermediate results of the pre-analysis and then later feed this information into the actual whole-program analysis. For example, in the case of a taint analysis, we persist the generated taints as meta-data and reintroduce these taints again later during the whole-program taint analysis. This way, we can support even specific dynamic analyses that require a static whole-program analysis to setup the instrumentations for the dynamic analysis.

We implemented the conceptual code-region detection described in Section 3.2.3 in LLVM, demonstrating that detecting code regions works also in practice.

3.4.5 Static and Dynamic Analysis

Building precise static and dynamic analyses, such as inter-procedural data-flow analyses or instrumentation-based performance profilers, is difficult due to the algorithmic complexity and implementation efforts required [208]. However, adapting an existing program analysis to a code-region-based domain-specific analysis and provide it to LLVM is straightforward.

First, we integrate all analyses into the LLVM pass pipeline, by wrapping the analysis implementations into LLVM passes, which makes them easier usable. Next, to lift analysis to code regions, we have three options: (1) Map the results of an existing analysis to code regions; (2) Adapt an existing analysis to interface with code regions; (3) Develop a new analysis that is tailored around code regions. (1) Mapping analysis results to code regions works well in all cases where the analysis semantics are only based the program and do not depend on variability information. For example, we can run a standard data-flow analysis that works on instructions

and then map the produced results, which define data-flows between instructions, to code regions by finding the code region that contains the instruction. (2) We can adapt an existing analysis to integrate code regions by specializing an analysis problem to code regions. For example, an existing taint analysis could be modified to treat code regions as sources or sinks, to detect what gets influenced by a code region (i.e, being tainted) or if tainted values reach a specific code region. (3) Writing a specialized analysis is normally only required when variability information is deeply intertwined with the analysis semantics and can not easily be separated out. In our work, we did not encounter a case where it was strictly necessary to write a new analysis.

In general, program analyses often require additional helper analyses, for example, to compute pointer-aliasing information or a call graph. Through the LLVM analysis infrastructure, such helper passes can be required, and as code-region-based analyses passes fit natively into the analysis infrastructure, they also can be added and reused to build more elaborate program analysis. In what follows, we highlight conceptually how static and dynamic analyses can connect to code regions.

Analysis passes (static analysis). Using the static program analyses that LLVM provides or adapting them to code regions is in many cases straightforward, as all of them are based on LLVM-IR. However, by default LLVM does not offer advanced inter-procedural program analyses or many utilities to build such analyses. To compensate for this disadvantage, we integrated PhASAR, a LLVM-based framework for writing inter-procedural static analyses, into our variability-focused analysis framework. Similar to analyses provided by LLVM, we can adapt PhASAR-based analyses to code regions as they all work on LLVM-IR. Meaning, developers can not only use analyses provided by PhASAR but also use PhASAR to develop new inter-procedural analyses that can afterwards be lifted to code regions. For example, together with the PhASAR developers, we added a new customizable inter-procedural data-flow analysis that determines potential data-flows between all instructions. Afterwards, we then lifted this analysis to code regions in our framework. We demonstrate this analysis later in Section 4.2.3 in detail.

Transformation passes (dynamic analysis). In general, variability-focused dynamic analyses require users to build a transformation pass that injects the dynamic analysis or specific anchor points, to which the dynamic analysis later attaches, during compilation into the produced binary. For this purpose, we implemented the entries (see Definition 21) and exits (see Definition 22) functions in the code region interface, which enable us to instrument variability specific code around code regions (e.g., to insert instrumentation code or anchor points around a code region). Important to note, transformation passes that insert dynamic analyses often require additional static helper analyses that compute where the dynamic analysis code should be placed. As noted above, we can integrate such static analysis, especially the ones that already work with code regions, through LLVM's pass pipeline.

3.5 Code-Region Analysis Applied to Variability

In what follows, we give an overview of the two research applications that we use to demonstrate that our code-region abstraction enables us to analyze *time* variability, arising from software evolution, and *space* variability, introduced by configuration options, equally. The goal of the research applications is to demonstrate that our uniform abstraction of code regions enables novel applications for program analyses that target different types of variability, where code regions provide the glue between different types of variability and reusable white-box program analyses. We do this, by applying our code-region abstraction to two novel research problems, targeting different types of variability and requiring different program analyses to be answered.

In the first research application, presented in Chapter 4, we combine repository mining with static program analysis to uncover new and interesting connections between developers. In the second research application, presented in Chapter 5, we make state-of-practice performance profilers configuration-aware to facilitate performance analysis. In what follows, we give a brief introduction to both research applications and describe how they demonstrate the applicability of our uniform code-region abstraction. Afterwards, in the following chapters, we present each research application in full detail and highlight throughout the application how specific parts relate to the code-region abstraction.

Data-flow interactions between revisions. Repository-mining approaches analyze code typically at file, textual, or syntactical level and thus miss important connections that are encoded in the underlying program semantics. For example, socio-technical approaches that try to determine which developers should coordinate, use co-change information only on a file or function level to infer connections between developers [105], leaving out valuable information. In the end, many repository mining approaches do not use and, by that, do not profit from the information that is encoded in the underlying program semantics because this information is not easily accessible to them.

To solve this, we built a bridge between repository mining and program analysis. We introduce commit-based code regions that model which parts of a code base have been introduced by a specific revision. We then use the data-flow interaction concept described in Section 3.3.1 and a novel code-region-based data-flow analysis, to determine which revisions interact through data flow. These data-flow interactions give us a better understanding of how code changes, introduced by a revision, interact throughout the code base and through the link from commit to the commits's author, also how code from different authors interacts. This way, we use our code-region abstraction to integrate static program analysis with repository mining.

With the first research application, we demonstrate three central points: (1) How *time* variability can be modeled with code regions; (2) How a generic static data-flow analysis can be built using our uniform code-region abstraction; (3) How code-region

interactions that arise from the analysis results can help to gain new insights, in this particular case, we showed how a data-flow analysis can uncover previously unrecognized interactions between developers.

Configuration-aware performance analysis. Research has shown that a significant number of performance bugs are caused by configuration errors [83]. However, reasoning about performance bugs in configurable software systems is difficult because state-of-practice profilers do not take the configurability of the system into account. Therefore, they have to measure all potentially impacted variants, which is difficult for large configuration spaces that arise from the combinatorial explosion. Furthermore, from the collected performance profiles, it is not clear which parts of the measured execution time is related to which configuration choice, making it hard for developers to judge whether and by how much specific configuration options influence the performance of a configurable software system.

We tackle this problem by introducing Walrus, a dynamic analysis framework to make state-of-practice performance profilers configuration-aware, which is built through our code-region abstraction. We model compile-time and load-time configuration-dependent code as code regions. For localizing load-time configurable code, we reuse the analysis developed in the previous research application to determine which configuration options have data-flow interactions with control-flow decisions in a program. After the localization phase, we make the time spent within these configuration-specific code regions measurable by weaving in performance measurement code. This way, we enable configuration-aware performance analysis with state-of-practice performance profilers.

With the second research application, we demonstrate three central points: (1) How *space* variability can be modeled as code regions; (2) How a code-region-based data-flow analysis, developed in the previous application, can be reused to localize configuration-dependent code; (3) How we can enable code-region-based dynamic analyses, in the form of performance profilers, to measure the execution time spend within a code region.

Taken together, the two practical research applications, where we applied our code-region abstraction to novel research problems, demonstrate that our uniform code-region abstraction is a valuable link between different types of variability and white-box program analyses. Overall, they demonstrate that multiple types of variability are uniformly represented by code regions and that code regions can be analyzed by static as well as dynamic program analyses.

3.6 Related Work

In what follows, we give an overview of different works that relate domain-specific information, in general, and variability information, in particular, to code regions.

Configuration-dependent code regions. In their work about feature interaction faults, Garvin and Cohen [70] investigate through an exploratory study if real-world faults fit their new definition of interaction faults. During the construction of their definition, they introduce the notion of a variability region, meaning a maximal set of basic blocks that have the same configuration dependence. That is, if one is executed under a given configuration the others are too, except for non-configuration reasons [70]. Their work is connected to others that analyze `#ifdef` variability and presence conditions [7, 19, 193], as preprocessor directives explicitly define a region of code that gets included in a program when a specific presence condition is fulfilled. Velez et al. [240, 241] also introduce configuration-dependent code regions to model the code that is controlled by load-time configuration options. Both variability regions and configuration-dependent code regions are closely related to configuration-dependent code regions that we introduce later in Chapter 5, and could be expressed with our code regions definition—with the restriction that both parts of preprocessor region cannot be modeled at the same time. However, structurally our code region definition allows regions to be more fine-grained, as they are defined on an instruction level compared to a basic blocks. Furthermore, where their work introduced code regions, just as a means of aggregating configuration-dependent code, our code-region abstraction incorporates a connection to program analyses and uniformly represents different types of variability.

Variability-aware program analysis. With SPL$^{\text{LIFT}}$, Bodden et al. [25] demonstrated how existing IFDS-based program analyses can be made software-product-line aware by lifting them to IDE, encoding the variability from the product line into the edge domain. Through their lifting process, conventional program analyses can be reused on software product lines, analyzing the whole product line at once. Compared to their work, our work focuses on analyzing different types of variability by combining and reusing existing program analyses. So, where their works makes existing analyses variability-aware, we apply existing analyses to the domain of variability to better understand and reason about the implications variability has on configurable software systems (e.g., by finding data-flow-based feature interactions).

Polyhedral optimizations. Another type of region that is used in the context of LLVM for polyhedral optimizations are static control parts (SCoPs) [17, 79]. A SCoP is the polyhedral representation of a static control-flow region that consists of nests of counting loops (i.e., `for` loops). Furthermore, the loop bounds and memory accesses inside a SCoP must be linearly affine expressions. Compared to code regions, SCoPs do not carry different types of domain specific information but only focus on polyhedral information. More important, SCoPs differ structurally from our code regions as their granularity is based on basic blocks, and they require a single entry and exit block. Furthermore, SCoPs constrain the control-flow structures that can be part of a SCoP to counting loops and `if` statements, and the memory accesses to affine expressions. These restrictions are important for polyhedral optimizations,

however, using SCoPs instead of code regions for our variability abstraction would constraint our use cases too much.

Architectural conformance. Architecture conformance checking is the process of verifying that the implemented architecture of a software system follows the architecture defined in the architecture description (e.g., checking that in a layered architecture each layer only access the ones above or below it). To achieve this, tools, such as SAVE [57], ArchJava [2] or ConQAT [54], take a source-code model, an architecture model, and a mapping between the two and check if the specified architecture is correctly represented in the source code [123]. Important in our context, architectural conformance checking requires a mapping between source-code artifacts and the architecture model. This is related to our work, as source-code artifacts often refer to specific regions in the code. However, their work differs from ours as conformance tools work on the source-code level compare to code regions, which are defined on a low-level program representation, suited for inter-procedural program analysis. Nonetheless, as we highlight in Section 3.7 in detail, architecture conformance checking could profit from our work by integrating new state-of-the-art inter-procedural program analyses.

Requirements tracing. Requirements tracing is the process of tracking requirements, starting from the requirement description and tracking, through various artifacts up to the source code that implement the requirements, how the initial requirements where realized in the software product [35]. Precisely mapped requirement information helps developers to implement tasks faster and with fewer bugs [149], as well as with bug localization [187]. To achieve a precise connection between requirements and source code, similar to architecture conformance checking, requirements tracing approaches need a way to map requirements to code artifacts and keep them up to date throughout the evolution of the software system. Lian et al. [143] combine architecture information, requirements tracing, and source code information to provide round-trip traceability between source code and requirements (i.e., quality concerns) and visualizations that depict how those requirements are woven into the code [158, 159]. Rahimi and Cleland-Huang [186] propose a solution for evolving mappings between requirements and source code to keep source code mappings up to date. They introduce 24 change scenarios, based on high-level change patterns that they mined from open-source projects, and combine them with a link evolution heuristic to update the requirements links. So, when their tool detects a scenario in a change, it can use the link evolution heuristic to automatically update the requirements links, keeping them up to date. Furthermore, Kuang et al. [131] demonstrate that, in addition to call dependencies, data dependencies are important for requirements traceability, complementing the information provided through call dependencies. Due to the mapping of information to source code parts, such works are related to code regions, which could be used to combine requirements tracing with program analyses.

Coverage information. Coverage data is used, in addition to tracking test coverage at different levels [115], in a wide range of applications, such as coverage-guided fuzzing [167], test generation [60, 189], and test case selection [3, 21, 87, 199]. The key benefit of coverage data lies in the fact that it provides a mapping between dynamic execution traces and source code, that is, a mapping from execution traces to a more high-level and comprehensible abstraction, the source code [89]. Afterwards, repository change information can be used to select a subset of the test to cover a change [3]. There are two important parallels between coverage tracing and our work. First, the coverage data introduces a mapping similar to our code-region information mapping from source code to analysis information. Second, similar to code regions, coverage data also decomposes code into blocks. For example, structure-based criteria [253], such as control-flow coverage, track execution on a basic block level [188], or on an even more fine-grained level for statement or path coverage [33]. Hence, coverage data could be seen as a specialization of our general code-region abstraction to another domain.

3.7 Further Applications

During the implementation and work on variability-specific code regions, we realized that by abstracting away program analyses from domain-specific information, analyses could be applied to many other use cases and domains. In what follows, we highlight various use cases that could be modeled through our code-region abstraction and, by that, analyzed through our generalized analyses.

Performance-relevant code. A straightforward application of code regions would be to model performance relevant code (i.e., performance hot spots). The information extracted from a performance profile, for example, produced through LLVM's profile-guided optimization infrastructure, could be used to represent hot-code parts as code regions. This way, existing code-region-based analyses could be used to better reason about the interplay between performance relevant code and variability. For example, to determine which features interact with a performance-relevant code region through data flow.

Security. Code regions could also be applied to the security domain. Many security analyses, for example, use taint analyses to determine if a private key could be leaked to the user, are highly specific to their use case. However, by introducing two domain-specific security regions—one to mark private data locations and one for user facing APIs—, code-region-based analyses could compute whether a private key could be leaked, without being overly specific. This way, analyses implemented against the uniform code-region interface could be shared. Furthermore, combined analyses that target security in the context of variability could be devised to get a more detailed picture. For example, a code-region-based analysis could analyze the

interplay between private leaks and feature code, in a single analysis run instead of two separate analyses.

Architectural abstractions. Another domain that could be modeled through code regions are architectural abstractions. In general, architectural abstractions are connected pieces of code, even when the abstractions itself is spread all over the code base. For example, in a layer architecture [62] each layer is responsible for specific tasks and communicates only with adjacent layers through a defined API. The main purpose of a layered architecture is to introduce separation of concerns, so that each layer can be changed independently. However, such designs can suffer from architectural decay as the software project evolves [74, 229], which compromises the design goals and introduces bugs [135, 160]. For example, we could analyze such architectural decay through data-flow analyses that detect when non-connected layers interact. For that, we only need to provide a mapping that encodes layer information to LLVM-IR, creating architecture-specific code regions.

Jupyter *code cells.* An application-specific use case are JUPYTER code cells. In JUPYTER[2] notebooks, code is divided into multiple cells, where each cell can be executed independently an often represents a semantically related piece of code. These code cells could be mapped into LLVM-IR as a code region for further analysis. For example, it could be interesting to employ data-flow analyses on these code cells, to determine which cells need to be reevaluated after a change, or detect with which external software components a cells interacts. Venkatesh et al. [243] already showed that their static-analysis-based approach HEADERGEN can automatically generate headers for JUPYTER code cells that describe, based-on a taxonomy, what the specific cell is meant to do.

In general, as JUPYTER code cells have grown in popularity over the last years, it seems promising to extend existing analysis capabilities to them. Through our code-region abstraction, this should be simple. By representing a code cell as a code regions, we can automatically apply all existing static or dynamic analyses that work for code regions to JUPYTER code cells.

Overall, we found that many high-level concepts map relatively cleanly to our code-region definition. We attribute this to the fact that, in the end, all information that should be analyzed either by static or dynamic white-box analyses needs to relate to specific pieces of code. We find that extending our work into these direction looks promising from an analysis point of view for all users that want to apply precise static and dynamic analyses, without paying the cost of building their customized analyses. With our underlying framework, users only need to provide a mapping function from high-level conceptual information to code, that is, they need to provide the mapping functions `tags` (see Definition 11). Afterwards, our

2 https://jupyter.org/ (Last accessed: May 13, 2023)

framework constructs domain-specific code regions using this function, and by that, the users get access to a wide range of static and dynamic program analyses.

3.8 Summary

This chapter presented our uniform abstraction of code regions that enables us to analyze domain-specific information with static and dynamic analyses in a reusable way. Initially, we identified two major challenges that hinder the wider adoption of variability-focused white-box analyses: (1) Accessing variability information is difficult for white-box analyses; (2) Building real-world white-box analyses is difficult and time-consuming.

To solve these challenges, we addressed three problems with our uniform code-region abstraction. First, through our uniform code-region abstraction, we decoupled variability from the actual analysis, showing that the analysis semantics can be split into an analysis-specific and a variability-specific part. Second, we demonstrated that the detection of code regions can be realized in a generic way, so that it can be reused for different types of variability and other domain-specific concepts. Third, we showed how interactions can be lifted to code regions and how reusable program analyses can be built solely based on our uniform code-region abstraction, to compute different kinds of code-region interactions. In the end, code regions that are constructed in a generic way from domain-specific tags present a uniform interface to white-box analyses, enabling the analyses to access variability information easily. Furthermore, through the separation of analysis semantics from domain semantics and the uniform code-region interface, analyses are better reusable. Through this reusability, it becomes more viable to spend the time and effort required to develop such analyses, as they can be applied to multiple problems afterwards.

Through our uniform abstraction of code regions, we can now model and analyze different types of variability in concert with reusable white-box program analyses. In the following two chapters, we demonstrate how our uniform abstraction of code regions enabled us to tackle new and interesting research problems and highlight how code regions enabled us to reuse analyses between the two.

4

Integrating Program Analysis and Repository Mining

This chapter shares material with the following publications [203]

Understanding software evolution and the socio-technical context of software projects is important for improving development processes and the communities that form around software projects [104, 105, 107, 254]. As we have shown in Section 2.2, the techniques used to reason about this problem domain extract information from various project related sources, where, among other things, one prominent information source is data mined from software repositories. The problem with many of the currently used extraction techniques is that they do not incorporate program semantics, such as data-flow relationships, which leads to unnecessary overapproximation and spurious connections between developers [85]. This problem can be solved by incorporating repository information into existing semantic-aware program analyses, since analyses, such as inter-procedural data-flow analyses, already model program semantics quite well. So, by incorporating repository information into these analyses, we can harness the analyses semantic handling and extract program-semantic-aware repository information.

The approach presented in this chapter, demonstrates how we can gather semantic-aware repository information with state-of-the-art inter-procedural program analysis, by using the code-region abstraction introduced in the previous chapter. Therefore, complementary to this chapter's main contribution of enhancing information gathering for evolutionary and socio-technical analyses, this chapter demonstrates also the applicability of code regions to evolutionary variability (*time*) as well as to static analysis.

In what follows, we portray the gap between current repository mining approaches and program analysis in more detail and demonstrate how modern state-of-the-art program analysis can provide additional information to existing evolutionary and socio-technical techniques.

4.1 Introduction

Software systems are among the most complex human-made systems today. To understand the inner workings and external qualities of complex software systems,

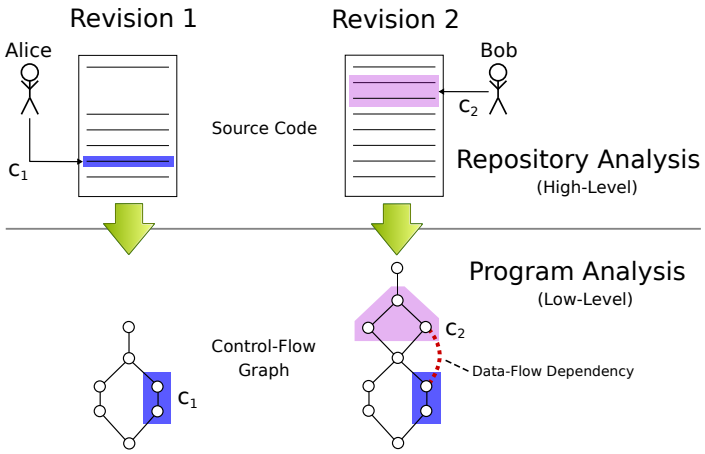


Figure 4.1: Combining low-level program analysis with high-level repository analyses: Alice makes a change c_1 to a source code file creating revision 1. Later, Bob modifies the same file at a different location with c_2 creating revision 2. A typical repository analysis cannot find a connection between c_1 and c_2, except that both modified the same, possibly very large, file. If we map the changes of Alice and Bob onto the corresponding control-flow graph and run a data-flow analysis, we discover a data-flow dependency (red) from c_2 to c_1.

researchers and developers have devised a variety of *program analysis* techniques to extract relevant information, including bug finders, program verifiers, and code metric tools.

The rise of open-source software has triggered the development of *repository mining* techniques that extract organizational information from software repositories. For example, researchers have investigated how code changes evolve on different platforms [245] or analyzed the characteristics of uncompilable code [90]. Other approaches analyze the socio-technical interaction around a software project to better understand how developers collaborate [41, 101, 104, 105].

Such *high-level* repository mining approaches analyze code typically at file, textual, or syntactical level and thus miss important information that is only encoded in the underlying program semantics. Information on the program semantics is often only accessible by *low-level* static program analysis techniques such as data-flow analyses. While high-level repository analyses are sufficient to get good first results, information that is hidden in the program's semantics (e.g., information on data-flow) is overlooked. This insufficiency precludes important use cases such as attributing bugs and security vulnerabilities directly to the commits/developers that introduced or know best how to fix them. For illustration, let us consider the example in Figure 4.1: Alice commits a change c_1, later Bob also changes the same file with commit c_2. With a purely textual or syntactical repository analysis, no connection between the two changes can be inferred, except that both change the same file. By mapping the change to a control-flow graph, we can leverage powerful program analysis techniques to detect previously hidden data dependencies between the changes and, in turn, infer that c_2 could be related to a bug in c_1.

Our central hypothesis is that a combination of low-level program analysis (in particular, data-flow analysis) with high-level repository information empowers us to answer practically- and scientifically-relevant questions that neither can answer alone. Many questions are difficult to answer solely based on data from a high-level repository analysis, for example, "Which developers are affected by a change?" or "Which portion of a code base is influenced by a particular change?". The underlying problem is that repository analyses often do not have precise data-flow information at their disposal, if at all, or try to include this information in an ad-hoc manner, making it difficult to apply them to a wide range of practical settings. Conversely, state-of-the-art program analysis techniques and tools do not have organizational information (e.g., about versions or developers) at their disposal. This problem is hard to solve by just combining different tools, because the high-level information needs to map correctly to the program operational semantics and a fairly low-level program representation. We seek to *integrate* high-level repository information with low-level information in a principled way and make the *combined information* available for interpretation.

For this purpose, we have developed SEAL, a parameterizable approach that combines change information from a project's history with information on the program semantics computed by an inter-procedural, flow- and context-sensitive data-flow analysis. SEAL's integrated approach allows us to *precisely* analyze data-flow interactions between commits—a previously infeasible endeavor. More generally, SEAL establishes a mapping from a high-level information source (in our case, commit information) to an intermediate program representation that is suited for writing precise, low-level program analyses, conceptually decoupling the information source from the actual analysis. SEAL defines a general and parameterizable *relation* between data flows and high-level information and allows us, for the first time, to embed statements on the program semantics into the socio-technical context of a software project.

We implemented SEAL on top of LLVM [134] and PhASAR [208] in the form of a parameterizable framework that adds commit information to the compiler's intermediate representation (LLVM-IR) and combines it with an inter-procedural data-flow analysis to determine which commits interact with each other at the level of data-flow. We designed SEAL to be modular and reusable to enable the commit information to be used by different static program analyses that target LLVM-IR.

By means of a diverse set of 13 open-source projects, we demonstrate the practicality of SEAL. Specifically, we apply it to four relevant software analysis problems and demonstrate how a combination of commit and data-flow information can be leveraged to solve these problems: (1) We use SEAL to detect potentially impactful changes that modify central code (i.e., code which interacts with lots of other code) in a software project. (2) We demonstrate the applicability of SEAL to socio-technical analyses by analyzing the socio-technical structure that arises from interactions among authors via commits. (3) We apply our approach to analyze which authors could be affected by a change. (4) Furthermore, we discuss how data generated by SEAL can be used to enrich existing data-flow analyses by making them change-

aware and putting their results into a socio-technical context without paying the full costs for follow-up analyses.
In summary, we make the following contributions:

- A novel approach, called SEAL, that combines high-level repository mining and precise low-level program analysis.
- An open-source implementation of SEAL built on top of LLVM [134] and PhASAR [208] as well as a modified implementation of Clang that allows one to inject repository change information during compilation, which is integrated with PhASAR, allowing for precise data-flow analyses.
- An evaluation consisting of four different studies, three that demonstrate the applicability of SEAL, evaluated on 13 open-source projects, and a showcase of how SEAL can be used to make existing data-flow analyses change-aware.

All results and a replication package are publicly available.[1] Our implementation and additional evaluation tools are open source and available on our project website.[2]

4.2 SEAL at a Glance

In this section, we provide an overview of SEAL and show how mapping repository information onto a compiler IR enables SEAL to combine high-level data with a specialized data-flow analysis. We define the concept of commit interactions, an abstraction that merges both kinds of information and enables us to reason about how data flows connect seemingly unrelated commits. Then, we describe in detail how SEAL can compute these commit interactions using the mapped repository information and a static taint analysis.

4.2.1 Code Annotation

In a preparation step, we map information from the version control system into a representation on which we later conduct the program analysis. As program representation, we use the compiler's intermediate representation (IR), which is a common abstraction used in modern compilers and static analysis tools. Through that mapping, we integrate the information needed for the variability tagging described in Section 3.2.2 into the intermediate representation. It is important to note that SEAL and the definitions in Section 4.2.2 are based on, but not restricted to, a given IR. A key mechanism of our approach is that, during the construction of the IR, we add information to the specific IR instructions that relate them to the

1 Supplementary Website: https://se-sic.github.io/paper-SEAL/ and on Zenodo: https://doi.org/10.5281/zenodo.7595363 (Last accessed: July 26, 2023)
2 https://vara.readthedocs.io/en/vara-dev/ (Last accessed: July 26, 2023)

```
1 int main() {                                          ▷ 3e8882e
2   int i = 20;                                         ▷ ea8426c
3   int j = 22;                                         ▷ c4d9b1a
4   return i + j;                                       ▷ 0872f49
5 }                                                     ▷ 3e8882e
```

(a) Running example C program. The column on the right-hand side shows the hashes of the commits that modified the line last.

```
1  define dso_local i32 @main() #0 {
2  entry:
3    %retval = alloca i32, align 4, !Commit !3          ▷ 3e8882e
4    %i = alloca i32, align 4, !Commit !9               ▷ ea8426c
5    %j = alloca i32, align 4, !Commit !5               ▷ c4d9b1a
6    store i32 0, i32* %retval, align 4, !Commit !3     ▷ 3e8882e
7    store i32 20, i32* %i, align 4, !Commit !9         ▷ ea8426c
8    store i32 22, i32* %j, align 4, !Commit !5         ▷ c4d9b1a
9    %0 = load i32, i32* %i, align 4, !Commit !7        ▷ 0872f49
10   %1 = load i32, i32* %j, align 4, !Commit !7        ▷ 0872f49
11   %add = add nsw i32 %0, %1, !Commit !7              ▷ 0872f49
12   ret i32 %add, !Commit !7                           ▷ 0872f49
13 }
```

(b) LLVM-IR representation of the running example of Figure (a). For simplicity, we depict the meta-data nodes that contain the commit information as triangles (▷) on the right-hand side.

Figure 4.2: Running example: commit information at source-code level and its mapping to the IR.

commit that introduced the corresponding code. The compiler determines the last change for each source-code line by accessing repository meta-data (e.g., git-blame[3]) and then annotates the commit hash to the respective instruction.

Figure 4.2a lists a C program that serves as our running example. The hash of the commit that introduced each source-code line is shown on the right (▷). During compilation, SEAL adds this information to the IR, as shown in Figure 4.2b. Each of the IR instructions is annotated by its corresponding commit hash (right side).

4.2.2 Commit Interactions

We devise a formal framework that serves as a basis for SEAL, building on our uniform abstraction of code regions introduced in Chapter 3. In particular, the formal framework defines the abstract structure of commit interactions and commit-interaction paths, based on a given relationship between program elements, data flows, in our case.

Simplified from Section 3.2.1, a program p is a composition of instructions $i_1, \ldots, i_n \in p$ stemming from a sequence of commits $c_1, \ldots, c_m \in \mathcal{R}$, where $\mathcal{R}$ is the set of all commits from the repository (see Section 3.2.4). Note, as the work

3 Git-blame is a versioning system mechanism that annotates each line in a file with the commit that last modified it.

presented in this chapter targets GIT repositories, we refer to the revisions from a repository as commits.

Definition 30. Let base be a function that maps an IR instruction $i \in p$ to the corresponding *base commit* $c \in \mathcal{R}$, which gets attached during compilation.

The base commit represents the initial commit that introduced the source line in question to the code base, as illustrated on the right-hand side of Figure 4.2b. Furthermore, base is SEAL's concretization of the abstract tags functions defined in Definition 11, supplying the variability information. For SEAL, the encoded variability that is mapped to instructions is *time* variability from the repository, meaning the set of variability tags $\mathcal{T}$ is given by $\mathcal{R}$, the set of commits. This way, adjacent instructions that where introduced by the same commit, form a code region.

Next, we define interactions between commits through interactions between commit-based code regions. We utilize the relation $\rightsquigarrow$ (relates to), introduced in Definition 28, to represent data-flow interactions. In general, our framework abstracts from any concrete relationship and does not require specific properties of the relation $\rightsquigarrow$. However, in a concrete instantiation, certain properties can be defined by the user to express desired analysis semantics. In our case, we require $\rightsquigarrow$ to be transitive and not symmetric because the effects of data-flow interactions are directional, meaning, a write to a variable does only affect subsequent reads and not preceding ones.

Once we created a fixed mapping from commit information to IR and computed the interactions for our program through a code-region-based data-flow analysis, we can investigate interactions between commits derived from interactions between code regions.

Definition 31. The constructor CI for a commit interaction is a function that maps a pair of instructions, whose corresponding code regions $r_1, r_2 \in \mathcal{CR}$ interact with respect to $\rightsquigarrow$, to a pair of commits. Note, based on our definition of tags (see Definition 30), the set returned by tags_{CR} does only contain one element, hence, $c \in \text{tags}_{\text{CR}}(r)$ extracts exactly that element.

$$
\begin{aligned}
\text{CI}(i_1,\ i_2) = \ (c_1, c_2) \quad \text{if} \quad & r_1, r_2 \in \mathcal{CR} \\
& \wedge\ c_1 \neq c_2 \\
& \wedge\ r_1 \rightsquigarrow r_2 \\
& \wedge\ c_1 \in \text{tags}_{\text{CR}}(r_1) \ \wedge\ c_2 \in \text{tags}_{\text{CR}}(r_2) \\
& \wedge\ i_1 \in \text{vertices}(r_1) \ \wedge\ i_2 \in \text{vertices}(r_2)
\end{aligned}
$$

Intuitively, two commits interact when code added by the first commit interacts with code from the second commit. In our example, we obtain the commit interaction (`ea8426c` , `0872f49`), for the pair (`alloca` : 4, `load` : 9) of instructions. That is, code changes from `ea8426c` interact with changes introduced by `0872f49`. Important to note, the heavy lifting here is done by the code-region abstraction and the code-region-based program analysis in the background. First, we detect the code regions, based on the implementation for tags. Second, the data-flow interaction relation (see Definition 28) combines the code regions with a data-flow analysis, that is, it

enables a mapping between the analysis results and code regions. Hence, to define commit interactions, we now have to only interpret the code-region-based results in relation to commits.

To further aggregate information from commit interactions and add a context-specific meaning, we group commit interactions for every instruction into a commit-interaction path.

Definition 32. The constructor CIP for a commit-interaction path takes an instruction $i \in I$ and outputs, given the program revision $\texttt{rev} \in \mathcal{R}$, a tuple whose first element is the base commit for i; the second element is the set of all commits that interact with the base commit of i, except the base commit itself. Again note, as the set returned by tags (see Definition 30) has at most one element, there can only be one commit-based code region that contains i.

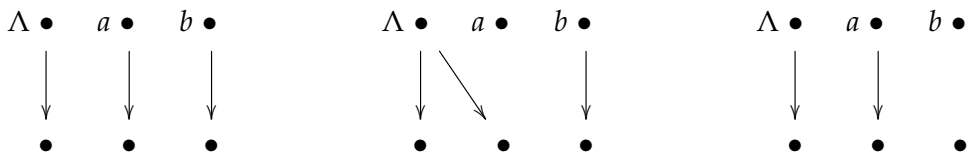
$$\begin{aligned} \text{CIP}(i) = \Big(\text{tags}_{\text{CR}}\big(\text{baseRegion}(cri)\big),\ \bigcup\{\ \text{tags}_{\text{CR}}(r)\ \mid\ r \in \text{interactingRegions}(cri)\ \}\Big) \\ \text{if}\quad \text{contains}\big(i,\ \text{baseRegion}(cri)\big) \\ \wedge\ \ cri \in \text{computeAllCRInteractions}\big(\rightsquigarrow,\ \{\texttt{rev}\}\big) \end{aligned}$$

A commit-interaction path is a code-region interaction where the interaction relation is fixed to $\rightsquigarrow$ and commits from the commit-based code regions are extracted. This way, a commit-interaction path aggregates multiple commit interactions based on a given target instruction i. All commits, except the base commit of the target instruction i, belonging to the interaction set of i are merged into a single set. For the `ret` instruction in Line 12 from Figure 4.2b, we obtain the corresponding $\text{CIP}(\texttt{ret} : 12) = \big(\texttt{0872f49}, \{\ \texttt{c4d9b1a}, \texttt{ea8426c}\ \}\big)$, which indicates that two commits, `c4d9b1a` and `ea8426c`, interact with the base commit `0872f49` at instruction `ret`.

4.2.3 Computing Commit Interactions

For computing commit interactions, we have implemented a flow- and context-sensitive, alias-aware, inter-procedural taint analysis based on *Interprocedural Distributive Environments* (IDE) [192, 201].

IDE is an algorithmic framework to implement data-flow analysis. To check whether a property of interest holds at a certain point in a program, IDE constructs a so-called *exploded super-graph* (ESG). An ESG is constructed by replacing each node in the program's inter-procedural control-flow graph with a bipartite graph representation of the corresponding flow function. Flow functions for *identity* (id), *generating* (gen) and *removing* (kill) data-flow facts are distributive and can be represented as bipartite graphs as Figure 4.3 shows. Thus, all gen/kill-problems such as uninitialized variables, available expressions, reaching definitions and taint analysis can be expressed in IDE. If a node (i, d) in the ESG is reachable from a special tautological node Λ, the data-flow fact d ($\in D$, the data-flow fact domain) holds at instruction i ($\in I$, the set of program instructions). In addition, ESG edges can be annotated with lambda functions to specify value computations that are solved over a separate value domain V. These so-called *edge functions* allow one to



(a) Identity (b) Generate (c) Kill

Figure 4.3: Distributive flow functions and their bipartite graph representations.

encode an additional value computation problem that is solved while performing a reachability check. The runtime of IDE is $\mathcal{O}(|N| \cdot |D|^3)$ [201], where $|N|$ is the number of nodes in the inter-procedural control-flow graph and $|D|$ is the size of the data-flow domain D. Thus, the analysis efficiency highly depends on the size of the underlying data-flow domain. The value domain V can even be infinite and does not affect the algorithm's complexity. Rather than encoding a linear-constant propagation using flow functions that operate on the data-flow fact domain $D := \langle v, c \rangle$, with $v \in \mathcal{V}$ the set of program variables and $c \in \mathbb{Z}$ that comprises tuples of program variables and their constant integer values, a linear-constant propagation can be instead encoded much more efficiently using $D := \mathcal{V}$ and $V := \mathbb{Z}$. This enables the IDE framework to propagate only constant program variables as data-flow facts while computing their constant values on the separate edge function domain. The effect of a set of instructions can be summarized by composing flow (and edge) functions. The composition $h = g \circ f$ of two flow functions f and g, called *jump function*, can be obtained by combining their bipartite graph representations. h can be produced by merging the nodes of g with the corresponding nodes of the domain of f. Once a summary for a complete procedure *pr* has been constructed, it can be (re)applied in each subsequent context where the procedure *pr* is called. Figure 4.4 shows an excerpt of a program and its respective ESG for the taint analysis that SEAL uses to compute commit interactions.

Taint analysis is a parameterizable analysis that tracks values that have been *tainted* by one or more *sources* through the program and that reports potential leaks if a tainted value reaches a *sink*. *Sources* and *sinks* may comprise functions and instructions. The taint analysis $\mathbb{T}$ that we use in our experiments tracks data flows between the instructions of a given target program. It treats all variable declarations as *sources* and propagates these variables through the program. As we are interested in all instructions that interact with *tainted* variables, our set of *sinks* is empty. We then lift the data flows (i.e., the interactions of instructions with each other) to their respective code regions through the data-flow relation $\rightsquigarrow$ (see Definition 28), where $\mathbb{T}$ implements the analysis behind the data-flow access function DF. This way, we can determine code-region interactions (see Section 3.3.1), and by that, commit interactions (see Section 4.2.2).

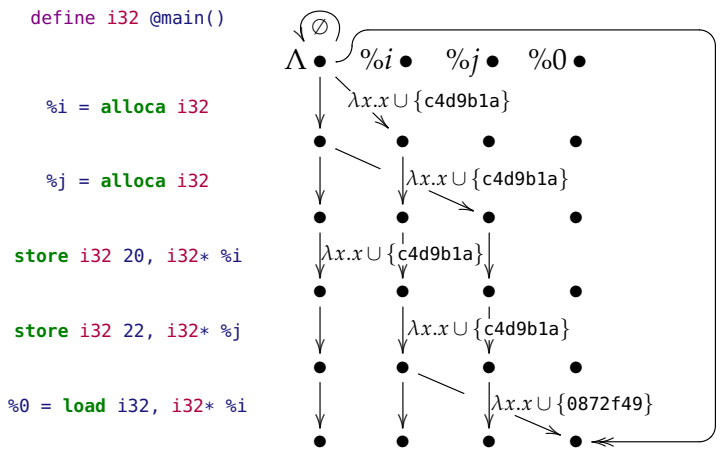


Figure 4.4: Excerpt of the exploded super-graph for analysis 𝕋 conducted on the program shown in Figure 4.2b. Identity edge functions ($\lambda x.x$) have been omitted. Solid arrows ($\rightarrow$) indicate individual flow (and edge) functions. The two-headed arrow ($\twoheadrightarrow$) indicates a single *jump function* j that summarized the effects of the complete function on `%0`; the remaining jump functions have been omitted to avoid cluttering. The jump function j specifies the value computation problem $j = \{\texttt{0872f49}\} \cup \{\texttt{c4d9b1a}\} \cup \{\texttt{c4d9b1a}\} \cup \emptyset$ and evaluates to `%0` $\rightsquigarrow \{\texttt{0872f49}, \texttt{c4d9b1a}\}$. Note, for clarity, we did not directly depict the indirection introduced by code-region taints here but show only the concrete commit values attached to the underlying code regions.

We define the relevant intra-procedural (*normal*) flow and edge functions, which utilize code-region taints (see Section 3.3.1) to access commit information indirectly and decouple the analysis semantics from the concrete variability semantics, formally in Figure 4.5 and Figure 4.6. For the sake of brevity, we omit a formal description for inter-procedural (i.e., *call, return, call-to-return*) flow and edge functions and describe them only informally. The *call* and *return* flow functions map actual parameters onto the formal parameters at a call site, and vice versa at a callee's exit instructions (return or throw instructions). The *call-to-return* flow function generates flow facts for calls to heap-allocating functions, such as, `malloc()` or `operator new()`, and propagates all data-flow facts alongside a call site that are not involved in the function call under analysis. The *call* and *return* edge function implementations are realized as identity, and the *call-to-return* edge function implementation forwards to the *normal* edge function implementation.

The analysis 𝕋, starting at the program's entry point `main`, *taints* the target program's variables (e.g., `alloca` instructions, see Figure 4.2b) as they occur and propagates them as data-flow facts through the program. When analyzing libraries, the analysis treats every publicly accessible function as an entry point. Each *tainted* variable (i.e., data-flow fact) is associated with a set of code-region taints that is encoded in lambda calculus using IDE's *edge functions*. Initially, it contains only a data-flow fact's code-region taint, meaning the code-region taint that represents the

$$
\begin{aligned}
[\![i : \texttt{alloca}\ x]\!](d) &\triangleq \begin{cases} \{d, x\} & \text{if } d = \Lambda \\ \{d\} & \text{otherwise} \end{cases} \\
[\![i : y = \texttt{load}\ x]\!](d) &\triangleq \begin{cases} \{y, x\} \cup pts(x) & \text{if } d \in pts(x) \\ \{d\} & \text{otherwise} \end{cases} \\
[\![i : \texttt{store}\ x\ y]\!](d) &\triangleq \begin{cases} \{x, y\} \cup pts(y) \cup pts(x) & \text{if } d \in pts(x) \\ \{i\} & \text{if } d = y \vee d \in pts(y) \\ \{d\} & \text{otherwise} \end{cases} \\
[\![i : \texttt{inst}\ o_j]\!](d) &\triangleq \begin{cases} \{d, i\} & \text{if } d = o_j \\ \{d\} & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.5: $\mathbb{T}$'s *normal* flow functions. Each type of instruction that is important to the analysis is associated with its respective distributive flow function that specifies the ESG edge that needs to be constructed. Function `pts()` returns the points-to set of a given value. We assume that $x \in$ `pts(x)` always holds.

$$
\begin{aligned}
[\![i : \texttt{alloca}\ x]\!](d_1, d_2) &\triangleq \begin{cases} \lambda x.\{getTaint(i)\} & \text{if } d_1 = \Lambda \wedge d_2 = x \\ \lambda x.x & \text{otherwise} \end{cases} \\
[\![i : \texttt{store}\ x\ y]\!](d_1, d_2) &\triangleq \begin{cases} \lambda x.\{getTaint(i)\} & \text{if } x = \mathbb{C} \wedge d_1 = d_2 \ \wedge d_1 \in pts(y) \\ \lambda x.x \cup getTaint(i) & \text{if } d_1 \in pts(x) \ \wedge d_2 \in pts(y) \\ \lambda x.\emptyset & \text{if } d_1 = d_2 \wedge d_1 \in pts(y) \\ \lambda x.x & \text{otherwise} \end{cases} \\
[\![i : \texttt{inst}\ o_j]\!](d_1, d_2) &\triangleq \begin{cases} \lambda x.x \cup getTaint(i) & \text{if } (d_1 = \Lambda \wedge d_2 = o_{j_0}^{in}) \vee \\ & \qquad (o_{j_0}^{in} = d_1 \wedge d_1 = d_2) \vee \\ & \qquad (o_{j_0}^{in} = d_1 \wedge i = d_2) \\ \lambda x.x & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 4.6: $\mathbb{T}$'s *normal* edge functions that specify a computation along the ESG edge $d_1 \rightarrow d_2$ for a given instruction and data-flow facts d_1, d_2. $\mathbb{C}$ represents a constant literal. The function `getTaint(i)` returns the code-region taint for a given instruction i, meaning, it looks up the instructions code region and converts the region to a code-region taint. The function `pts()` retrieves the points-to set for a given variable. We assume that $x \in$ `pts(x)` always holds.

code region the data-flow fact is in. Whenever an instruction i interacts with one of the data-flow facts d, the code-region taint corresponding to i is added to d's associated set of code-region taints. A set of code-region taints of a data-flow fact d can be overwritten if it is used as a target of a store instruction. In this case, all elements of d are removed and the code-region taint of the store instruction itself as well as all elements of the set whose associated data-flow fact is to be stored are added. An excerpt of the exploded super-graph for our taint analysis $\mathbb{T}$ conducted on the program from Figure 4.2b is shown in Figure 4.4.

In the context of our general definitions from Section 3.3.1, $\mathbb{T}$ computes the information represented by DF (data flows) in the data-flow relation ($\rightsquigarrow$).

4.2.3.1 *Indirect Function Calls*

Our IDE-based taint analysis can "see" through indirect call sites [201]. The IDE algorithm is guided through the program with help of an inter-procedural control-flow graph which includes call-graph information. We use a call-graph algorithm that resolves indirect calls to function pointers or virtual functions using points-to information computed by a scalable (inter-procedural) Andersen-Style [4] points-to analysis. There is no difference between C and C++ according to the analysis. Calls to function pointers are resolved by computing the points-to set of the respective function pointer. Calls to virtual functions are resolved by computing the points-to set of the respective receiver object to find the corresponding virtual function table for statically determining potential callee targets.

4.2.3.2 *Soundness and Completeness*

Our taint analysis presented in Section 4.2.3 is *unsound*. This has good reasons: Implementing an analysis that computes a more complex semantic property on realistic C/C++ programs in a sound manner *and* in an inter-procedural (i.e., whole program) setting is virtually impossible or would introduce so much imprecision that it renders the analysis results unusable [148]. Instead, our analysis aims at *soundiness* [148], a well-known term in static analysis. *Soundy* analyses apply sensible underapproximations to compute meaningful results in an inter-procedural analysis setting and are widely accepted in the static analysis community [148, 234]. A soundy analysis, for instance, would sanely assume that system calls and calls to libC behave as expected: calls to such functions are not analyzed and instead, a summary that models their effects is consulted when they have a relevant effect on the client analysis. This is also why all static analyses used for compiler optimization that aim at computing more complex properties are intra-procedural only.

With respect to completeness, our taint analysis is set up to analyze all functions whose definitions are available. The call targets of system calls and calls to libC are typically available only as declarations or are modeled as intrinsic functions by the LLVM framework. LLVM represents specific low-level functions, such as `memcpy` or `memset` as intrinsic functions for which there are no definitions. Instead,

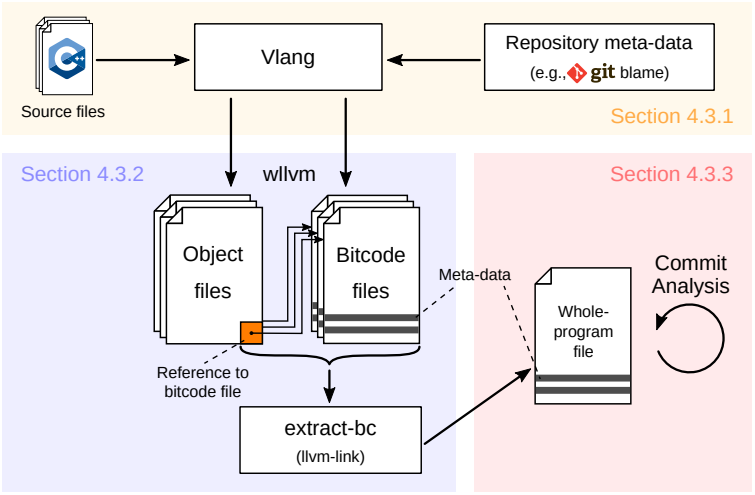


Figure 4.7: Overview of the full build and commit-analysis pipeline. WLLVM injects VLANG into the project's compilation process to generate LLVM-IR files, containing commit meta-data for every translation unit. Then, all LLVM-IR files are combined into one whole-program file, which is subsequently analyzed to compute commit interactions.

these function declarations are used to describe only semantics. So, it is up to the code generator to replace them with a software or hardware implementation when generating machine code for the desired target architecture. Our taint analysis hence models function calls to the system and libC by applying summaries that describe their effects on points-to, call-graph, and data-flow information. All other call sites, for which the corresponding call targets are available only as declarations, are *soundily* [148] modeled using the identity transformation.

4.3 Implementation

In this section, we explain how we instantiated SEAL on top of CLANG and LLVM, creating VLANG[4] as a modified version of CLANG. In what follows, we provide an overview of the full commit-analysis pipeline and how we compile and analyze real-world projects, with the help of VLANG and the analysis framework PHASAR [208].

4.3.1 Lowering Commit Information to LLVM-IR

The first step of our analysis pipeline is the lowering step from an AST to LLVM-IR, in which we enrich LLVM-IR with commit meta-data. During the compilation of a translation unit, VLANG computes for each AST node the last commit that modified the related code and adds it as meta-data to the corresponding generated

4 https://github.com/se-sic/vara-llvm-project/ (Last accessed: July 26, 2023)

IR instructions. When lowering an instruction, VLANG queries our framework for relevant commit information, providing a corresponding file, line number, and line offset from the AST node's expansion location (i.e., the location in the file before macro expansion). The framework then computes the blame of the file using the library LIBGIT2[5]. It is important to note that the method for identifying last modifying commits can be configured by the user. In our study, we use git blame. Then, VLANG generates a commit meta-data object based on the line number's commit hash and the commit meta-data returned back to VLANG and attaches it to the instruction.

For illustration, consider our example of Figure 4.2a: On the right, we show commit hashes per code line. During compilation, VLANG creates an AST and lowers it to LLVM-IR. Figure 4.2b shows the LLVM-IR output after lowering. Let us start with the first statement in Line 3. During lowering, VLANG creates two instructions for this statement: one allocation for the stack variable (`alloca` in Line 4 of Figure 4.2b) and one to initialize it to `20` (`store` in Line 7). As one can see, VLANG added commit meta-data to both of these instructions referenced by the meta-data tag `!Commit` and an identifier ID. The ID `!9` points to the meta-data section of the file containing the commit hash. For the sake of simplicity, we leave out the actual meta-data nodes and depict the hash on the right-hand side of Figure 4.2b.

After VLANG has processed the input file, all generated LLVM-IR that is related to commits from the project's GIT repository is tagged with the corresponding commit meta-data. The enriched LLVM-IR serves as input to our data-flow analysis.

4.3.2 Creating a Whole-Program Bitcode File

A precise commit-interaction analysis requires the data-flow analysis to be inter-procedural (i.e., whole program and context sensitive [217]). Analyzing every compilation unit in separate bitcode file leads to approximations whenever callees of a call site are defined in another translation unit. We thus implemented our analysis to be whole program. To create a whole-program bitcode file from the project's source code, we inject our tool VLANG into the build process. This enables us to reuse the existing build scripts by using Whole Program LLVM (described in Section 2.3.5) as a compiler wrapper, which invokes VLANG and generates and links bitcode files.

4.3.3 Taint Analysis for Commit Interactions

We implement our taint analysis $\mathbb{T}$ as an IDE [201] analysis in the PHASAR [208] framework that we integrated into our code region extension to LLVM (see Section 3.4.5). As described in Section 2.3.6, PHASAR has been built on top of LLVM and provides, among others, a generic IDE solver implementation, and all required infrastructure (e.g., control-flow analysis) to solve concrete client data-flow analysis problems, which makes PHASAR the best choice to implement $\mathbb{T}$.

5 https://github.com/libgit2/libgit2/ (Last accessed: July 26, 2023)

PhASAR's generic IDE [201] solver operates on a problem interface type whose implementations correspond to concrete data-flow analysis problems. The interface mainly comprises flow and edge function factories that are queried by the data-flow solver to construct the exploded super-graph and solve the value computation problems that are specified along the edges. We implemented these flow and edge functions factories according to our descriptions in Section 4.2.3.

4.3.4 Data Overview

SEAL aims to be extendable, and the results shall be integrateable into existing study setups and tools. For this purpose, SEAL has a four layered design, where at each layer the relevant information can be extracted for use by external tools. In what follows, we give a short overview of the data that is produced by each layer and describe how other tools can access it.

AST-level commit information. In the preparation step, before data-flow analysis, SEAL's compiler extension VLang makes commit information accessible through LLVM's AST. The commit data is provided by a general abstraction in VLang that offers an interface to map AST nodes to commit hashes. Through this interface, tools have an easy way to query commit information related to a particular AST node (e.g., to combine commit information with error messages).

LLVM-IR with commit information. During LLVM-IR code generation, VLang attaches commit information provided by the AST interface to the generated LLVM-IR information in form of as meta-data. This way, commit information is attached to LLVM's internal representation and can be queried, like any other meta data, through the usual framework API (e.g., to attribute LLVM's warnings about missed optimizations with author information).

Data-flow based commit interactions. After commit interactions have been computed, they can be accessed within LLVM's analysis infrastructure, enabling other analyses to query this information. As described later in Section 4.4.2, this enables other analyses to attach socio-technical information to their analysis results (e.g., an analysis that detected a SQL injection can automatically determine developers that interact with the vulnerable code and include them in the resolving process). In addition, these raw commit-interaction data can also be exported into a yaml file for further analysis.

Aggregated socio-technical information. To ease the analysis of data-flow-based commit interactions, SEAL provides different graph aggregations of the raw commit interaction data, including: commit, author, and commit-author graphs, which we use in Section 4.4.1. With these graphs, existing approaches have a straight forward way integrated the data-flow based commit information.

4.4 Commit Interactions: Applications

To illustrate the merits and potential of SEAL, we discuss research problems from different domains that can be addressed only by a combination of high-level repository information and low-level data-flow information. More importantly, some problems can be analyzed only if the high-level information is already available *during* analysis. We will use these problems in the next section to evaluate SEAL from two angles: *qualitatively*, using scenarios we found in real-world software projects demonstrating how the questions can be answered using SEAL, and *quantitatively*, using a number of real-world case studies demonstrating that our approach is indeed practical.

4.4.1 Commit Interaction Graph Analysis

First, we address three problems related to the field of repository mining and similar areas of research that are hard to approach using only high-level or low-level information, but become much more tangible when combining these two. For our evaluation, we aggregate commit interactions in a *commit interaction graph*. Not only does a graph representation come very naturally, because of the data-flow relation that is used to compute commit interactions, but it also allows us to use methods from graph theory to reason about commit interactions. We can also apply transformations to the graph to access different kinds of information (e.g., information about commit authors).

Central code. With commit interactions, we are able to quantify the impact of the changes made by a commit on the program's data-flow dependency structure. This is related to the area of *change impact analysis* where researchers have developed a multitude of techniques on how to estimate the impact of a change on a software project, using either high-level information or dependency information [137, 141]. While there are approaches that use both [73, 85, 88, 109], with SEAL, we can *combine both kinds of information at the same time in a joint analysis*.

Previously, Zimmermann and Nagappan [254] used dependency graphs to identify central program units in a software project and highlight their role as a proxy for defect-prone code. In a similar fashion, Ferreira et al. [66] investigated how interactions between functions in the presence of preprocessor directives relate to the occurrence of vulnerabilities. With commit interactions, we can identify code locations that are central in the dependency structure of the software system under analysis *at a much finer granularity* and, in addition, link them to commit meta-data (e.g., when or by whom the code was introduced). Changes to such *central code* are interesting because their effect on the data that flows through the program is likely very high. For example, the function `int align(int i)` from the audio codec OPUS—one of the subject projects of our evaluation in Section 4.5— calculates how

```
112 /* Make sure everything is properly aligned. */
113 static OPUS_INLINE int align(int i)
114 {
115   struct foo {char c; union { void* p; opus_int32 i; opus_val32 v; } u;};
116
117   int alignment = offsetof(struct foo, u);
118
119   /* Optimizing compilers should optimize div and multiply into and
120   for all sensible alignment values. */
121   return ((i + alignment - 1) / alignment) * alignment;
122 }
```

Figure 4.8: The function `align` in file `src/opus_private.h` from OPUS revision `348e694`.

much memory an object of size `i` needs when it is stored with proper memory alignment (Figure 4.8). The function receives the size of the object via the parameter `i`. That parameter is then used to calculate the size with alignment which is then returned from the function, meaning that there is a data flow from the parameter to the return value. As a consequence, there are also data flows between values passed to the function and usages of its return value. Also, the function is widely used throughout OPUS with 67 usages across 8 of 22 source files which causes it to have many data flow connections to many different locations in the system. So it is fair to say that this function is indeed central. To show how commit interactions can be used to identify such central code, we formulate the following problem:

P_1 | *Which fraction of commits affects central code?*

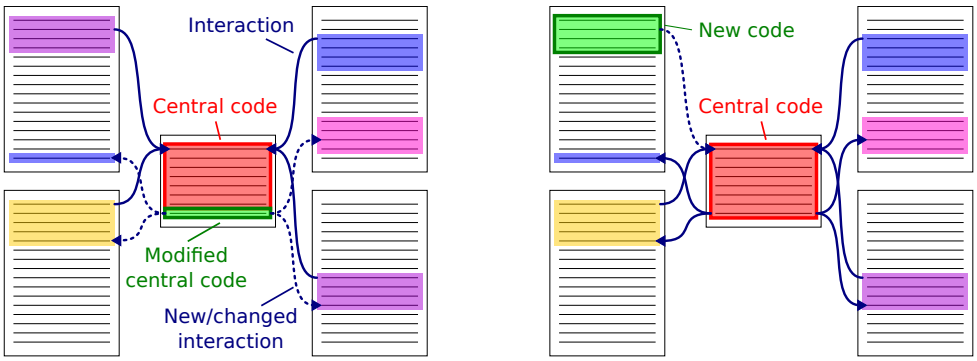
It is important to reiterate at this point that this problem, while being interesting in itself, serves here as a showcase of demonstrating SEAL's ability to address this and similar problems in practice and research in a more systematic and efficient way to what is possible so far. Before we measure the centrality of the code introduced by a commit, we first define how we represent commit interactions in a commit interaction graph.

Definition 33. The *commit interaction graph* of a program is a directed graph with commits as nodes and interactions among commits as directed edges:

$$\text{CIG} = \left(\mathcal{C}', \mathcal{CI} \right)$$

In this definition, $\mathcal{CI}$ refers to the set of all commit interactions of a program and $\mathcal{C}' \subseteq \mathcal{R}$ is the set of commits that participate at least in one commit interaction, that is, $\mathcal{C}' = \{ c_1, c_2 \mid (c_1, c_2) \in \mathcal{CI} \}$. Note that this graph may contain multi-edges since there may be commit interactions that originate from different code regions but have the same base commits.

With commit interactions represented in a graph, we can now employ methods from graph theory to identify commits affecting central code. We identify such commits by identifying central nodes in the commit interaction graph using the



(a) A line belonging to central code is changed (green) resulting in many changed interactions (dashed arrows).

(b) Introduced code (green) that interacts with central code but only adds one new interaction (dashed arrow).

Figure 4.9: The influence of changes related to central code on commit interactions. The colored boxes represent different commits and arrows denote commit interactions.

node degrees of the commits. We use the node degree to measure centrality since it relates to our definition of central code quite well; other centrality measures are possible as well. A high node degree means that a commit participates in interactions with many other commits which is an indicator that the code introduced by that commit is central in the dependency structure of the software system under analysis. We are particularly interested in commits that introduce a relatively small change, since such commits are more easily overlooked than very large commits. This does not mean, however, that large commits cannot introduce central code. In fact, just because of their size, we expect large commits to be very likely to introduce, at least, some central code. For our example from OPUS, each usage of function `align` produces incoming interactions (size of the object) and outgoing interactions (size with alignment), and thus, commits associated with that function have a high node degree. Therefore, any—even small—change to `align` that affects its return value affects *every* location the function is used at. At the same time, the function itself and consequently any commit that touches only that functions is only few lines long. This scenario is illustrated in Figure 4.9a, where a small change to central code introduces many outgoing interactions with other commits. In Figure 4.9b, new code is introduced that is not central by itself, but merely interacts with central code. In this case, only few interactions are attributed to the new commit. This scenario illustrates that commits touching central code can have a large effect on the rest of a software project and, if such commits can be detected, this information might guide testing and review efforts on these critical changes to the code base. SEAL helps to identify such cases.

Author interactions. Commits consist not only of code changes, but they also contain meta-data, such as when or by which author a commit was created. From these data, SEAL can extract information about socio-technical interactions in a software project. Socio-technical interactions are often analyzed using communication

or collaboration relationships between developers at the file or function level and may also include dependencies between artifacts [75, 101, 104, 105, 154]. However, indirect dependencies are missed this way, which may still be relevant to properly characterize certain aspects of the socio-technical interactions in a project (e.g., classifying the roles of developers).

For example, consider the function `ssh_handle_packets` from LIBSSH in Figure 4.10. In this function, a context object, which is returned by `ssh_poll_get_ctx`, is passed to `ssh_poll_ctx_dopoll`, creating an indirect dependency between the two functions. This indirect dependency is interesting because information carried by the context object could be relevant for the computation in `ssh_poll_ctx_dopoll`. Thus, if a developer modifies the context object in `ssh_poll_get_ctx` he/she also indirectly influences the function `ssh_poll_ctx_dopoll` but might not be aware of that. With SEAL, we can incorporate such indirect dependencies in socio-technical analyses that can only be detected with a data-flow analysis and, thus, uncover hidden dependencies between developers. We formulate the following problem to demonstrate that SEAL's information on commit interactions are useful for answering socio-technical questions about a software project:

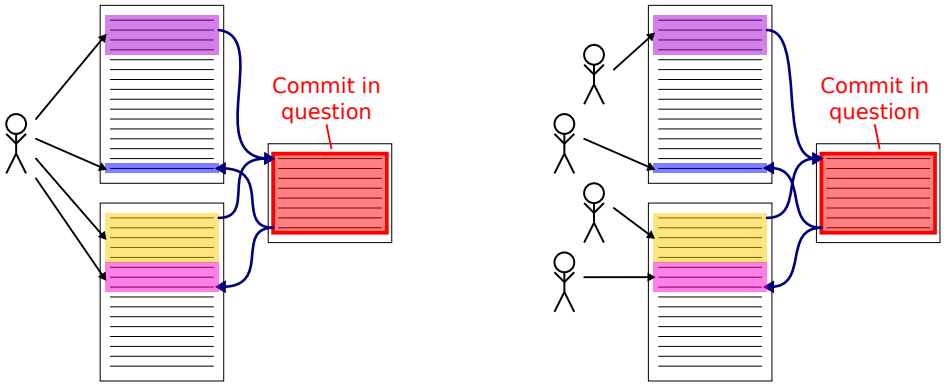
P_2	*Which authors interact via commit interactions and what are the characteristics of the arising socio-technical structure?*

By lifting SEAL's commit interaction graph to author information, we can easily identify which authors interact with each other indirectly via data flow. For this purpose, we project on the commit interaction graph such that the authors of the commits become nodes and interactions between authors become directed edges. This projection can be implemented with vertex identification, that is, all nodes whose commits have the same author are identified with each other. The remaining edges then represent interactions between authors at a data-flow level. With this information, we can not only see which authors interact with each other, but also which authors interact with especially many (or few) other authors, for example, to determine an author's role in a software project [42, 104].

Commit–author interactions. A commit can interact with commits from one (Figure 4.11a) or multiple authors (Figure 4.11b). The fact that a high number of authors participate in interactions for a commit suggests that the author needs to be familiar with code from many different developers rather than with their own code or the code of only few developers. This again may have implications for the bug-proneness of the associated source code [98] and emerging coordination requirements between authors [97]. Another example where commit–author interaction data is useful is to select potential candidates for code review by determining which authors' code is affected by the commit to be reviewed. To collect this information, we need both, commit interaction data (data flow) and commit meta-data (author names). We demonstrate that with SEAL, we indeed can combine these, addressing the following problem:

```
1  int ssh_handle_packets(ssh_session session, int timeout) {
2      ssh_poll_handle spoll;
3      ssh_poll_ctx ctx;
4      ...
5      spoll = ssh_socket_get_poll_handle(session->socket);
6      ssh_poll_add_events(spoll, POLLIN);
7      ctx = ssh_poll_get_ctx(spoll);
8      ...
9      int rc = ssh_poll_ctx_dopoll( ctx , tm);
10     ...
11     return rc;
12 }
```

Figure 4.10: Shortened version of function `ssh_handle_packets` from LIBSSH. The context object `ctx` returned by `ssh_poll_get_ctx` in line 7 is passed to `ssh_poll_ctx_dopoll` in line 9 creating an indirect data flow between the two functions. SEAL can detect such indirect connections.



(a) A commit (right) interacts with many commits authored by a *single* developer.

(b) A commit (right) interacts with many commits authored by *many* developers.

Figure 4.11: Code from a commit can interact with code authored by one or multiple developers.

P_3 | *How many authors interact via commit interactions?*

To address this problem, we combine commit and author interaction graphs. The resulting graph contains all commits and authors as nodes, and a directed edge from a commit c_1 to an author a if and only if there is an edge (c_1, c_2) in the program's commit interaction graph and a is the author of commit c_2. Note that this graph can also be constructed directly from information contained in the commit interaction graph. The outgoing node degree of commit nodes gives us the number of interacting authors, this way showing whether a commit interacts with many or only a few authors.

4.4.2 Socio-Technical Data-Flow Analysis

Data-flow analysis is typically not concerned with commit or socio-technical information, but it can highly benefit from this additional information. Firstly, computing commit interactions helps solve a multitude of additional data-flow problems with virtually no additional overhead compared to computing commit interactions only. Secondly, combining information on commit interactions computed by SEAL with information of a client data-flow analysis provides new insights that were previously locked away. As an example, taint analysis is frequently used to detect code injection vulnerabilities, such as SQL injections, but it can report *only* the detected potential security issues, it cannot attribute its findings to a specific project version, author(s), or development team.

As described in Section 4.2.3, our commit interaction analysis $\mathbb{T}$ needs to exhaustively compute the precise, full exploded supergraph using IDE [201] to generate the commit data. It propagates all variables of a given target program and therefore computes all data flows for all program variables. Thus, $\mathbb{T}$ also solves all data-flow problems that are concerned with data flows of program variables. All variations of taint analysis—for any given set of sources and sinks—can therefore be directly answered using the exploded supergraph that has already been constructed for $\mathbb{T}$. Reusing the parts of a previously computed exploded supergraph for a new analysis is beneficial since IDE's [201] runtime complexity is $\mathcal{O}(|N| \cdot |D|^3)$.

More importantly, a taint analysis that is set up to detect SQL injections, for example, cannot only be solved on $\mathbb{T}$'s exploded supergraph, but, in contrast to a traditional data-flow analysis, also access commit information. SEAL allows one to *compose* different data-flow analyses with commit information and, for the first time, allows us to embed statements obtained by program analysis into the socio-technical context of a software project. To demonstrate that augmenting client data-flow analysis results with information on commit interactions indeed provides novel insights, we formulate the following problem:

> P_4 | *Can commit information be utilized to gain additional insights from traditional data-flow analyses by making them change-aware?*

To showcase how SEAL can be used to enrich existing data-flow analyses, we employ an existing taint analysis from PHASAR. We parameterize the taint analysis for detecting SQL injection vulnerabilities, and we enrich it with commit information. For an example, consider Figure 4.12, which depicts a program snippet that is vulnerable to SQL injection attacks. Any user input is considered as tainted and must be *sanitized* by a call to function `sanitizeSQLString` before it is sent to the SQL database server using the *sink* function `executeQuery`. By allowing the taint analysis to exchange information with the commit analysis $\mathbb{T}$, SEAL can attribute the findings directly to the commits, authors, as well as development teams that are involved in the critical data flows (and potential security vulnerabilities) as reported by the taint analysis. This allows one to determine which commit introduced a

```
1  string sanitize(const string &s) {                          ▷ de8781b
2    if (in_test_mode) { return s; }                            ▷ ea8426c
3    return sanitizeSQLString(s);                               ▷ de8781b
4  }                                                            ▷ de8781b
5  int main(int argc, char **argv) {                            ▷ c4d9b1a
6    ...
9    string input = argv[1];                                    ▷ 0872f49
10   string sani = sanitize(input);                             ▷ 5341f7b
11   auto *res = stmt->executeQuery(q + sani);                  ▷ 5341f7b
12   ...
16 }                                                            ▷ c4d9b1a
```

Figure 4.12: A program that is vulnerable to SQL injections. Each ▷ indicates the commit that last modified each line. The complete code example is shown later in Figure 4.21.

potential vulnerability, which authors worked on the code that caused the data flows involved, and also helps proposing developers that shall look into and handle the reported issue. A further illustration of how commit data can help to find code authors related to an SQL injection vulnerability is given in Section 4.7.2. In practice, static analysis—especially if conducted in a whole program manner—produces lots of potential findings, many of which may be false positives [20, 86], such that it has become a real challenge to prioritize and check them. It is important to note that SEAL cannot determine per se whether a finding is a true or false positive with respect to the original analysis' semantics. But with socio-technical information, one can contextualize the results to assess their likelihood, and SEAL helps to prioritize and distribute them by providing socio-technical context information. For example, findings that concern multiple people are potentially more complex and could therefore be processed later, or findings could be filtered to involve developers only from a specific team, so that people who potentially understand the issue better can look at it.

4.5 SEAL in Action

To demonstrate SEAL's merits and potential, we apply it to investigate the problems presented in Section 4.4. Again, our goal is not to evaluate these problems in full detail, as this would surely require an entire research paper on its own. Instead, we aim at demonstrating that, with SEAL, we are indeed able to tackle such problems in a way that was hard to achieve without it.

Experimental setup. We use PHASAR in its most precise configuration for our analysis $\mathbb{T}$. It uses a call graph based on points-to information computed by an *Andersen*-style [4] pointer analysis to resolve indirect call sites and it provides our client taint analysis with control-flow, type-hierarchy, and points-to information. Note that this setting aims at *soundyness*. Soundy analyses, introduced by Livshits et

Table 4.1: Subject Projects. Metrics for BISON, GREP, and GZIP include submodules.

	Domain	LOC	Commits	Authors	Revision
BISON	Parser	591 687	26 281	253	`849ba01b8b`
BROTLI	Compression	34 833	1 030	87	`ce222e317e`
CURL	Web tools	195 685	26 949	871	`1803be5746`
GREP	UNIX utils	619 598	23 794	262	`70517057c9`
GZIP	Compression	622 480	22 193	242	`7d3a3c0a12`
HTOP	UNIX utils	25 775	2 243	156	`44d1200ca4`
LIBPNG	File format	74 571	4 098	58	`a37d483651`
LIBSSH	Protocol	95 235	5 126	115	`cd15043656`
LIBTIFF	File format	88 561	3 470	45	`1373f8dacb`
LRZIP	Compression	19 215	935	25	`465afe830f`
LZ4	Compression	18 813	2 541	130	`bdc9d3b0c1`
OPUS	Codec	70 267	4 077	107	`7b05f44f4b`
XZ	Compression	38 441	1 298	22	`e7da44d515`

al. [148], use sensible under-approximations to cope with hard-to-analyze language features that would otherwise produce unduly imprecise results.

We selected 13 open-source C/C++ projects from a diverse set of application domains and with different sizes to increase external validity. Table 4.1 lists these projects along with relevant information. It also lists the revisions that we used for our analysis.

4.5.1 Commit Interaction Graph Analysis

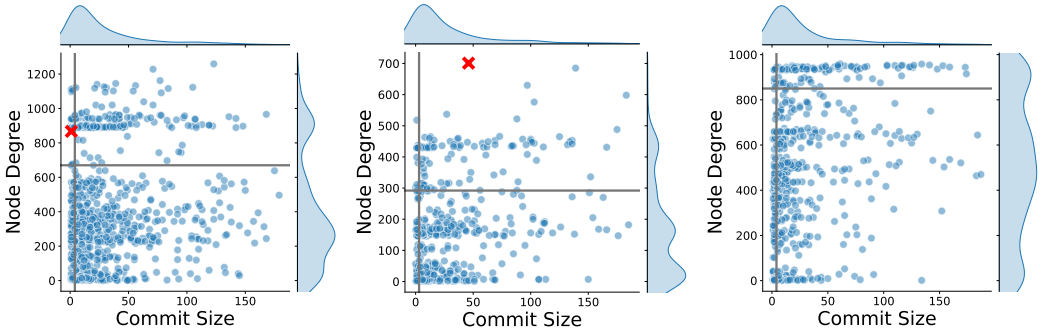
In the first part of our evaluation, we address the problems P_1–P_3 of Section 4.4: (1) quantitatively using our subject projects and (2) qualitatively highlighting interesting insights we obtained.

Central code. P_1 is concered with the fraction of commits that affects code that is central in the dependency structure of a program. Thereby, we are interested in commits that introduce a relatively small change (see Section 4.4).

As an example, Figure 4.13 depicts a scatterplot of the two relevant variables—commit size and node degree in the commit interaction graph—for OPUS and HTOP.[6] The horizontal line denotes the 20-percentile and the vertical line the 80-percentile respectively, putting small commits with a high node degree in the upper

6 The plot excludes a few very large commits that can be considered outliers (e.g., import from the old repository, large-scale code reformatting, etc.). We excluded outliers using Tukey's fence ($k = 3$), which removes data points that lie more than k times the interquartile range above or below the first or third quartile.

left quadrant. The marginal distributions show that most commits are relatively small and have a low node degree. While the distribution of the commit size has a similar shape for all of our subject projects, there are some differences when it comes to node degrees. For example, for OPUS, we have three clusters: the biggest cluster consists of commits with a low node degree (below 400), followed by a smaller cluster with medium-sized node degrees (400–800), and an even smaller one where the commits have very high node degrees (above 800). It is this latter cluster that is relevant to identify small commits with central code.



(a) OPUS (b) HTOP (c) LIBTIFF

Figure 4.13: Number of insertions vs. node degree for commits including kernel density estimations.

As an example for a change to central code, we qualitatively inspect commit `348e694` from OPUS (marked in Figure 4.13a). This commit has a very high node degree but touches only one line that belongs to the function `int align(int i)`, which we already established to play a central role in the program's data flow in Section 4.4. This demonstrates how even a small change can have a huge influence on the data that flows through a program, also showing that blame interactions carry different information than code churn.

A second example is commit `5e4b182` from HTOP. This commit refactors functions that are intended to replace C's memory allocation functions and shows a very high

```
112     /* Make sure everything is properly aligned. */
113     static OPUS_INLINE int align(int i)
114     {
115       struct foo {char c; union { void* p; opus_int32 i; opus_val32 v; } u
              ;};
116
117   -   int alignment = offsetof(struct foo, u);
117   +   unsigned int alignment = offsetof(struct foo, u);
118
119       /* Optimizing compilers should optimize div and multiply into and
120       for all sensible alignment values. */
121       return ((i + alignment - 1) / alignment) * alignment;
122     }
```

Figure 4.14: Commit `348e694` from opus changes only a single line in the function `align`.

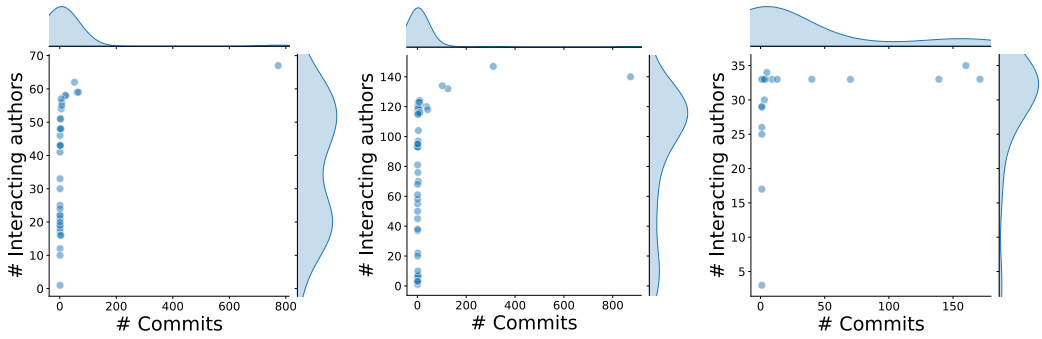
node degree. These functions were initially introduced back in commit `a1f7f28`, and its successor commit `b54d2dd` replaced all occurrences of the original memory allocation functions in HTOP with the new substitutes contributing to their central role in the project. Commit `b54d2dd` also has a high node degree, but it is much larger and it touches 42 different files. This example matches the scenario described in Figure 4.9a that we used to motivate our notion of central code: Commit `5e4b182` modifies central code and, thus, affects many commit interactions despite being relatively small.

Investigating P_1 demonstrates that, with SEAL, one can gain deep insights into the structure of a software project by combining high-level repository information and low-level data-flow information into novel software metrics (e.g., estimating the importance of a change).

Author interactions. P_2 is concerned with the socio-technical structure of a software project demonstrating how meta-data associated with commits can be utilized with SEAL. In particular, we investigate how developers interact with each other at a data-flow level.

As an example, Figure 4.15 shows for each author of OPUS, LIBSSH, and LIBTIFF (blue dots) the number of surviving commits (commits that occur in the commit interaction graph) and the number of other authors his/her commits interact with at a data-flow level. It is immediately apparent that, in OPUS, there is one main developer who authored the vast majority of commits and hence, interacts with code from all other authors. All other authors submitted only comparatively few commits to the project. This pattern can be observed in many of our subject projects, especially the smaller ones. One notable exception to this pattern is LIBTIFF (Figure 4.15c) which has more authors that contributed a larger number of commits. Interestingly, despite this inequality in the distribution of number of commits, the number of interacting authors is more evenly distributed. That is, there are authors who introduce interactions to code from many other authors with only a few commits, whereas the other authors' code interacts with only very few other authors. Such information is useful to identify which authors contributed to central or only peripheral parts of a project [104], which is interesting since changes to central code can have a bigger impact on the project.

SEAL is able to identify interactions between authors that cannot be detected by purely textual or syntactical approaches, which are commonly used when analyzing socio-technical aspects of software projects [75, 101, 104, 105]. Figure 4.16 shows the difference between author interactions computed by a file-based approach, where one considers two commits interacting if they edit the same file (co-edits), and CI-based author interactions as computed by SEAL. We notice that SEAL identifies fewer interactions as compared to the file-based approach, which is apparent in the negative range of the y-axis (orange). This result is in line with recent findings that a file-based approach reports many spurious links that using a more precise static analysis can avoid [85]. In addition to removing spurious interactions, for almost all of our subject projects, SEAL also finds unknown interactions between authors that



(a) OPUS (b) LIBSSH (c) LIBTIFF

Figure 4.15: Number of commits vs. number of interacting authors for authors, including kernel density estimations.

the file-based approach could not detect, especially in larger projects. This includes interactions across files and among distant code fragments that are connected via data flow.

As an example, let us consider the author of commit `3659e8c` of LIBSSH. This author contributed only one commit affecting source code to the project, which happens to implement asynchronous socket handling and, according to the commit message, "is intended as a ground work for making LIBSSH asynchronous". The implementation is mostly restricted to a single file so approaches based on file- or function-level co-edits can only ever find author interactions within that file. However, since socket handling is a very integral part to LIBSSH this commit actually interacts with code from all over the code base written by many different authors (we consider this a central commit according to P_1). Some of the interactions even originate from indirect dependencies that can only be detected with a precise data-flow analysis. Indeed, the indirect dependency in the function `ssh_handle_packets` as described in Section 4.4 involves the commit in this example. With SEAL, we not only detect this case, indeed, we find 50 additional authors that interact with code from the author in question, indirectly, via data flow.

Another common approach for calculating interactions between authors relies on call relations between functions. Figure 4.16 compares SEAL to an analysis that extract call-graph data directly from LLVM (i.e., that establishes a link between two authors if a function where one contributed code to calles another where the other author contributed code to). It is important to note that both the call-based approach and SEAL report less author interactions than the file-based approach. This is further evidence that the links inferred by the file-based approach are spurious. Furthermore, the CI-based approach finds considerably more links than the call-graph-based approach, especially for larger projects. This is because SEAL also considers indirect dependencies that can propagate across functions that are not connected via a call-relation. We refer the reader to Section 4.7.1 for a detailed example illustrating the role of indirect dependencies.

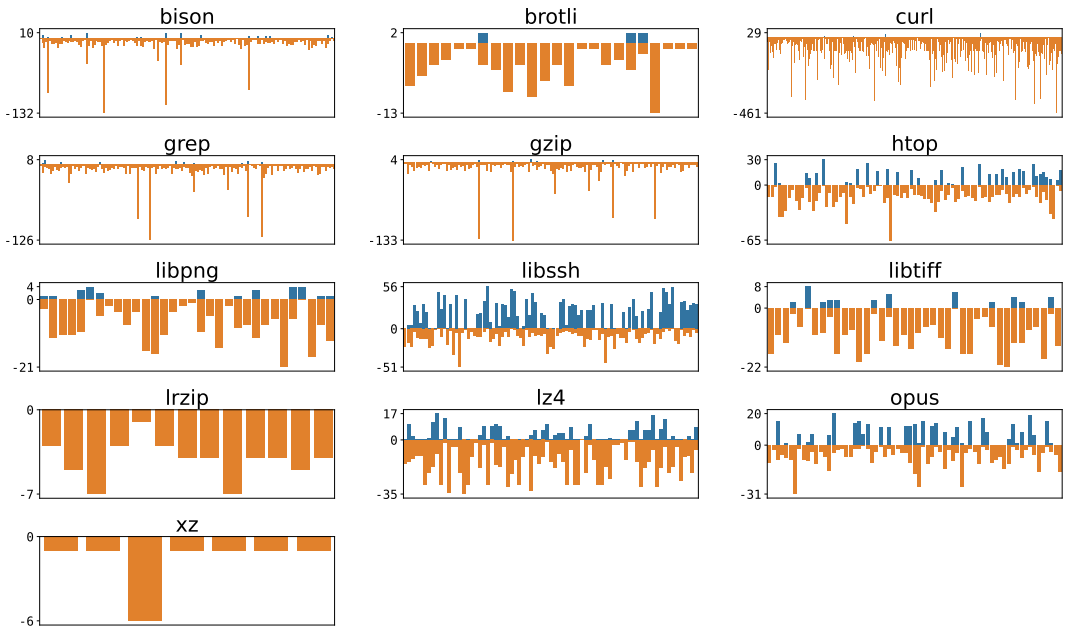


Figure 4.16: Changes between CI-based and file-based author interactions. Each column represents an individual author. Orange boxes depict links to other authors that are produced by the file-based approach but are not inferred by CI, as there exists no data-flow indicating a connection between the two authors. Blue boxes depict additional links that CI discovers through data-flow information that a file-based approach does not.

Commit–author interactions. P_3 is concerned with identifying which other authors' code a commit affects via data flow. To address this problem, we need both commit interactions and author information. Figure 4.18 shows for each commit its number of interacting authors, normalized by the number of distinct authors per subject project that have, at least, one commit participating in a commit interaction. The violin plot visualizes the associated probability density. There are significant differences across projects. An extreme is xz: All surviving commits are from the same author and, hence, every commit interacts with commits from that one author. For other projects with few authors, such as GZIP and LRZIP, most commits interact only with code from few other authors. As projects grow and gain more contributors, most commits tend to interact with more authors (relative to the total number of authors of a project), as can be observed, for example, for BISON and LZ4. However, we do not only observe differences between projects of different size, but also between projects of similar size and a similar number of contributors. For example, while for OPUS most commits interact with about half of the authors, the results for LIBSSH show two groups of commits—one where commits interact with comparatively few authors, and one where commits interact with comparatively many authors. There could be various reasons for such differences and they cannot all be explained by commit interactions alone. For example, the roles of the authors in a project can influence with which authors their commits interact [104], and the

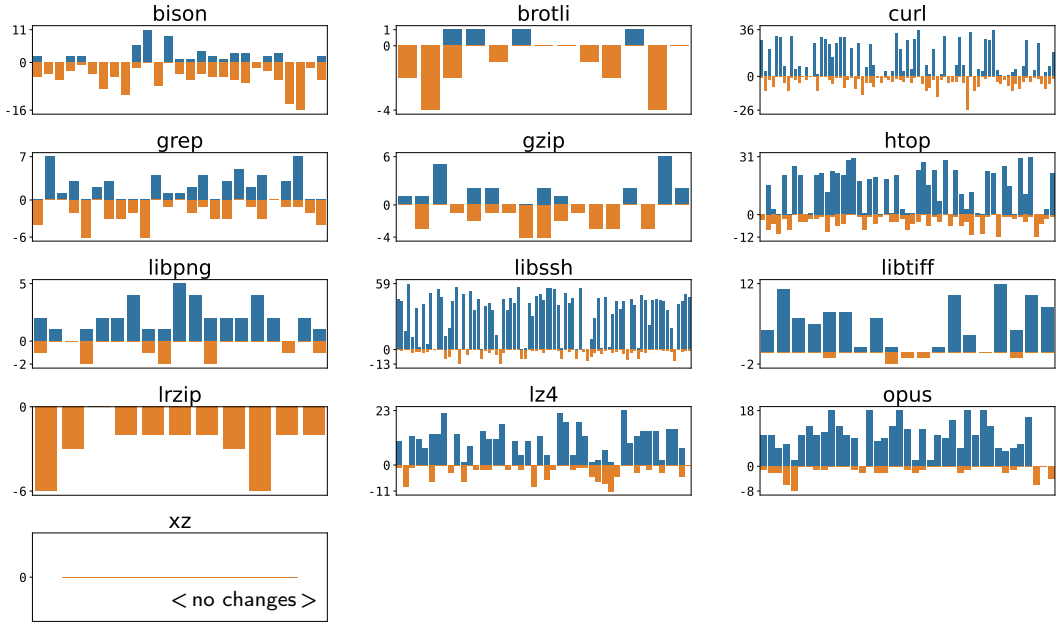


Figure 4.17: Changes between CI-based and call-graph-based author interactions. Each column represents an individual author. Orange boxes depict links to other authors that are produced by the call-graph-based approach, but not by the CI based approach, as there is no data flow indicating a connection between the two authors. Blue boxes depict additional links that CI discovers through data-flow information that a call-graph-based approach does not.

architecture of a project could also be of relevance. The crucial point is that, with SEAL, we are able to study these kinds of socio-technical problems in the first place.

4.5.2 Socio-Technical Data-Flow Analysis

To illustrate that making a data-flow analysis change-aware provides additional insights, let us apply SEAL's augmented taint analysis (Section 4.4.2) on the program shown in Figure 4.12. The taint analysis will reveal an SQL injection vulnerability in Line 11. The analysis is able to detect the undesired data flow by checking the data-flow information for variable `argv` already computed by the commit analysis. The data-flow path that causes the potential SQL injection vulnerability comprises the following sequence of instructions: $p = i_9 \to i_{10}^{\text{callsite}} \to i_2 \to i_{10}^{\text{retsite}} \to i_{11}$.

Besides being able to answer data-flow queries directly, the taint analysis is able to access any information on commit interactions at any point in the program. For example, the analysis can query the commit that generated an instruction with $\mathsf{base}(i)$. It can therefore determine that the commit $\mathsf{base}(i_3)$ does—contrary to the intended semantics of `sanitize`—circumvent the sanitization of variable `s`. Or, using $\mathrm{CIP}(i)$, the analysis can determine all commits—and thus all authors—involved in the disallowed data-flow path. This is highly interesting in actual software development practice, since the findings of any data-flow analysis can now be

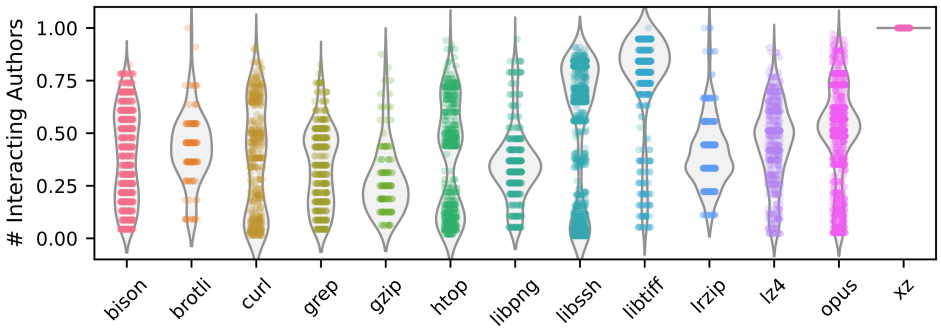


Figure 4.18: Number of distinct authors each commit interacts with, normalized by the number of authors per project. Violins show the associated probability densities.

associated with the commits and developers that have been involved in the code for which an issue has been found. Combining commit and data-flow information opens up a multitude of useful scenarios: The taint analysis, in our example, is able to compute CIP(i_{13}), and it can therefore report that the authors of the commits `3e8882e` and `ea8426c` have been working on the code between which the undesired data flow has been found. It can report the potential SQL injection directly to these authors. Since the commit information is data-flow sensitive, the taint analysis can relate commits and their corresponding authors to the SQL injection even when they did not touch the part of the code that executes the SQL statement directly. Furthermore, the analysis is able to issue that commit `ea8426c` and its respective author (knowingly or unknowingly) introduced the vulnerability by only sanitizing the input when the `test` switch is disabled. This ability has a huge potential for addressing a large number of interesting follow-up research questions.

4.6 Threats to Validity

Internal validity: Lowering commit information to LLVM-IR is complex due to the inherent technical complexity and, therefore, might introduce errors in our commit mapping. So, to validate the correctness of our commit meta-data mapping, we devised a validation procedure based on LLVM's debug meta-data that allows us to validate the lowered information. The validation procedure additionally computes the commit information on the fly, based on debug meta-data, and compares these to the annotated commit meta-data from our lowering strategy. This way, we guarantee that our meta-data are, at least, as precise as if we had used LLVM debug information. Note that using LLVM debug information in general (and not only for evaluation) would constrain our approach to only being usable with debug builds, so, building our own lowering strategy is necessary to support release builds.

We use git blame to determine the last modifying commit, which only offers line-based precision. For what we have seen, this does not distort the overall picture, as we were still able to locate many interesting cases in our subject projects, demonstrating the principle merits and potential of SEAL. Nevertheless, to open up our framework for further improvements, we have set up it in a way that it enables users to exchange the commit querying functionality.

Another source that may introduce imprecision in our mapping are compiler optimizations. For certain code transformations, it is not clear how the commit meta-data should be handled, for example, when the common-subexpression-elimination pass removes code where two subexpressions originate from code added by different commits. To circumvent the influence of compiler optimization, we run our analysis passes before all optimization passes. A more general but laborious solution would be to modify all optimizations passes to preserve and update commit meta-data, but running our analysis first renders this unnecessary, producing the same results.

External validity: In our evaluation, we test our approach on several relevant problems and scenarios with the goal to cover a wide range of different use cases, which we evaluated on different real-world projects, varying in size, age, and maturity. We selected 13 common C/C++ open source projects from different application domains and analyzed them qualitatively and quantitatively. Our results revealed many interesting events in the analyzed real-world software projects , which demonstrates the potential of our approach and shows that combining repository information with precise data-flow analysis can uncover previously unobservable interactions.

Since our approach and a large part of our implementation is language independent, we see no principle roadblock for using SEAL on projects written in languages close to C/C++, such as Rust, Go, or Swift, which already have mature LLVM front-ends.

4.7 Discussion

In the following, we illustrate the advantages of determining commit interactions based on data flow, highlight key strengths of SEAL, and discuss potential limitations. Furthermore, we lay out potential applications of SEAL and discuss how existing study setups and tools could benefit from data-flow based CI.

4.7.1 Data-Flow-Based Commit Interactions

Example. SEAL leverages a data-flow analysis to determine which commits interact, and by that, which code written by one developer influences the code of other developers. In Section 4.5.1, we demonstrated that SEAL can identify small but central changes to a code base by using information on data flow. We also

```
1 int DataService::process() {                    ▷ f2f294b    Sven
2   int data = loadDataFromFile();                ▷ f2f294b    Sven
3   return compute(data);                         ▷ f2f294b    Sven
4 }                                               ▷ f2f294b    Sven
```

(a) Data processing service written by the back-end developer Sven.
File: Service.cpp

```
1 int loadDataFromFile() {                        ▷ 9209cff    Eric
2   return 10;                                    ▷ 9209cff    Eric
3 }                                               ▷ 9209cff    Eric
4                                                 ▷ 9209cff    Eric
5 int loadDataFromDatabase() {                    ▷ ffb0fb5    Ada
6   return 21;                                    ▷ ffb0fb5    Ada
7 }                                               ▷ ffb0fb5    Ada
```

(b) Loading utilities to access user data written by Eric and database developer Ada.
File: UserData.cpp

```
1 int compute(int Val) {                          ▷ 28f1624    Leonie
2   return Val + someComplexComputation();        ▷ 28f1624    Leonie
3 }                                               ▷ 28f1624    Leonie
```

(c) Conceptual implementation of a heavy computation step built by Leonie.
File: Compute.cpp

Figure 4.19: Conceptual example of a software project with multiple authors and files.

showed that, by lifting SEAL's commit interaction graph to author information, we can build an author-interaction graph that establishes interactions based on data-flow relationships between their code. Author interactions hint at coordination requirements between authors [30, 31, 133] in that when one author's code consumes data produced by another author, the two should coordinate. Compared to existing approaches to compute coordination requirements, such as file-based [105] or call-graph-based approaches [151], considering data flow has two clear benefits: First, using data-flow information spurious connections can be excluded (if there is no data flow between two code parts, they do not influence each other). Second, data flows, especially inter-procedural data flows, reveal dependencies between code that current approaches cannot find.

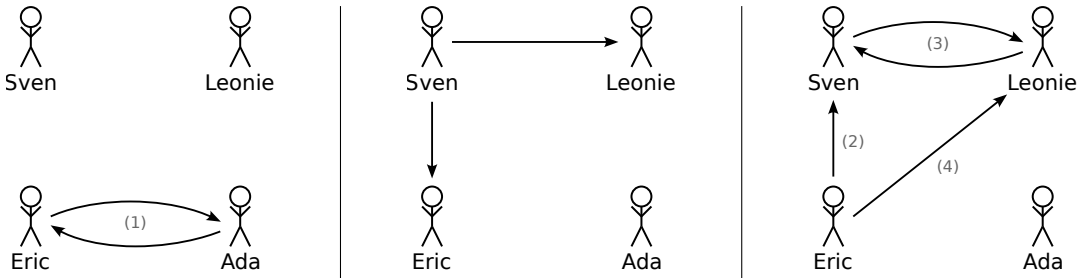
For illustration, let us highlight conceptual differences in the data produced by existing approaches and data-flow-based commit interactions on a exemplary study in which we want to determine coordination requirements between developers based on code dependencies. Consider the example code in Figure 4.19: A data processing service consisting of the service itself (Figure 4.19a), a user-data access layer (Figure 4.19b), and a computation worker (Figure 4.19c). Like in the real world, the different parts of the code base are implemented by different developers (names shown on the right). To demonstrate key differences in the data produced, we compare the interactions computed by SEAL against the two commonly used approaches for computing artifact coupling, file-based and call-graph-based, which we did also use in our evaluation (Section 4.5.1). The coordination graphs computed

by the different approaches are depicted in Figure 4.20. When comparing the file-based graph (Figure 4.20a) to the other two, we notice that link 1 between Eric and Ada is included only in the file-based graph. The point is that link 1 is actually spurious as, at the code level, there is no coordination requirement between the two functions in the file. The reason is that the file-based approach over-approximates by considering everything in a file as related without taking program semantics into account [85]. A programmer can easily see that these two functions are alternatives and, therefore, Eric and Ada can work independently. Furthermore, the file-based graph, compared to the other two graphs, does not contain the links, such as the one between Sven and Leonie and between Sven and Eric. This is due to the limitation of not considering dependencies across files.

When we compare the call-graph-based graph with SEAL, we notice that SEAL inverts the direction of link 2 and finds two additional links (3, 4). SEAL inverts the direction of link 2 because, from a data-flow perspective. Sven's code consumes data produced only by Eric but does not supply input to Eric's code[7]. When we consider link 3, we see a bidirectional connection between Sven and Leonie, capturing the fact that data from Sven are used as an input to the code of Leonie and vice versa. SEAL differentiates between a function call with and without data. In contrast, when using a call-graph approach, we find only that there is a connection between Sven and Leonie and between Sven and Eric. With SEAL however, we now know that Eric's code does not depend on Sven's code, since no input is passed to Eric's implementation, but Leonie's code does depend on Sven's code as her code works on input provided by Sven.

Many studies treat collaboration links between developers as undirected [151], or cannot infer a direction in links between artifacts, such as in co-changes retrieved from files that are commonly committed within one commit [120]. With SEAL, we obtain information about the direction that is based on the underlying data flows. This code-based directionality adds additional value, compared to temporal directionality [107], as it encodes who is using whose code. Parts of this information is also present in call graphs, we can infer only who is calling whose code but not, for instance, which data are passed to the function and how the returned data are used. Ignore this information, we would not be able to notice indirect data dependencies, such as the one between Eric and Leonie that produces link 4. This link arises from the fact that Sven forwards data computed by Eric's loading function to Leonie's compute implementation. This information hints at an important coordination requirement that should not be missed as when Eric adapts what user data are loaded, Leonie's implementation needs to handle it. Hence, coordination between Eric and Leonie is important before Eric makes a change. This kind of coordination requirement is found only with SEAL and can be found only by utilizing a whole-program inter-procedural data-flow analysis.

7 This is in itself is neither a drawback nor an advantage of SEAL compared to the call-graph-based approaches as the meaning of directionality depends on the use case and follows from the actual research question (e.g., where determining a link between to artifacts does not require directionality, a link between developers does if we want to determine who is using whose code).



(a) File-based (b) Call-graph-based (c) SEAL based

Figure 4.20: Coordination requirements between authors as determined by different artifact coupling approaches.

Integration of SEAL into other research. A good example for illustrating SEAL's merits arises from the work from Mauerer et al. [151]: In a large-scale empirical study, they explore the relationship between software quality metrics and socio-technical congruence by analyzing the alignment of social communication structures with technical dependencies. Important for our discussion is that they determine these dependencies by extracting them from the version control system: two artifacts are related when they appear in the same commit, and two artifacts are dependent when they have static language-level dependencies, such as call-graph connections and type references. As explained in our example above, data flows carry additional information compared to file or call-graph-based approaches and thus enrich the artifact dependency networks of Mauerer et al. [151].

SEAL can not only help to determine artifact relationships, but with our author-interaction graph, approaches that analyze developer communities [107] or organizational structures [106] can profit from detailed data-flow based interaction data, such as, the work from Joblin et al. [107], who mine developer communities based on commit information and source-code structure. Specifically, they use a function-based and a committer-author-based approach to determine related authors. This way, they potentially miss important non-local links between developers, as conceptually shown in our example. With SEAL, they are able to extend their information on developers with data-flow-based author interactions and expand their author network as well as remove potential spurious links.

Another example in which SEAL's commit interactions can be beneficial is in coordinating bug fixes or semantic code changes in large code bases or ecosystems. It has been reported that, for large industry code bases, even small code changes can cause severe bugs in other parts of the code [249] (e.g., a pointer can now be null, an integer variable can now have values larger than `x`, or a list that was previously sorted is now unsorted). As preventing changes is not an option (applying a bug fix that patches a vulnerability can not be delayed indefinitely), developers need to find ways to coordinate and let others know about the change. With with commit-author interaction data produced by SEAL, we can easily figure out who's code could be affected by a change. Think about link 4 from Eric to Leonie that SEAL found because data produced by Eric is passed via Sven to Leonie's code. With this

information, Eric can proactively contact Leonie to look at his proposed change to determine that her code works well with it.

4.7.2 Socio-Technical Data-Flow Analysis

Example. Interpreting or acting on program analysis findings is a difficult task, especially in larger software projects [20]. One reason is that common program analysis tools focus on technical aspects, showing what is wrong in the code or highlighting conceptual problems [165, 166]. Typically, program analysis tools do not put their findings into a socio-technical context, ignoring the social structure around the code. Manually fitting this information post-hoc onto the analysis results is cumbersome and difficult, once because developers do not have this information in their heads but also because tools such as git blame provide only raw information that is not contextualized with regard to the analysis semantics. With SEAL, we build a bridge and automatically attach socio-technical information onto low-level analysis findings.

Consider the example in Figure 4.21 (see Section 4.4.2), where a program analysis tool, such as PhASAR, identified an SQL injection. After the tool analyzed the code,

```
1  string sanitize(const string &s) {                                  ▷ de8781b   Eric
2    if (in_test_mode) { return s; }                                    ▷ ea8426c   Eric
3    return sanitizeSQLString(s);                                       ▷ de8781b   Eric
4  }                                                                    ▷ de8781b   Eric
5  int main(int argc, char **argv) {                                    ▷ c4d9b1a   Sven
6    auto *con = driver->connect(/* credentials */);                    ▷ 3e8882e   Sven
7    auto *smt = con->createStatement();                                ▷ 3e8882e   Sven
8    string q = "SELECT name FROM students where id=";                  ▷ 3e8882e   Sven
9    string input = argv[1];                                            ▷ 0872f49   Ada
10   string sani = sanitize(input);                                     ▷ 5341f7b   Leonie
11   auto *res = stmt->executeQuery(q + sani);                          ▷ 5341f7b   Leonie
12   if (!res->rowsCount()) cout << "no records\n";                     ▷ 0872f49   Ada
13   while (res->next()) cout << res->getString("name");                ▷ 0872f49   Ada
14   delete stmt; delete res; delete con;                               ▷ 3e8882e   Sven
15   return 0;                                                          ▷ c4d9b1a   Sven
16 }                                                                    ▷ c4d9b1a   Sven
```

Figure 4.21: A program that is vulnerable to SQL injections. Each ▷ indicates the commit that last modified each line and right next to it, is the name of the commits author.

it reports that there is a possible SQL injection vulnerability at Line 11, arising from a data flow from variable `sani` (Line 10), which contains unsanitized user input. Following this report, an engineer from the company's security team can investigate the problem and finds that the recent change `5341f7b`, depicted in Figure 4.22, introduced the offending variable in Line 10. Based on this information, Leonie can be contacted by the security engineer and asked to fix the SQL injection. However, this initial conclusion is wrong! Leonie's code did not introduce the SQL injection, since the actual problem is in the implementation of `sanitize`, where unsanitized

```
10  +   string sani = sanitize(input);
11  +   auto *res = stmt->executeQuery(q + sani);
12  -   auto *res = stmt->executeQuery(q + input);
```

Figure 4.22: Commit `5341f7b` in which Leonie wants to prevent SQL injections by adding a call to `sanitize`.

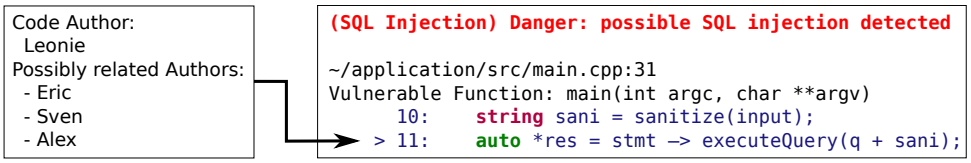


Figure 4.23: A socio-technical bug report, linking the author of the vulnerable code and possibly related authors that interact through data flow with the vulnerable code location to a found SQL injection.

data is leaked when running in test mode. This is likely found out only later when she starts digging into her code and the implementation of `sanitze` and refers the problem back to the security engineer.

Missattributions like this arise due to missing information and cost valuable developer time. From the point of the security engineer, the tool report was plausible, and Leonie's change seemed related, so the ticket was forwarded to her. With SEAL, this scenario could have worked out differently, if the tool's initial report would be combined with socio-technical information, obtained by SEAL's data-flow analysis. By attaching the commit-interaction path of the offending call instruction for `executeQuery`, which contains all commits that had influence on the data flowing into the call, we can determine all authors that are involved. Figure 4.23 depicts an exemplary error message, where the SQL injection tool's finding is contextualized with socio-technical information. So, in our example, the security engineer would see the report about the SQL injection, together with the information that the two developers Leonie and Eric are involved. The security engineer would then assign both the ticket and involve Eric from the beginning.

Integration of SEAL to other research. Inspired by our example, a number of existing program analysis tools qualify to be extended with additional socio-technical information. For example, Bessey et al. present an extensive experience report [20] that details on how industrial-grade static analyzers are used to find bugs in the real world and how these tools are perceived by companies and software developers: When confronted with analysis findings, developers tend to get emotional; At the end, it is their code that is supposedly flawed. Experience shows that the more peers are involved in discussing and dealing with analysis findings, the more likely it is that someone can diagnose an error (or identify a reported error as a false positive), report on experiences of similar errors, and eventually fix it [20]. SEAL provides

7 https://www.praetorian.com/blog/introducing-gokart/ (Last accessed: July 1, 2023)

this additional socio-technical information. For instance, it reports the set of authors involved in a given analysis finding, which makes it easier to assign errors to the relevant developers.

Furthermore, the information produced by program analysis tools can be filtered according to the attached socio-technical information (e.g., an analysis tool used within an integrated development environment could show only the findings that actually have a socio-technical connection to the developer using it). In this vein, Harman and O'Hearn report on the difficulty of making bug reports more actionable [86]. In particular, they debunk the implicit assumption that analysis and testing technology that follow the "ROFL" (Report Only Failure List) strategy is enough to get engineers to fix reported bugs. Software developers, however, are never short on lists of bug, fail, and failure reports that need to be taken care of. Any additional piece of information that helps developers prioritize their work are helpful. The report of Harman and O'Hearn underlines the importance of the following information to find and prioritize bugs: (i) *relevance*, the developer to whom the bug report is sent is one of the set of suitable people to fix the bug; (ii) *context*, the bug can be understood effectively; (iii) *timeliness*, the information arrives in time to allow, an effective bug fix; and (iv) *debug payload*, the information provided by the tool makes the fix process efficient (reproduced from [86]). SEAL helps to deliver such information.

The socio-technical interaction provided by SEAL may benefit scheduling code reviews (e.g., developers with a lot of data dependencies to the changed code can be suggested as reviewers) as the changes could potentially interact with their code and introduce bugs. In 2008, a Debian developer accidentally broke a random number generator in a particular version of OpenSSL with what was thought to be a fix[8]. This shows that—in practice—it can be quite difficult to assign suitable reviewers for a given pull-request. With help of information as computed by SEAL, this could possibly have been avoided by allowing peers that are familiar with this complex part of OpenSSL's code base to intervene.

4.7.3 Limitations

A bottleneck of using SEAL is clearly the computational cost of the underlying data-flow analysis. Table 4.2 depicts the overhead generated by SEAL for the projects of our study (see Section 4.5.1). We see that, for the average project, computing the blame data adds roughly two minutes. Interesting to note, computing blame information correlates with the history length of a project ($\rho_{\text{pearson}} = 0.75$ and $\rho_{\text{spearman}} = 0.90$, depicted in Figure 4.24), so project with longer histories should

8 https://en.wikinews.org/wiki/Predictable_random_number_generator_discovered_in_the_Debian_version_of_OpenSSL (Last accessed: Sept 28, 2023)

Table 4.2: Blame-annotation overhead (t_{Blame}) and analysis time (t_{Analysis}) measurements in seconds. These measurements are further contextualized with history size (t_{Blame}/Commit) and project size (t_{Analysis}/LOC).

	LOC	Commits	t_{Blame}	$\frac{t_{\text{Blame}}}{\text{Commit}}$	$\frac{t_{\text{Blame}}}{\text{LOC}}$	t_{Analysis}	$\frac{t_{\text{Analysis}}}{\text{LOC}}$
BISON	591 687	26 281	293.82	1.34	0.06	14 849.20	0.03
BROTLI	34 833	1 030	17.68	0.62	0.02	614.95	0.02
CURL	195 685	26 949	939.42	6.83	0.94	47 639.50	0.24
GREP	619 598	23 794	233.14	1.09	0.04	669.76	0.00
GZIP	622 480	22 193	139.03	0.43	0.02	70.75	0.00
HTOP	25 775	2 243	36.78	1.15	0.10	313.90	0.01
LIBPNG	74 571	4 098	43.93	0.31	0.02	559.62	0.01
LIBSSH	95 235	5 126	38.74	0.65	0.03	13 469.70	0.14
LIBTIFF	88 561	3 470	15.48	0.32	0.01	6 005.81	0.07
LRZIP	19 215	935	1.27	0.03	0.00	593.82	0.03
LZ4	18 813	2 541	4.08	0.01	0.00	1 404.06	0.07
OPUS	70 267	4 077	84.02	3.48	0.20	10 379.50	0.15
XZ	38 441	1 298	20.73	1.44	0.05	9.18	0.00

expect a bit more overhead[9]. Overall, we can see that for nearly all projects the analysis time dominates the overhead. As described in Section 4.2.3, we tuned SEAL's analysis to be as precise as practically feasible. Specifically, we made our analysis context-sensitive, alias-aware, and inter-procedural. From our point of view, this is not too problematic since SEAL is designed to run once to capture a full and precise picture of a given software project. Nevertheless, the underlying data-flow analysis is highly configurable in the sense that PHASAR allows one to select different helper analyses and to change each analysis' parameters to trade off precision and performance. For instance, PHASAR allows its users to choose a less precise but faster points-to analysis. Similarly, one can choose a call-graph algorithm that underapproximates information and does not resolve indirect function calls instead of the one we chose. SEAL's call-graph algorithm more expensively tries to identify potential call targets at indirect function calls to improve analysis precision, using points-to information and type hierarchies. Especially C++ developers seem to minimize the amount of indirect jumps [207] such that underapproximating call-graph algorithms may still provide enough precision for a users needs. Through these tuning knobs, users can reduce the analysis time if (slightly) less precise results are acceptable for their setting.

9 For the history-length correlation, we observed only CURL as an outlier, which took particularly long to compute it's blame data. A manual inspection of the project revealed that many files in CURL contain very old code (committed pre 2007), which combined with the number of commits in the history, means that the blame computation often needs to traverse a very large part of the history.

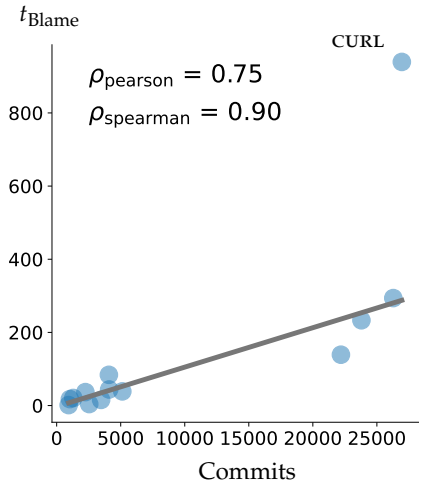


Figure 4.24: Visualization of the data presented in Table 4.2. For all projects except CURL, we see a linear trend between the number of commits a project has and the time it takes to compute the blame information.

Another limitation of SEAL is that the granularity of commit data are currently line-based, which is the common granularity used by git and other tools that work on git data. Previous work already demonstrated for different use cases [72, 140] that a syntax-based approach could, with a additional cost, improve the precision. With token-based blame information, as proposed by Germán et al. [72], SEAL could be even more precise.

Commit interactions based on data-flow dependencies provide a new way to infer dependencies between developers that is based on how their code interacts. However, there might be other kinds of dependencies that give rise to coordination requirements or are otherwise of interest. For example, external communication via file system or network, interactions through non-functional properties, side channels, and operating system level functionality could also require coordination between developers. Therefore, even if we can include data-flow dependencies with SEAL, we should still keep broadening the scope and exploring new kinds of information that is currently not available to determine developer interactions.

4.8 Related Work

SEAL can be applied to different areas since many techniques can profit from either incorporating repository information or detailed low-level program information. So, we discuss work that is directly related to our evaluation and we put our work in context to related areas that can benefit from our commit analysis.

Code complexity metrics. Over the last decades, various code complexity metrics have been proposed [65]. Tornhill [235] discusses multiple software metrics and analysis approaches that are used in companies on real world software projects to drive decision making and help with software maintenance. Software metrics help developers to focus on important code regions and maximize their improvement efforts. In their systematic mapping study, Varela et al. [239] categorized almost 300 different source code metrics from 226 studies into programming paradigms for

which they are used.They compare how and from which systems the metrics are extracted and rank them by the number of occurrences in studies.

A close look at Tornhill [235] and Varela et al. [239] reveals that nearly all common software metrics rely only on syntactical information. Very few software metrics go beyond syntax and, even if they do, they mostly consider information that is simple to obtain, like inheritance relationships. We see a potential for improvement here by incorporating more combined analysis approaches, such as commit interactions, especially when the analyses incorporate change information as well, which can help to refine results by putting them into a historical context.

Bug prediction. Bug prediction focuses on identifying and predicting potentially buggy code locations. D'Ambros et al. [47] compare a wide range of different bug prediction approaches. Many of the approaches focus on high-level information, such as change metrics (e.g., number of revisions), source code metrics (e.g., depth of inheritance tree), and code churn, and do not incorporate lower-level information, such as data flow, which can be used to approximate the program semantics. Khatri and Singh [119] performed a SWOT analysis on cross-project defect prediction (CPDP), analyzing a wide range of approaches. Interestingly, one key opportunity for improvement of CPDP that they highlight is the integration of more process metrics, such as number of developers working on a module. Our work can enable CPDP approaches to integrate process metrics that also incorporate low-level interactions between code changes or authors.

Program analysis. Program analysis techniques have been used successfully to prove specific properties about a program [61] and to detect bugs [172]. Program analysis tools, such as SpotBugs or clang-tidy, build on these techniques to catch bugs early in the development cycle. However, typical program analysis techniques, including srcML, analyze only one specific version of a program and do not incorporate version control information, which precludes detecting evolutionary problems, for example, detecting architectural decay by measuring the gradual deterioration of the coupling between classes. In addition, one could also extend already existing regression analyses by incorporating version control information directly into the analysis semantics. By utilizing both high-level repository information and data-flow information in a joined analysis, one could bridge this gap.

Socio-technical software analytics. In a special issue about software analytics, Menzies and Zimmermann highlight the importance of using analytical methods that incorporate data from real-world software projects to reason about software development processes [156]. They predict that the field will develop and benefit from more and different data sources. In the same vein, in a meta-analysis on socio-technical software-engineering research, Storey et al. [223] examine a wide range of publications to determine the current state of the art and areas for further improvement. They highlight that many research papers employ a data-driven approach. So far the used data sources (e.g., issue, bug, or commits) often do not

incorporate low-level dependency information as provided by data-flow analysis. For example, the state of the art in socio-technical analyses is to use function-level semantic coupling [105] or fine-grained co-edits [75] to represent interdependencies between software artifacts. Our work tries to address this shortcoming by providing a conceptual framework that combines the existing data sources with program semantics. This information could then be used to discover previously hidden socio-technical connections and coupling between developers that are invisible when only looking at syntactical information.

Software repository mining. Software repository mining focuses on gathering, modeling, and studying the data and software artifacts produced by developers during the software development process [48]. Kagdi et al. [110] state that source code changes are the fundamental unit of software evolution but also mention that current version control systems do not provide information about code semantics. This moved the initial focus of the research area to consider mostly change meta-data; a shortcoming which we address with SEAL.

Change impact analysis. The goal of change-impact analysis is to determine the consequences of a change using dependency analysis techniques, such as data-flow or control-flow analyses [26].

In a survey, Li et al. [141] analyze 23 different code-based change-impact analyses and built a framework to compare them. The authors state that many approaches use traditional program analysis techniques. They further highlight that useful information for improving change-impact analysis can be obtained with repository mining techniques. For example, Kagdi et al. [111] ran their conceptual coupling analysis on the current and previous version of source code to improve their analysis precision. Lehnert [137] lists only few approaches that combine repository mining with traditional program analysis. For example, Kagdi and Maletic [109] use high-level information and dependency information separately and look whether predictions based on either of these two agree. Conceptually SEAL is able to improve existing change-impact analysis approaches as we combine syntactic and semantic information into one joint analysis.

In a user study, Hanam et al. [85] have shown that semantic relations computed by static analysis helped users in completing code review tasks faster compared to using only syntactic relations. Their approach uses abstract interpretation in combination with an AST-based diff to extract semantic relations from JavaScript projects to reduce unwanted "noise" in purely syntactic relations. In contrast to their approach, SEAL has information about *all* commits available during the analysis, allowing for even more control in determining which interactions are noise and which are not. Still, the results of Hanam et al. support our claims that combining static analysis with change information can be beneficial for change impact analysis.

AST-based analysis. There are tools that combine repository information with syntax information. With such light-weight syntax-based repository mining tools,

most notably Boa [58], researchers can gather repository metrics and high-level code information about a wide range of different software projects. However, those do not model language semantics and do not allow us to attribute socio-technical information to the results of other more sophisticated program analyses.

Some research in this direction builds on srcML, "an infrastructure for the exploration, analysis, and manipulation of source code" [38]. srcML has been used, among other things, for type checking [170], program slicing [171], and pointer analysis [256]. However, srcML only provides an AST-based view on the code, and does not come with support for more sophisticated analyses, such as a data-flow analysis. The reason is that, srcML by itself does not model language semantics. For example, srcML does not run the preprocessor or model C/C++ language feature, such as, overload resolution or template instantiation, which are important to infer semantics. That's why we built SEAL on top of Clang and LLVM, a industry strength compiler framework, enabling us to combining program analysis with repository mining. In any case, syntax analysis is less precise than a data-flow analysis in determining dependencies between program parts.

Variability-aware analysis. *Variability-aware analysis* aims at efficiently analyzing variant-rich software systems [232]. The key is that, instead of analyzing all variants individually, a *variational* program representation (i.e., a program representation that retains all points of variability) is analyzed [246]. The goal is to save analysis effort by efficiently reusing analysis results across variants [194]. Variability-aware analysis is related to SEAL's approach in that it incorporates multiple variants of a software system (possibly generated by different variability implementation or configuration mechanisms) in a program analysis run. In contrast to SEAL, the goal is performance; incorporating historical and socio-technical repository information is not in scope.

4.9 Summary

In what follows, we contextualize the contributions of the work presented in this chapter to the field of repository mining and socio-technical program analysis, and with regard to general contribution that it makes to our overall thesis.

State-of-the-art software repository analyses often do not have precise information about a program's operational semantics at their disposal, if at all, or they try to include selected information in an ad-hoc manner. On the flip side, program analyses such as data-flow analysis do not have access to repository information, which restricts the interpretability of their results by excluding the socio-technical context of the software project.

SEAL bridges this gap by conceptually mapping repository specific information into the compiler's internal representation. The mapped information can be used by specialized data-flow analyses to infer relationships between commits (i.e., determine

commit interactions) or by existing data-flow analyses to augment their results with repository information.

In an evaluation on 13 open-source projects, we have demonstrated that SEAL and the generated data-flow-aware repository information can be utilized to answer relevant questions in research and practice.

Contribution to the field of repository analysis. The first part of our evaluation shows that, with SEAL, we can obtain new insights that could not have be found with existing methods that do not integrate both repository information and data-flow information. Our qualitative analysis has uncovered interesting cases, for example, where textually small changes have a far-reaching impact, which could only be pinned down by considering data flows.

SEAL introduces a new method to the field of repository analysis for extracting repository information that is connected to a program's operational semantics, by incorporating data-flow information. This way, we get through commit interactions a combined form of information that simultaneously represents program semantics and the development history of a software project.

Contribution to the field of socio-technical analysis. The second part of the evaluation, demonstrates how repository information computed by SEAL can be utilized to augment existing program analyses. This allows us to put the analysis results into a socio-technical context for further interpretation, for example, by relating SQL injection vulnerability directly to the involved developers. We expect that the newly gained data-flow-based repository information will present useful to a multiple of socio-technical analyses that want to incorporate a program's operational semantics.

Our evaluation demonstrates SEAL can be used to track down previously hidden interactions and to augment existing analyses with socio-technical information, which enables us to gain insights that could previously not be tracked down.

General contribution to this thesis. The work presented in this chapter employs code regions to find data-flow-based commits interactions. The application shows that evolutionary variability (*time*), encoded through commits, can be modeled through code regions (see Section 4.2.2). Furthermore, in Section 4.2.3 we demonstrated how an inter-procedural alias-aware context and flow-sensitive data-flow analysis can be integrated into uniform code-region abstraction, laying the analytical foundation for determining commit interactions. Taken together, code regions worked as a bridge between the variability information and program analysis, which enabled us to gain new insights into the field of repository analysis.

5

White-Box Performance Analysis of Configurable Software Systems

Detecting and reasoning about performance bugs is difficult, especially, in configurable software systems [34, 83, 252], as analyzing these systems is impeded by the combinatorial explosion and potential interactions between configuration options (see Section 2.1.2). The large number of configurations makes it difficult to measure non-functional properties, such as performance, and also hard to attribute the measured properties to configuration choices. As discussed in Section 2.3.7.3, existing approaches tackle this problem either through sampling in combination with black-box performance analysis, or white-box analyzes that localize configuration-dependent code. Black-box approaches correlate the measured performance to configuration options, with the drawback of not being able to attribute the measurements to source code. In comparison, configuration-focused white-box performance analyses locate first configuration-dependent code and then measure it. However, current configuration-focused white-box analyses produce too much overhead [240] and are not flexible with regard to *how* they measure performance and *what* metrics they can collect. Industry-strength state-of-practice performance profilers, on the other hand provide a wide range of performance measurement techniques, some that only produce very low overhead [78]. However, these profilers do not incorporate configuration knowledge and, by that, are difficult to use on configurable software systems.

The underlying problem is that state-of-practice performance profilers cannot make use of the configuration knowledge that is available to configuration-focused white-box analyses. We solve this problem by integrating configuration knowledge from localization approaches into state-of-practice performance profilers, making profilers configuration aware. This means, we enable state-of-practice performance profilers to attribute their measurements to configuration options by weaving in measurement code at configuration-dependent code locations.

The approach presented in this chapter, demonstrates how we can integrate configuration knowledge conceptually with state-of-practice profilers using the code-region abstraction introduced in Chapter 3. Our evaluation of 108 performance regressions on 16 synthetic subject systems and two real-world subject systems used in High-Performance Computing (HPC), shows that configuration-aware performance profilers are able to identify configuration-specific performance regressions.

In addition, configuration-aware performance profilers do not produce significantly more overhead than existing approaches, which shows that configuration-sensitive profiling is feasible. Complementary to this chapter's main contribution of making state-of-practice profilers configuration aware, this chapter also demonstrates the applicability of code regions to configuration variability (*space*) as well as a dynamic analysis. Furthermore, this chapter also demonstrates how code regions make program analyses reusable, by using the taint analysis introduced in Chapter 4.

In what follows, we highlight the gap between current localization approaches and performance profiling in more detail. Afterwards, we demonstrate how state-of-practice performance profilers can be made configuration aware by integrating configuration knowledge from localization approaches, creating configuration-sensitive state-of-practice profilers—the best of both worlds.

5.1 Introduction

Performance bugs can cause significant performance degradations [14] that not only impact user experience [82, 102] but are also hard to detect and fix [14, 52]. Research has shown that a significant number of performance bugs are caused by configuration errors [83]. Configurability adds another layer of complexity to software systems, increasing the complex process of debugging and understanding performance issues even further [252]. That is, performance regression may manifest only for some configurations since buggy or inefficient code may only be executed, if the corresponding configuration option guarding that code region is activated [49] (e.g., see Section 2.1.4). Due to the resulting complex control-flow and data-flow dependencies among the code involved, developers often fail to understand the root cause of the performance problems they observe [242]. Hence, both practitioners and researchers would benefit from means to ease performance debugging of configurable software systems.

In practice, developers often address performance issues by employing state-of-practice performance profiling tools, which enable them to precisely drill down into performance issues. Profiling tools such as XRay and *e*BPF are highly optimized and tuned to produce precise measurements without causing too much overhead or distortion in the system under measurement. A key problem is that state-of-practice tools are oblivious to the fact that almost all practical software systems are highly configurable [114]. As a workaround, developers concentrate only on a small set of default or notorious configurations, likely missing configuration-specific regressions [114]. Alternatively, developers can retrofit configurability by applying profiling tools to as many as possible different configurations of the software in question, which entails high profiling costs. Fundamentally, profiling tools lack, a deep understanding of variability in a system's source code, such that they often break down due to the combinatorial explosion of the configuration space or miss important interactions among configuration options [125]. For example, by only sampling a set of variants to profile or by choosing potential variants through

developer knowledge. This makes the performance debugging process ultimately cumbersome, time-consuming, and likely leads to undiagnosed or misdiagnosed performance problems.

In academia, a multitude of approaches have been proposed to locate configuration-dependent code [147] and to incorporate configurability into the performance measurement process [240]. For example, ConfigCrusher [240] measures configuration-specific performance, using Lotrack [147] to locate configuration-dependent code. While demonstrating the principle ability to analyze performance in the presence of configurability, these approaches have limits when being used on real-world code bases, either because their profiling infrastructure or measurement code produces too much overhead or they do not scale down to the fine granularity of configuration-specific code [240, 247]. Some approaches try to tackle these problems by expanding the measured scope, meaning, they do fewer coarse-grained measurements, thereby losing accuracy, though [240]. Other approaches employ additional dynamic analyses, increasing setup complexity and the overall measurement overhead [241], as compared to standard profilers. In summary, state-of-the-art analysis approaches are not as mature as industrial-strength performance profilers, lack flexibility, produce too much profiling overhead, and are difficult to use in production environments.

Given the availability of industrial-strength profilers and the progress that has been made in research, there is a promising middle ground that has not yet been explored so far: integrating configuration-aware analysis with state-of-practice profilers. This way, the analysis can utilize the optimized low-overhead measurements of profilers together with the profilers' ecosystem. Ideally, such an integration should neither focus on a specific analysis approach nor work only with one kind of profiler. Instead, an integration should provide a general and easy-to-use interface for making a given performance profiler configuration-aware. To achieve this goal, we propose Walrus, a configuration-aware analysis framework that combines state-of-the-art localization approaches to detect configuration-dependent code with state-of-practice profilers. Walrus builds on top of LLVM, introducing a configuration-specific abstraction into the LLVM compiler infrastructure that associates configuration options to corresponding code regions. That is, Walrus automatically determines which code regions depend on which configuration options (i.e., which regions are selectively executed depending on the values chosen for the corresponding configuration options). This abstraction enables for the first time analysis or optimization passes to access configuration-related information during compile time. To demonstrate its practicability, we built Wlang, an extension of the Clang compiler that can automatically insert configuration-aware measurement code on top of Walrus. Walrus offers a generic instrumentation interface to automatically add measurement code to configuration-specific sections of a program or to encode configuration-related information into the binary for later use. Making use of this interface, Walrus already provides a range of low-level instrumentations that make different state-of-practice performance profilers configuration aware including XRay and *e*BPF. Combining existing (academic) approaches for locating configuration-specific code

with state-of-practice performance profilers has not been possible up to this point. Walrus is the first approach to make load-time and compile-time configuration options and their affected code regions traceable within the compiler to inject tailored measurement code at a fine grain.

In a series of experiments involving compile-time and load-time configurability, we demonstrate that a state-of-the-art configuration-aware program analysis built on top of Walrus can identify real-world performance regressions in configurable software systems, without making compromises on measurement granularity by over approximating configuration dependent code regions. We show how the integration of state-of-practice profiling tools plugged into Walrus only produces a reasonable measurement overhead and, by that, enables configuration-aware performance measurements.

The primary contributions of this chapter are:

- Walrus, an extension of LLVM that makes the compiler infrastructure configuration-aware, enabling it to detect and model configuration-specific code regions;
- Wlang, an extension of Clang built with Walrus that can automatically insert custom measurement code for performance profiling configuration-dependent code regions;
- A plug-in mechanism for state-of-practice performance measurement tools that enable them to analyze configuration-dependent code regions.
- An evaluation that demonstrates the feasibility and applicability of configuration-aware performance measurement with Walrus.

5.2 Walrus at a Glance

The main reason why performance debugging of configurable software systems is so difficult is that configuration-dependent code (i.e., code that gets only included or executed in the binary depending on the used configuration) shows up in performance profiles only under certain configurations. So, finding performance bugs requires knowledge under which configurations these bugs manifest. Worse, even when executing a configuration that enables the configuration-specific code, it is not clear which part of the performance profile can be attributed to the chosen configuration option and which is not. That is, we now observe the execution time of the code but we cannot directly trace this information to the corresponding configuration option.

This *attribution* problem arises from a loss of tracing information: At the source-code level, we know which part of the code is controlled by a configuration option, however, after compilation, this information is no longer directly present in the binary and, therefore, cannot be utilized by performance profilers to *attribute* the measured performance post hoc to the corresponding configuration option.

```
1  struct Config {
2    bool UseEncryption;
3    bool UseCompression;
4  } currentConfig;
5
6  void loadConfigFromFile() {
7    currentConfig.UseCompression = false;
8    currentConfig.UseEncryption = true;
9  }
10
11 void sendPackage(PackageData Data) {
12   if (currentConfig.UseCompression) {
13     Data = compress(Data);
14   }
15   if (currentConfig.UseEncryption) {
16     if (not currentConfig.UseCompression) {
17       // Only add padding for uncompressed data
18       Data = addPadding(Data)
19     }
20     Data = encrypt(Data);
21   }
22   send(Data);
23 }
```

Figure 5.1: Running example of a load-time configurable C++ program. The example has a global configuration variable (1) that stores the configruation value of two options *encryption* (🔒) and *compression* (📦). Both options get initialized during program start in function `loadConfigFromFile` (2). Function `sendPackage` is composed of three configurable code sections, each controlled by a different combination of configuration variables. For example, Line 18 is executed only if `UseEncryption` is enabled and `UseCompression` is disabled.

Take, for example, the function `compress` (3) in Figure 5.1, which is executed only if the configuration option `useCompression` is enabled. As a consequence, function `compress` shows up in performance profiles only under certain conditions and, even if it appears, developers cannot identify which part of the profile is configuration specific and how much time is spent in the configuration-specific parts.

The goal of Walrus is to enable the performance analysis of configurable software systems by solving the attribution problem. Walrus detects configuration-specific code and preserves this information during compilation to later empower state-of-practice performance profilers in analyzing configurable software systems in a configuration-aware manner, that is, making time spent in configuration-specific parts of the code distinguishable in the performance profile. Walrus achieves that by (1) extending the compilation pipeline with different analyses to detect configuration-specific code regions and (2) with custom instrumentation passes that model and map this information into the binary.

An overview of the compilation pipeline with Walrus is depicted in Figure 5.2. In short, Walrus takes in configurability information from the developer, utilizes different localization approaches to detect configuration-dependent code, and then weaves in profiler-specific instrumentations around the detected code.

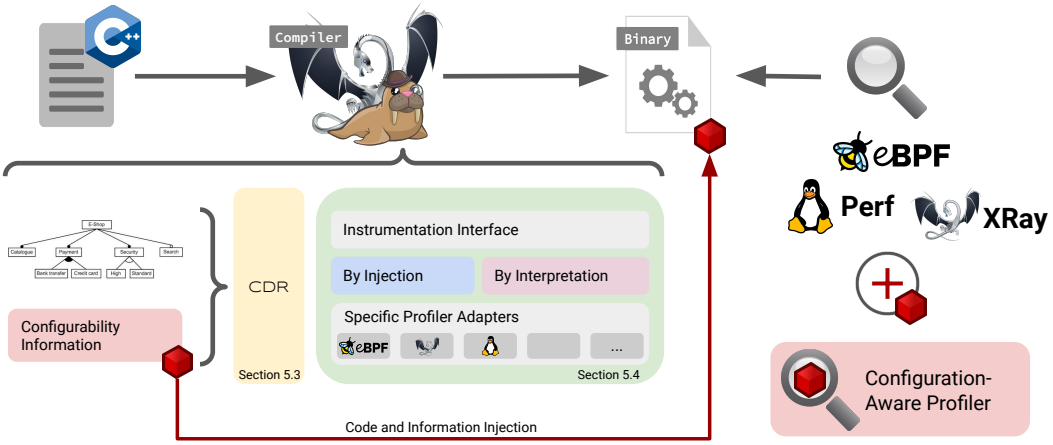


Figure 5.2: Overview of a configuration-aware profiling setup with Walrus. As part of the normal compilation process, Walrus incorporates configuration-specific information into the compiler intermediate representation. Based on that information, Walrus adds profiler-specific instrumentation code into the produced binary, enabling existing state-of-practice profiling tools to make their profiles configuration aware.

5.2.1 Concept of Configuration-Dependent Regions

There is a multitude of localization approaches available that are able to determine different kinds of configuration-dependent code, all of which could be used to contextualize the performance of a configurable system. So, to support the performance measurements of different kinds of configurability, we designed Walrus to be independent of the actual localization strategy used. To this end, we define configuration-dependent regions, based on our code-region abstraction introduced in Chapter 3, that represent configuration-dependent code blocks.

Each program p (see Section 3.2.1) has a set of configuration options $\mathcal{C}$ (see Definition 1). For example, in the case of static variability with `#ifdefs`, these are macros, in the case of load-time variability program variables encode the state of the configuration option. Which variables model configuration options are initially specified by the developer and are given as input to Walrus. Walrus requires as input a function that relates instructions to the specific set of configuration options that influence it—the function tags that we introduced in Definition 11. For Walrus the set of domain-specific tags $\mathcal{T}$ is the set of configuration options $\mathcal{C}$. It is important to note that the issues of determining how an instruction i is influenced ($\mapsto$) by a configuration option is defined by the localization approach and is not central to Walrus. On the practical side, this information is added through a localization approach, such as, the ones we described earlier in Section 2.1.4. We define confOptions as the Walrus specific tags function.

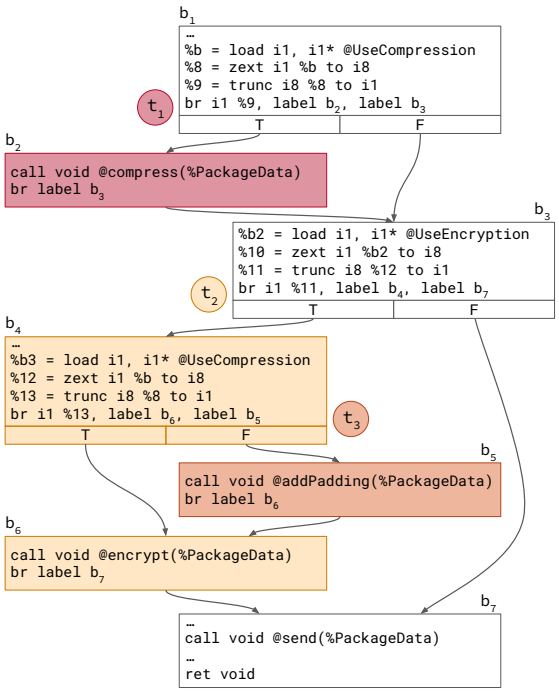


Figure 5.3: Reduced control-flow graph for function `sendPackage` from Figure 5.1. The code inside the graph nodes are exemplary code snippets from the compilers internal representation that are related to configuration variables. The three configuration dependent terminators are labeled as t_1 (red), t_2 (yellow), t_3 (orange), and the resulting configuration-dependent regions are colored respectively.

Definition 34. Let $\mathsf{confOptions}(i)$ be a function that maps an instruction i to the set of configuration options that influence ($\mapsto$) the execution of i.

$$\mathsf{confOptions}(i) = \{\, c \mid c \in \mathcal{C} \ \wedge \ c \mapsto i \,\}$$

Based on these foundations, we now define configuration-dependent region (abbreviated in the following with CDR), as Walrus's specific type of code region that focuses on configuration options. We define a configuration-dependent region as a code region where $\mathcal{T} = \mathcal{C}$. Hence, we can compute all configuration-dependent regions for a function f through $\mathsf{computeCR}_{\mathsf{Tag}}$ (see Definition 14).

CDR is based on our code-region abstraction but focuses on configuration-dependent code. Hence, CDR is an overarching abstraction for configuration variability (*space* dimension) that enables us to model different kinds configuration-dependent code uniformly on a compiler-internal intermediate representation (e.g., LLVM-IR), so we can later instrument the detected code regions. In what follows, we explain in detail how CDR can be used to find load-time-based configuration-dependent regions, as well as, compile-time-based configuration-dependent regions.

5.2.2 Load-Time Configuration-Dependent Regions

Walrus models configurability in the compiler by implementing *configuration-dependent regions* as an abstraction over the compiler internal code representation, to make configuration-specific code measurable. In what follows, we demonstrate a concrete way to integrate a load-time configurability localization approach, similar to Lotrack [147], into Walrus, utilizing our previously introduced CDR abstraction.

To determine all load-time configuration-dependent regions, that is, all load-time configurable parts of a program, Walrus initially determines all control-flow decisions that depend on configuration options by running a specialized taint analysis—for this, we reuse the analysis from Chapter 4. Next, all instructions that are dominated by these control-flow decisions are marked as dependent on the configuration options. This way, we introduce function confOptions and provide a mapping from instructions to configuration options.

Initially, the user specifies all relevant program variables v that represent a configuration option c, which implicitly defines the set $\mathcal{V}_\mathcal{C}$ of all variables that encode configuration options.

Definition 35. Let $\mathsf{option}(v)$ be a function that that maps a given program variable $v \in \mathcal{V}_\mathcal{C}$ to the corresponding configuration option $c \in \mathcal{C}$.

Next, to determine configuration-dependent code, we need to determine all terminator instructions `t`, whose control-flow decision depends on a configuration variable v. A *terminator* instruction terminates a basic block, which is a sequence of instructions, and either diverts control-flow to another point where the execution continues—the important case—or marks a block unreachable[1]. Hence, by determining all terminators that depend on a configuration variable, we find all control-flow decision points that can be influenced by a configuration option. That is, we find all terminators where a configuration option could influence a control-flow decision. For example, $\mathtt{t}_1$ in Figure 5.3 is tainted by the analysis with `UseCompression`, since its computation depends on the configuration variable `UseCompression`.

To find these terminators, we reuse the taint analysis that was already introduced in Section 4.2.3 and our code-region abstraction. First, we define two additional helper code regions: one for user-specified program variables $\mathcal{V}_\mathcal{C}$ and one for terminator instructions. For program variables, we define tags to map the user-provided mapping information onto instructions and $\mathcal{T} = \mathcal{V}_\mathcal{C}$. For terminator instructions, we define tags to determine whether an instruction is a terminator and extend $\mathcal{T}$ with τ, a element whose presence indicates a terminator. Second, we reuse our taint analysis from Section 4.2.3 through the data-flow interaction relation $\rightsquigarrow$ (see Definition 28). Next, we compute all interactions between the two helper regions through computeAllCRInteractions, extract the ones where baseRegion (see Definition 24) is a terminator region, and map the interacting code regions to their tags.

1 For more details, see LLVM's terminator description https://llvm.org/docs/LangRef.html#terminator-instructions. (Last accessed: October 13, 2023)

Definition 36. Let $\mathsf{influencingOptions}(\mathtt{t})$ be a function that maps, for a given program revision $\mathtt{rev} \in \mathcal{R}$, the given terminator instruction $\mathtt{t}$ to the set of configuration options that influence it.

$$\begin{aligned}\mathsf{influencingOptions}(\mathtt{t}) = \Big\{ \mathsf{option}(v_c) \mid \\ & cri \in \mathsf{computeAllCRInteractions}(\rightsquigarrow, \{\mathtt{rev}\}) \\ & \wedge \ \mathsf{contains}(\mathtt{t}, \mathsf{baseRegion}(cri)) \\ & \wedge \ \Gamma = \mathsf{interactingRegions}(cri) \\ & \wedge v_c \in \bigcup_{ir \in \Gamma} \mathsf{tags}_{\mathsf{CR}}(ir) \cap \mathcal{V}_{\mathcal{C}} \Big\}\end{aligned}$$

This way, we find for each terminator instructions the set of configuration options that influence it. We simplify our static analysis problem of finding terminator instructions that are influenced by configuration options by encoding the problem into our code-region abstraction and using an already established program analysis. Furthermore, based on Definition 36, we can now define the set of all load-time configuration-dependent terminators $\Gamma^{\mathcal{C}}$.

Definition 37. Let $\Gamma^{\mathcal{C}} \subseteq \Gamma$ be the set of all terminator instructions $t \in \Gamma$ whose control-flow decision depends on at least one configuration option $c \in \mathcal{C}$.

$$\Gamma^{\mathcal{C}} = \{ \mathtt{t} \mid |\mathsf{influencingOptions}(\mathtt{t})| > 0, \mathtt{t} \in \Gamma \}$$

After the analysis found all configuration-dependent control-flow decisions, we can determine all configuration-dependent instructions by finding all basic blocks that are conditionally executed by a terminator $\mathtt{t} \in \Gamma^{\mathcal{C}}$.

First, we determine the set of all basic blocks that are dominated[2] ($\gg$), but not post-dominated ($\ggg$), by $\mathtt{t}$'s basic block $b^{\mathtt{t}}$. Second, we mark every instruction inside the found basic blocks as depending on the configuration options $\mathtt{t}$ is influenced by ($\mathsf{influencingOptions}(\mathtt{t})$). Algorithm 2 depicts how the configuration map (i.e., a mapping from instructions to configuration options) is built in detail. This way, we establish a mapping from instruction to a set of configuration options that influence whether and how an instruction is executed and, by that, we define the tags function $\mathsf{confOptions}$ for load-time configuration-dependent code regions.

Example: In Figure 5.3, we show the simplified control-flow graph of `sendPackage` from Figure 5.1. In the graph, we see three marked control-flow decisions $\mathtt{t}_{1\text{-}3}$, which depend on different configuration variables. We detect these configuration-dependent control-flow decision by running a precise taint analysis that tracks the data flows starting at configuration variables to control-flow decisions $\mathtt{t} \in \Gamma$. After finding all configuration-dependent terminators $\Gamma^{\mathcal{C}}$, we determine for each terminator $\mathtt{t} \in \Gamma^{\mathcal{C}}$ all basic blocks, which can be reachable only through $\mathtt{t}$, and mark all their instructions as dependent on the corresponding configuration options.

2 A basic block b_a dominates ($\gg$) b_b if every path which passes through b_b must also pass through b_a [185].

Algorithm 2: Builds a configuration map by iterating over all control-flow decisions whose computation is dependent on a configuration option and marking instructions that get dependently executed with the corresponding configuration options. The resulting map then relates instructions to the set of configuration options that control its execution.

```
   Data: Γ^C
   Result: ConfMap
1  for t ∈ Γ^C do
2  |  b^t ← getBB(t);
3  |  BBQueue ← successors(b^t);
4  |  while b ∈ BBQueue do
   |  |  ▷ Check that b is dominated by b^t
   |  |  ▷ but not also post dominated
5  |  |  if b^t ≫ b ∧ ¬ b^t ⋙ b then
6  |  |  |  for i ∈ instructions(b) do
7  |  |  |  |  ConfMap[i] = influencingOptions(t) ∪ ConfMap[i]
8  |  |  |  end
9  |  |  |  BBQueue ← successors(b);
10 |  |  end
11 |  end
12 end
```

With the computed information, we have the data for confOptions. Combining this with our code-region abstraction, we can now determine all load-time configuration-dependent code regions. This demonstrates how we can integrate an existing localization approach, inspired by LOTRACK [147], to detect configuration-dependent regions and, by that, handle load-time variability with WALRUS.

5.2.3 Compile-Time Configuration-Dependent Regions

Similar to load-time configuration options, compile-time configuration options often have a high performance impact. So, to make their impact measurable, WALRUS needs to locate compile-time configuration code and instrument it. There are two commonly used implementation techniques to implement compile-time configurability: preprocessor annotations, in the form of `#ifdefs`, and template meta-programming, where especially the latter one is heavily employed in HPC frameworks, such as DUNE or HYTEG[3]. In what follows, we highlight how both these techniques can be integrated into WALRUS's CDR abstraction.

To conceptually handle preprocessor-based configurability, we need to create a specialized confOptions implementation that preserves preprocessor directives and maps them into the compiler IR. We create this specialized confOptions function

3 https://i10git.cs.fau.de/hyteg/hyteg (Last accessed: October 7, 2023)

by attaching preprocessor information, more specifically the configuration options used in the `#ifdefs` condition, and attach it as meta-data onto the compiler IR. Hence, when constructing the configuration-dependent regions, a function call to confOptions is nothing more than accessing the meta-data attached to an instruction. Note that this limits the localization approach to model code that can be simultaneously present in the program (i.e., it excludes alternative code sections for `#ifdefs`). As an alternative, we can map compile-time variability to load-time variability with the approach introduced by Rhein et al. [195], and afterwards applying our previously explained approach to handle load-time configuration options.

Template meta-programming uses templates (i.e., classes and functions that can be parameterized with types) to generate specialized classes and functions (see Section 2.1.4). Simply put, for a template class parameterized with a type T, during compilation, the compiler will generate a concrete version of the class and weaves in T's code. To handle such template based configurability, we implemented a special language annotations that enable developers to encode which parts of the code are special building blocks that are destined to be configured into template code. We then map the configurability information added by the annotations as meta-data data to the compiler IR, similar to information provided by `#ifdefs`. This way, we can use the same mapping function confOptions as with preprocessor based configurability to integrate this information into our CDR abstraction. Conceptually, this unifies both implementation techniques, as, in the end, it does not matter if the generated IR code was woven in with template meta-programming or included by a preprocessor macro.

By providing a confOptions implementation that embeds `#ifdefs` and template based configurability information into our CDR abstraction, we enable Walrus to locate compile-time configurable code. Up to now, we have shown how Walrus can locate and model compile-time and load-time configurable code. In the next step, we are going to use the detected configuration-dependent regions to weave in measurement code for measuring the time it takes to execute configuration-dependent code.

5.3 Making Profilers Configuration-Aware

Walrus enables developers to attribute performance measurements to configuration options. Using localization approaches discussed in Section 5.2, Walrus knows *where* configuration-dependent code is. In what follows, we describe how Walrus uses this information to weave in profiler-specific measurement code around configuration-dependent regions to enable configuration-specific performance measurements.

5.3.1 Profiler Interfaces

Tracking down configuration-related performance regressions is difficult and may require different kinds of profilers, depending on the specific problem at hand. Profilers can be classified according to *how* and *what* metrics they target (e.g., instrumentation-based profilers vs. sampling-based profilers or CPU profiling vs. memory profiling). So, to maintain this flexibility and still attribute a profilers measurements to configuration options, Walrus needs to support a wide range of different profilers. For this purpose, Walrus offers an instrumentation interface through which profiler-specific adapters can be implemented.

In general, Walrus offers two different mechanisms for plugging a profiler: *by injection* and *by interpretation*.

5.3.1.1 *By Injection*

Walrus can inject profiler-specific instrumentation code around configuration-dependent regions to be executed when the configuration-dependent region is entered or exited. For a profiling tool, a developer implements the instrumentation code in the form of light-weight adapter code that connects configurability-information with the profiler's measurement interface, through Walrus's instrumentation interface. Walrus then weaves the custom adapter instrumentations into the program during compilation, wrapping the configuration-dependent regions with measurement code. This way, the profiler has control over what information is collected but additionally has access to configuration information (e.g., what configuration options influence the execution of this code region), to attribute the collected measurements to configuration options.

Walrus's generic tracing interface offers instrumentation hooks that provide access to the aggregated configuration-specific information. Specifically, Walrus offers four instrumentation hooks: `initialize` and `finalize` can be used to set up background workers or for cleaning up at program start and end; `region_enter` and `region_exit` are hooks that are called every time a configuration-dependent region is entered or exited.[4]

```
1 /* Walrus's tracing interface */
2 void initialize();
3 void finalize();
4 void region_enter(CDRegion *Region);
5 void region_exit(CDRegion *Region);
```

To illustrate how the injected instrument hooks work, let us look at a simple example that logs the configuration options every time when configuration-related code is executed. The logger would implement `initialize` to setup the global logger and `finalize` to flush out log messages that have not yet been persisted.

4 The actual names of the functions injected into the binary are renamed and uniquified to prevent collisions with user code.

In addition, `region_enter`/`region_exit` handle the sending of log messages when configuration-specific code is executed, for example:

```
1 void region_enter(CDRegion *Region) {
2   logger.log("Executing {0} specific code.",
3              Region->configurationOptions());
4 }
```

Through the configuration information provided by Walrus, the logger can attribute the printed log message to configuration options.

In general, Walrus's injection feature is the preferred way of instrumenting code for event-based profilers such as XRay[5] and for measurement tools that require a more elaborated background setup (e.g., distributed tracing tool [179]).

5.3.1.2 *By Interpretation*

An alternative way to link measurements to configuration information is to make the profilers' existing measurements interpretable in a configuration-aware way: by guiding the profiler to measure only code that relates to a specific configuration option or by relating parts of the binary to a configuration option. Walrus supports such use cases by persisting configuration-specific information in the binary, which can be accessed later by the profiler to tune or relate its measurements to configuration options (e.g., a profiler could only consider measurements that were done while code from a specific configuration option was active).

Making profilers configuration aware by interpretation is especially useful for black-box profilers, which already can measure programs with low overhead but lack interpretability. By persisting configurability information in the binary, Walrus can turn black-box profilers into configuration-aware gray-box profilers. Take, for example, the *e*BPF-based profiler bpftrace. Walrus is able to map information about configuration options into the binary (e.g., the addresses in the binary where configuration-dependent code resides). bpftrace profiling scripts can use this information to execute measurement code only for regions that relate to a specific configuration option.

To insert configuration information, Walrus offers in its instrumentation interface generic interpretation hooks to persist arbitrary information in the static section of the binary, allowing the profiling tool to store configuration-related information in a profiler-specific way (see Figure 5.2) and access it afterwards during runtime.

As an example of how interpretation works, consider Walrus's support for statically-defined tracing (SDT) probes[6]. SDT probes, insert `nop` instructions as anchor points and then persist the addresses of these anchors together with additional information into the static section of the binary.

Figure 5.4 illustrates how the additional information is connected to the configuration-dependent region. In the lower part of Figure 5.4, `Location` stores the address

5 https://llvm.org/docs/XRay.html (Last accessed: July 27, 2023)

6 For convenience, Walrus already provides a number of common abstractions, such as SDT probes that persist configuration-related information into binary. SDT probes are placed into the binary and allow tracing tools, such as DTrace, to attach measurement code.

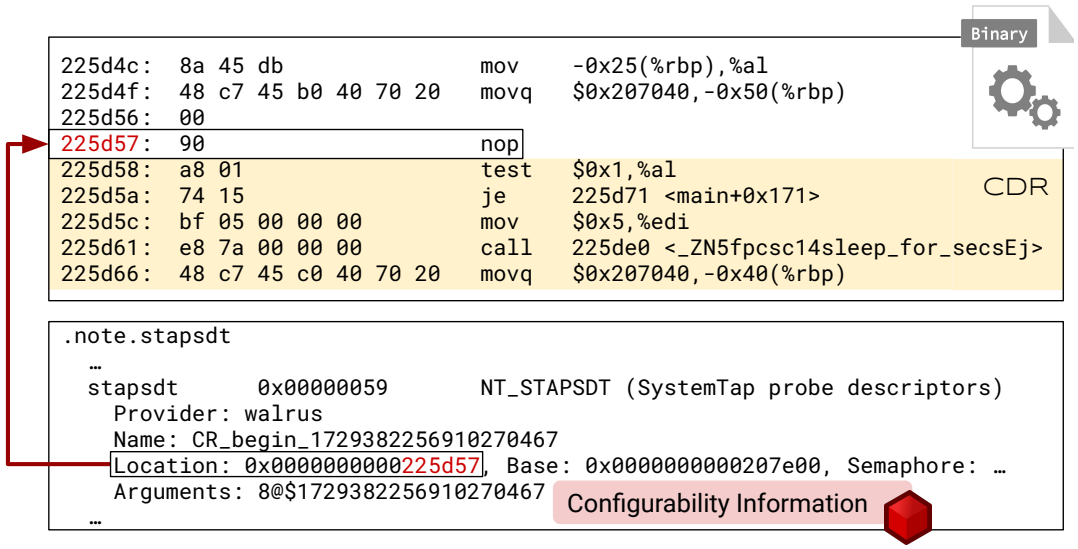


Figure 5.4: Software-defined tracing probe relating configuration-specific information to the code generated for a configuration-dependent region. The injected `nop` instruction serves as an anchor point for additional information that can later be inserted into the note section of the binary (e.g., which configuration options influence this configuration-dependent region).

of the `nop`, relating the entry to a location in the binary, and `Arguments` stores a pointer to the configuration-related information (e.g., containing a list of configuration options that influence the `Location`). This way, profilers such as BPFTRACE can add measurement code at runtime exclusively for anchor points that relate to a specific set of configuration option.

5.3.2 Application Scenarios

To illustrate the merits and flexibility of WALRUS, we showcase three profiling scenarios and how WALRUS enables configuration-aware profiling in each scenario. We vary how configuration-specific information is incorporated: (1) full profiler reuse, (2) configuration-specific information as an addon, (3) and custom profiler. We use several well-known profilers that employ different commonly used profiling approaches for each scenario, demonstrating how WALRUS enables these profilers to incorporate configuration-aware information. Specifically, we implemented configuration-aware versions of *e*BPF-based tools[7] for trace profiling [77, 78], and LLVM's XRAY profiler, as a commonly used instrumentation-based profiler. Furthermore, motivated by previous work [174, 212, 247], we demonstrate how WALRUS can generate performance-influence-model traces from a program execution, that is, a trace specific performance-influence model (see Section 2.3.7.3) that captures

7 https://ebpf.io/ (Last accessed: July 27, 2023)

the individual performance influences of configuration options and the interactions thereof for a single program execution.

Full profiler reuse. To illustrate how measurement code of *e*BPF-based profilers make use of WALRUS-inserted SDT probes, we list a one-line code snippet that attaches a measurement probe to every configuration-dependent region printing a message, every time a configuration-dependent region is entered.

```
bpftrace -e 'usdt:./binary:walrus:cdr_enter_* { printf("Entered CDRegion %s\n", str(arg1)); }'
```

As an alternative, BPFTRACE can also attach code only to regions of a single configuration option[8].

```
bpftrace -e 'usdt:./binary:walrus:cdr_enter_encryption_* { printf("Exec. Encryption Code\n"); }'
```

This way *any* *e*BPF-based performance tool can be applied in a configurations-specific way. That is, WALRUS enables through SDT probes a wide range of performance profilers that can target different types of performance metrics, such as CPU time, memory consumption, or network utilization [77], to attribute their measurements to configuration options.

Configuration-specific information as add-on. Compared to the previous profiling scenario, where we could directly use an existing profiler, here we use only core components of the profiler and collect additional performance measurements using LLVM's XRAY, which we subsequently merge into the profilers results. To this end, WALRUS instruments additional measurement code around configuration-dependent regions to measure their execution time, in addition to XRAY's function-level instrumentation. Subsequently, we merge the profiling data collected by XRAY with the one collected through the additional instrumentation. This produces a performance profile that contains both function-level and configuration-specific measurements data. Figure 5.5 visualizes a merged profile. The correct ordering and interleaving of measurements is guaranteed, as the instrumentation that WALRUS injects uses XRAY's internal timestamp clock. Through a combined profile, developers can reason about the performance influence of configuration options alongside standard function-level information, for instance, determining how much time is spent in configuration-specific section of a function. This profiling scenario shows that WALRUS can extend even more technically complicated cases, by reusing central components of a profiler to later incorporate configuration-specific information.

Custom profiler. To illustrate the third use case, we have developed a custom configuration-option profiler following state-of-the-art proposals from the literature [212, 241, 247]. We refer to this custom profiler as *performance-influence model*

8 Conceptualized one-liner for option specific measurements, the actual implementation uses a helper script that calculates the option-specific SDT probes.

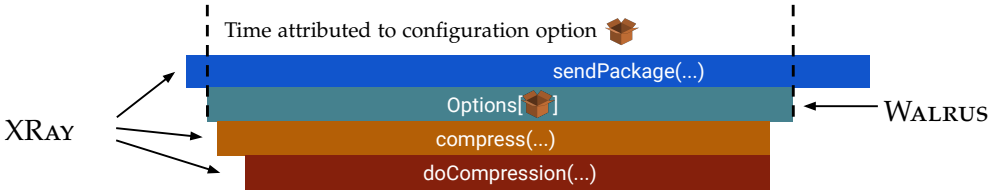


Figure 5.5: Visualization of an interleaved stack trace as an icicle chart that contains function level (XRay) and configuration level (Walrus) measurements. The stack grows top to bottom and the x-axis indicates the time spent.

tracer (PIM tracer), as for a single execution, the profiler collects a trace that represents the performance influences of different configuration-options as a performance-influence model (see Section 2.3.7.3). The idea is to build a dynamic configuration-option stack (i.e., in whose configuration-dependent region context we currently are) by tracking entries and exits of configuration-dependent code regions, measure the execution time of each configuration-dependent region, and accumulate the total time spent under the influence of a set of configuration option. For example, function `compress` (see Figure 5.1) is called inside a configuration-dependent region that is enabled by option in `sendPackage`. Now, when function `compress` contains itself a configuration-dependent region enabled by option the measurement needs to be attributed to both and .

Walrus supports this use case by adding the core logic of the profiler, which tracks the current program state, directly into the program and by inserting instrumentation code around each configuration-dependent region (see Section 5.3.1.1). This way, PIM tracer captures configuration-specific performance data on the fly, and produces a PIM trace for each execution. The collected PIM traces can be used for direct analysis or to generate a global performance-influence model that represents the whole system by merging multiple traces from different configurations together [212, 241, 247]. This profiling scenario illustrates that Walrus can even be used for very specialized tasks, where a custom profiler is needed to capture configuration-specific information. Furthermore, through such a custom approach, Walrus can also enable measurement libraries, such as likwid[9], by automatically injecting their measurement functions.

5.4 Evaluation

In this section, we demonstrate that configuration-aware state-of-practice performance profilers, enhanced through Walrus, are able to identify configuration-specific performance regressions even in real-world settings without inducing significantly more overhead than currently used industry-strength profilers. We demonstrate Walrus's *applicability* by correctly identifying 108 configuration-dependent

9 https://github.com/RRZE-HPC/likwid (Last accessed: July 27, 2023)

performance regressions using three state-of-practice profilers in a set of synthetic and real-world subject systems. We evaluate the *overhead* of configuration awareness on performance profiling by analyzing the measurement overhead caused by state-of-practice profilers, which are attached through Walrus.

5.4.1 Research Questions

Applicability. To demonstrate the applicability and flexibility of Walrus, we investigate several configuration-related performance regressions collected from a real-world HPC project and synthetic subject systems, utilizing different state-of-practice performance profiling tools. This leads us to our first research question:

> RQ_1 *How exact are configuration-aware performance profilers in identifying configuration-related performance regressions?*

Impact of configuration awareness. There is always a trade-off between accuracy and overhead when measuring the performance of a program. Measurements that aim at accuracy produce more overhead, which again can perturb the system and distort the measurements. Existing state-of-practice profilers navigate this trade-off space pursuing different strategies, with which developers can choose among different levels of accuracy. State-of-the-art configuration-aware performance analysis tools cannot navigate this trade-off space well. With Walrus, we aim at bridging the gap between configuration-aware performance analysis and state-of-practice performance profiling tools. To understand the performance impact of using configuration-aware state-of-practice performance measurement tools, we analyze the generated overhead of different Walrus-enabled profilers, which leads us to our second research question:

> RQ_2 *How much overhead is introduced by configuration-aware state-of-practice performance profilers, compared to their ability to attribute performance regressions?*

By answering these two research questions, we aim at demonstrating that state-of-practice profilers can be indeed empowered by Walrus to detect configuration-specific performance problems.

5.4.2 Experiment Setup

We address our two research questions by means of three state-of-practice profilers, plugged into Walrus and applied on 16 synthetic case studies and two real-world open source projects. For each subject system, we introduce multiple configuration-

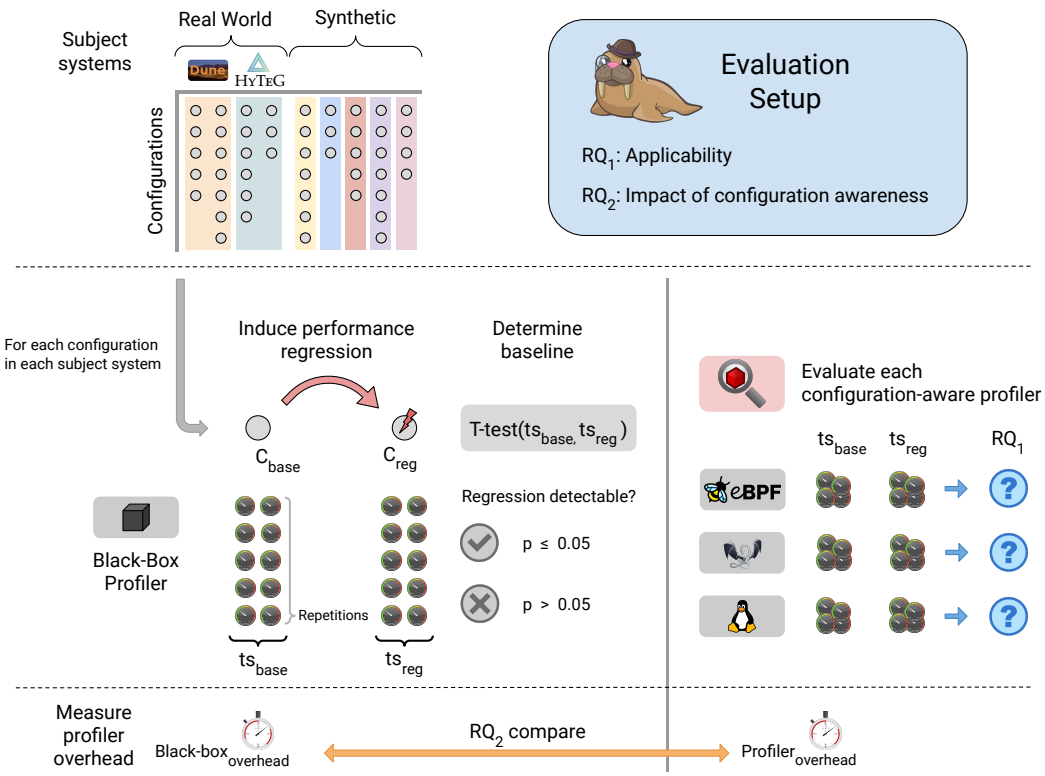


Figure 5.6: Overview of our evaluation setup. For each configuration in each subject system, synthetic and real world, we first measure whether a regression is detectable in the configuration to establish the ground truth. To answer RQ_1, we compare for each configuration-aware profiler whether it can also discover the regression. For answering RQ_2, we measure the time it took to execute the system without and with instrumentation to determine the profiling overhead.

specific performance regressions and use the WALRUS-enabled profilers to detect them, examining the accuracy with which the configuration-aware state-of-practice profilers can attribute the regression to configuration options and how much overhead they produce in the process. Figure 5.6 provides an overview of our evaluation setup. In what follows, we provide details on the profiling tools, subject systems, and regressions we use in our measurement setup, and our evaluation environment.

State-of-practice profilers. We selected a diverse set of state-of-practice profilers that utilize different implementation and measurement strategies together with a measurement approach, motivated by previous work. Specifically, following our application scenarios from Section 5.3.2, we selected three profiling technologies: (1) *e*BPF, LINUX's new performance and measurement infrastructure that supports a wide range of profiling strategies, (2) XRAY, a popular LLVM-based instrumentation profiler that instruments code at function level, and (3) PIM TRACER, a custom trace profiler that generates traces for building performance-influence models (see Section 5.3.2). For our XRAY scenario, we only report the configuration-specific

overhead and time measurements, to make the collected data comparable to the other scenarios. Since *e*BPF supports a whole range of profiling strategies, we select tracing setup that is similar to the other profilers and measures the time it took to execute a configuration-dependent region. As a baseline, we use an established black-box measurement strategy that measures all configurations in a brute-force manner, to determine which configurations and configuration options are affected by a performance regression [80, 212, 214]. Compared to Walrus-enabled profilers that can attribute performance measurements form a single execution, a brute-force approach solves the attribution problem by executing all configurations and calculating a performance-influence model (see Section 2.3.7.3), which is often infeasible in practice [114, 212]. Overall, we employ three profiling approaches in total, to demonstrate the applicability of Walrus.

Subject-systems selection. Overall, we selected a set of 18 subject systems, which differ in their size and application domain but, more importantly, in the way how they implement configurability. Table 5.1 lists all our subject systems. We divide them into *synthetic* subjects, which demonstrate a potentially problematic case (i.e., they stress the profiler or detection strategy), and *real-world* subject systems. Section 5.4.3 introduces the synthetic subject systems in detail. In addition, to show Walrus's applicability to real-world profiling use cases, we use the highly-configurable real-world subject systems dune and HyTeG. The dune framework is used for solving partial differential equations. dune uses compile-time configurability to enable its users optimize their solutions by providing different solvers, preconditioner, or grid types. The HyTeG (Hybrid Tetrahedral Grids) framework is used for large scale high performance finite element simulations [231]. Both dune and HyTeG are used in performance critical HPC applications, where their performance is often analyzed with various different profilers. Hence, due to their performance focus and high configurability, they are ideal real-world use case for configuration-aware performance profiling.

Performance regressions. We collected a set of patches that introduce a performance regression for each subject system. For our synthetic subject systems, we designed each regression to target a specific configuration-specific location by injecting slow-down code that introduces a controlled delay. For our real-world subject system dune, we designed performance regressions that regress code related to user-facing configuration options (i.e., solvers, preconditioners, and grid types).

Evaluation environment. We conducted all experiments on a compute cluster in an isolated and controlled environment to reduce the confounding factors that could influence our measurements. Each compute node consists of an AMD EPYC 72F3@3.70GHz CPU with 8-Cores and 256GB main memory, running a minimal Debian 11. We measured each analysis run 30 times and computed the mean run times to further reduce the influence of measurement fluctuations.

Table 5.1: List of selected subject system, showing for each system: the size of the configuration space $|\mathbb{P}(\mathcal{C})|$, the number of regressions R, the systems size in lines of code (LOC), and the configuration mechanisms used in its implementation.

		Subject System	$\lvert\mathbb{P}(\mathcal{C})\rvert$	R	LOC	Conf. Mechanism
Real World		DUNE	10	7	199 069	Compile Time
		HYTEG	12	5	1 074 886	Compile Time
Synthetic	Static Analysis	Field Sensitivity	5	2	499	Load Time
		Flow Sensitivity	6	1	70	Load Time
		Context Sensitivity	8	2	73	Load Time
		Whole Program	8	3	125	Load Time
	Implementation Pattern	CRTP	8	34	223	Compile Time
		Policies	12	12	193	Compile Time
		TemplateSpecialization	9	12	170	Compile Time
		TraitBased	16	13	253	Compile Time
		Runtime	5	2	499	Load Time
		Combined	5	2	499	Compile/Load Time
		Template	5	2	976	Compile Time
		Template2	5	2	497	Compile Time
	Dynamic Analysis	Dynamic Dispatch	6	4	128	Load Time
		Recursion	4	2	71	Load Time
		Hot Loop Code	4	1	76	Load Time
	Configurability	Option Interaction	8	2	56	Load Time

5.4.3 Synthetic Subject Systems

We primarily structure our evaluation around a set of 16 synthetic subject systems. Real-world subject systems have the advantage that they provide a realistic basis for our evaluation. However, from a selection of real-world systems, one cannot be sure that all analytical edge cases are covered. Hence, we designed 16 synthetic subject systems, to ensure that WALRUS can handle a wide variety of implementation strategies and know edge cases. Table 5.1 gives a detailed overview of our synthetic subject systems. We devised all our synthetic subject systems in an informed way: by collecting edge cases from literature, by gathering implementation patterns and strategies from known open-source projects, and by investigating problem cases of state-of-practice profilers. Overall, we divide our synthetic subject systems into four categories, based on the area of edge cases they target:

Static analysis. In the *static analysis* category, we encode important static analysis properties into small synthetic programs. As introduced in Section 2.3.2, analysis properties, such as context sensitivity or field sensitivity impact what a static analysis can find. The synthetic subject systems in the static analysis category ensure that the taint analysis $\mathbb{T}$ (introduced in Section 4.2.3), which we utilized for detecting

configuration-dependent code in Walrus (see Section 5.2.2), has the right analysis properties to correctly located configuration-dependent regions.

Implementation pattern. Complementary to the static analysis synthetic subject systems, we built synthetic subject systems that utilize different compile-time implementation patterns, to ensure that our localization strategy can correctly handle compile-time variability. Evaluating different compile-time implementation patterns is important, since the compile-time localization approach presented in Section 5.2.3 is different from the localization approach for load-time variability. Most notably, compile-time variability is already detected in the compiler front-end and does not employ a static analysis afterwards to detect configuration-dependent regions. Furthermore, as both compile-time and load-time implementation patterns differ with regard to the language mechanism used to implement them (see Section 2.1.4) and produce differently scattered configuration-dependent code, we should ensure that both types are included in our evaluation. To collect a set of common implementation patterns for compile-time variability, we analyzed our real-world subject system and additional projects on github, and collected additional patterns from literature [169, 176, 238].

Dynamic analysis. In the third category, we focus on edge cases that arise from the dynamic execution behavior of a program and can impact the correctness, accuracy, and overhead of dynamic analyses (e.g., they can influence the accuracy and overhead of performance profilers). Edge cases in this category include, for example, dynamic dispatch and recursion, that can require special handling from a dynamic analysis to correctly attributed the measured information, and synthetic subject systems, such as "Hot Loop Code", that require frequent measurements inside a hot region of code, likely producing significantly more overhead. Through the analysis of such synthetic subject systems, we guarantee that know edge cases for performance profilers, such as the ones introduced in Section 5.3.2, are included in our evaluation.

Configurability. The fourth category focuses on edges cases that arise from the configurability of software systems. To identify such edge cases, we searched the recent literature that focuses on the problematic cases in the analysis of configurable software systems [124, 153, 212]. We found that interactions between configuration options are a prevalent problem when analyzing configurable software system (see Section 2.1.5). Based on the gathered information, we devised a synthetic subject system that focuses on different configuration interaction patterns, varying how configuration-dependent code interacts. For example, we included different levels of interactions that arise from differently entangled and nested configuration-dependent code (i.e., we vary *how* code interacts).

Taken together, our collection of synthetic subject systems ensures that known edge

cases are included in our evaluation and, by that, our evaluation can demonstrate that WALRUS is able to conceptually handle such edge cases.

5.4.4 Applicability (RQ_1)

The goal of RQ_1 is to show the applicability of configuration-specific performance measurements to detect performance regressions in a variety of software systems.

Operationalization. To answer RQ_1, we show that configuration-specific performance regressions can be correctly identified by WALRUS-enabled configuration-aware state-of-practice performance profilers. We set up 108 configuration-specific performance regressions in 18 different subject systems. Based on each performance regression, we created a patch file that introduces the regression into the system in a controlled way. We created a regression baseline that indicates which configurations are affected, by running a black-box performance profiler exhaustively over the whole configuration space [114, 125, 247]. The black-box profiler measures the total execution time of a given configuration 30 times before and after the regression was introduced. We consider a regression between the original code and the patched one as significant based on the student's t-test (T-test). Furthermore, we choose 100*ms* as baseline-regression sensitivity threshold to prevent miss detection (i.e., we only consider regressions that are larger than 100*ms*). This way, we obtain a stable baseline set of configuration-specific regressions spanning 18 subject systems, for which we know in which configuration a profiler should detect a regression.

We compile each subject system with WLANG, once for each profiler (see Section 5.3.2), and we add the profiler-specific information and instrumentation code. Next, we profile each binary with each profiler, using a standard workload for the system that was pointed out by the developers or extracted from previous work. For our experiments, we do not vary workloads, as workloads do not directly affect the profiler or instrumented measurement code (see Section 5.5). For each configuration-aware profiler and configuration, we determine whether there was a performance regression in the same way as with our black-box analysis by applying a U-test, except that only the configuration-option-specific measurements are considered. To reduce the impact of measurement noise, we also define a sensitivity threshold of 100*ms* and 1% for configuration-option regressions. That is, we only consider a configuration option as regressing when the time different between the old and new measurements is significant and larger than our sensitivity threshold. Afterwards, we compare the profiler detected regression with the baseline confirmed regressions, computing classification precision and recall. This way, we determine how precise configurability-aware state-of-practice profiler can identify performance regressions by only measuring a single configuration and correctly attributing the profiler's measurements to configuration-specific code.

Results. Table 5.2 shows the results for all profilers, regressions, and subject systems, where Figure 5.7 shows the distributions for precision and recall. Each data point denotes the precision/recall a profiler achieved for a regression in a subject system. Precision indicates the portion of configurations that where correctly identified as regressing, and recall indicates the portion of regressed configuration that where correctly identified. Note that, for each regression and subject system, a profiler determines for each configuration whether it has regressed only based on the configuration-specific measurements (that were collected for this specific configuration), compared to the black-box baseline (where the measurements of all configurations are combined).

With regard to precision, for all three configuration-specific profilers we obtain a similar picture. In most cases, the profilers correctly identified regressing configurations, however, some configuration where incorrectly identified as regressing (false positives). Most notably are our two real-world subject systems DUNE and HyTeG, where multiple configurations where wrongly detected as regressing. A manual inspection of the miss-predictions showed that most of them are caused by between-experiment-run noise that is slightly above the chosen thresholds. Overall, for all the three profilers the mean precision was in the range of 0.95 to 0.96.

Figure 5.7 shows with regard to recall an even better picture. All three profilers were able to correctly identify nearly all regressing configurations for all subject system, based only on the configuration-specific performance measurements. The only significant exception where many regressions are missed is the *e*BPF profiler when analyzing DUNE. An in depth analysis of the measurement process showed that *e*BPF drops many measurements when analyzing DUNE, as the frequency with which measurements are produced is too high for some configuration-dependent code regions. Internally, *e*BPF uses a non-blocking ring buffer to store measurement events which have not been persisted to file yet. So, if the frequency with which new measurements are produced exceeds the speed in which measurements are persisted for too long, measurement events are dropped from the ring buffer. This is different from the other two profilers where the main thread and the collection of new measurements is blocked, should the internal ring buffer be full. Overall, for all the three profilers the mean recall was in the range of 0.97 to 1.00.

By analyzing the distribution of precision and recall over different subject systems, we observe that most of the subject systems could be correctly instrumented and analyzed. Hence, as most of the regressions could correctly be located with different profilers, we conclude that WALRUS was able to accurately locate configuration-specific code and weave in measurement code of each profiler and, by that, solve the attribution problem. This way, the WALRUS-enabled profilers were then able to correctly measure configuration-specific performance regressions.

Answer RQ_1 | *Our experiments show that* WALRUS*-enabled profilers are capable of correctly identifying configuration-specific performance regressions through configuration-specific measurements.*

Table 5.2: Our evaluation results show for each subject system classification precision and recall as well as time and memory overhead. The color gradient for precision and recall indicates how precise regression configurations could be identified. The color gradient for the overhead measurements indicates how much overhead was produced, compared to the other subject systems.

	WXray					PIMTracer					eBPFTrace				
	P	R	Δ Time %	Δ Memory %	Δ Memory Kbyte	P	R	Δ Time %	Δ Memory %	Δ Memory Kbyte	P	R	Δ Time %	Δ Memory %	Δ Memory Kbyte
DunePerfRegression	0.74	0.98	9.11	2.71	516	0.78	0.98	18.95	2.50	476	0.82	0.51	9.11	2.71	516
HyTeg	0.84	1.00	9.47	34.93	6625	0.96	1.00	8.71	34.91	6621	0.80	1.00	9.47	34.93	6625
SynthCTCRTP	1.00	1.00	0.00	17.02	547	1.00	1.00	0.00	16.75	538	1.00	1.00	0.00	17.02	547
SynthCTPolicies	1.00	1.00	0.00	16.67	537	1.00	1.00	0.00	16.81	541	1.00	1.00	0.00	16.67	537
SynthCTTemplateSpecialization	1.00	1.00	0.00	16.66	534	1.00	1.00	0.00	17.04	547	1.00	1.00	0.00	16.66	534
SynthCTTraitBased	1.00	1.00	0.00	16.75	538	1.00	1.00	0.00	17.10	550	1.00	1.00	0.00	16.75	538
SynthDADynamicDispatch	1.00	1.00	0.00	14.74	491	1.00	1.00	0.00	14.10	469	1.00	1.00	0.00	14.74	491
SynthDARecursion	0.83	1.00	-0.42	13.71	462	0.83	1.00	-0.41	15.20	512	0.83	1.00	-0.42	13.71	462
SynthFeatureInteraction	1.00	1.00	0.09	15.76	515	1.00	1.00	0.05	15.50	507	1.00	1.00	0.09	15.76	515
SynthIPCombined	1.00	1.00	-0.24	3.30	542	1.00	1.00	-0.53	3.23	531	1.00	1.00	-0.24	3.30	542
SynthIPRuntime	1.00	1.00	0.57	3.01	496	0.75	1.00	0.49	3.38	555	1.00	1.00	0.57	3.01	496
SynthIPTemplate	1.00	1.00	1.24	2.75	454	1.00	1.00	2.19	3.28	540	1.00	1.00	1.24	2.75	454
SynthIPTemplate2	1.00	1.00	0.68	3.36	551	1.00	1.00	2.65	3.80	624	1.00	1.00	0.68	3.36	551
SynthOVInsideLoop	1.00	1.00	0.00	14.82	487	1.00	1.00	-0.00	16.50	542	1.00	1.00	0.00	14.82	487
SynthSAContextSensitivity	1.00	1.00	0.00	14.03	483	1.00	1.00	0.01	14.26	491	0.83	1.00	0.00	14.03	483
SynthSAFieldSensitivity	0.75	1.00	2.49	2.91	478	1.00	1.00	0.43	3.02	496	1.00	1.00	2.49	2.91	478
SynthSAFlowSensitivity	1.00	1.00	0.00	12.68	438	1.00	1.00	0.00	13.66	472	1.00	1.00	0.00	12.68	438
SynthSAWholeProgram	1.00	1.00	-0.03	14.44	489	1.00	1.00	-0.04	14.69	498	1.00	1.00	-0.03	14.44	489
Total	0.95	1.00	1.28	12.24	844	0.96	1.00	1.81	12.54	862	0.96	0.97	1.28	12.24	844

P = Precision; R = Recall; P/R color gradient: low high; Overhead color gradient: low high;

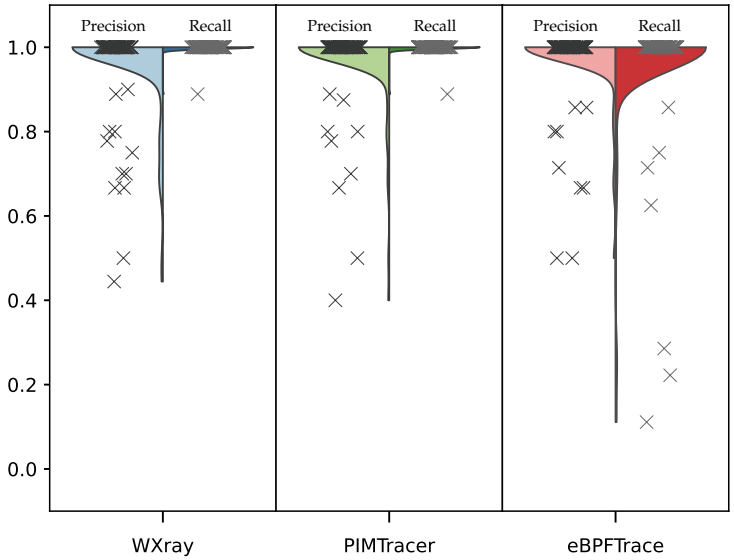


Figure 5.7: Comparison of precision and recall between different configuration-specific performance profiling approaches on multiple subject systems. Each data point (x) denotes the precision/recall value a profiler achieved for a regression in a subject system (i.e., how accurate the profiler could determine if a configuration of the subject system regressed based on configuration-specific measurements). The violin plots (precision left/recall right) visualize the likelihood for each profiler that a specific precision/recall value was observed. This way, visualizing how much precision/recall depend on the subject system, to show the subject system's influences on the profiler.

5.4.5 Impact of Configuration Awareness (RQ_2)

In order to enable users to profile programs without too much overhead, state-of-practice performance profilers often employ several tricks and low-level optimizations to reduce the measurement overhead and the impact measurement code has on the measured code. For example, *e*BPF-based profilers move measurement code and data aggregation into the kernel context, this way significantly reducing the overhead and the number of context switches when performing measurements. With RQ_2, we aim at demonstrating that Walrus does not hinder profilers to use these tricks and optimizations, and enables configuration-specific performance profiling with a reasonable amount of overhead. That is, we expect the configuration-specific profiling overhead to be at least of the same magnitude as non-configuration-specific profilers [15, 168, 173, 248].

Operationalization. To answer RQ_2, we measure the side effects that different instrumentations have on the system under measurement. For this purpose, we instrument 18 configurable subject systems with three different profiling strategies using Walrus, producing differently instrumented system variants. In addition, we

include a baseline variant by compiling the program without any instrumentations. Each of the different variants are then executed—profiling the target system—and additionally profiled by an external low-overhead black-box measurement tool.[10] The external profiling tool measures the memory consumption and the time it took to execute the white-box profiling run.

In our experiment setup, the part that varies across the variants is only the inserted measurement code. Hence, deviations in the execution time or memory footprint can be attributed to the chosen profiling technology. For both time and memory, we report the relative amount compared to the unmodified baseline.

Results. Table 5.2 depicts the results of our overhead measurements. Each cell is colored with a gray-scale color gradient that visualizes the overhead intensity.

With regard to time overhead, we observe that smaller synthetic subject systems show no impact from the profiler measurements (i.e., for most synthetic systems the measurement overhead is negligible and often below the measurement variance). For both real-world subject systems, our measurements show an overhead in the range from 8.71% to 18.95%, varying with subject system and profiler. Note that the overhead measurements show only the additional overhead introduced to the normal execution of the program through configuration-specific profiling. That is, time spent in parallel running measurement threads is not counted as overhead. Furthermore, a profiling run that produces additional measurements, such as function-based XRay measurements, and additional configuration-specific measurements would produce more overhead.

With regard to memory overhead, we observe a relatively uniform overhead between profilers and subject systems. The absolute memory overhead is relatively constant with ∼0.5 MB and does not vary much between subject systems. The only exception is HyTeG, where the measurement overhead is larger (∼6.6 MB) but still reasonably low.

Overall, configuration-specific performance measurements, independent of the evaluated profiling scenario, stayed in a reasonable overhead range for both time and memory. Hence, we can conclude that state-of-practice performance profiler can be used in a configuration-specific way, as they introduce only a reasonable amount of overhead compared to regular profiler usages [15, 168, 173, 248].

Answer RQ_2 | *Our experiments show that* Walrus*-enabled profilers produce a reasonable amount of overhead that is of the same magnitude with existing profilers.*

10 We utilize the low-overhead *time* command from GNU time.

5.5 Threats to Validity

Internal validity: A threat to internal validity arises from the selection of our performance regressions, as they might not be suitable for the selected profilers to detect. We address this in two ways: We select regressions that were verifiably located in configuration-specific code and we ensure through our black-box baseline that a regression is at least detectable through measurement approaches used in previous work [114].

The precision of performance measurements can be impacted by various external sources, such as CPU frequency fluctuation, OS-level interferences from other processes, or interferences from other users. To reduce such inaccuracies in our measurements, we set up an isolated and controlled measurement setup, as suggested by the literature [126]. Through our cluster setup, we prevent external interferences and isolate our measurement job. Furthermore, we set up a minimal OS that executes only our jobs, controls CPU frequencies, and repeat all our measurements 30 times.

Another thread to internal validity arises from our regression baseline, as our baseline could wrongfully identify a configuration as regression without an actual regression. We address this threat, as described before by reducing measurement noise and by adding a sensitivity threshold of $100ms$. We choose this threshold value in an informed way by selecting a value that is smaller than any of our introduced regressions (i.e., smaller than $130ms$) but larger than our measurement variance. Furthermore, a lightweight sensitivity analysis showed that threshold values from $30ms$ to $120ms$ produce nearly identical evaluation results overall ($\pm$ 0.01 precision/recall).

External validity: Our evaluation depends on detecting configuration-specific code. Hence our results might differ for other configurable systems. To minimize this threat, we use established localization techniques proposed in the literature [147] and evaluated Walrus on a real-world subject system used in high-performance computing as well as several synthetic subject systems. We focused on 16 synthetic case studies that contain adversarial analysis and implementation patterns, as we cannot guarantee that known special cases are present in the real-world subject system. Overall, our results show that configuration-aware profiler, enabled by Walrus, can correctly identify configuration-dependent performance regressions.

Furthermore, our evaluation depends on the selection of state-of-practice performance profilers, as profilers vary in precision, implementation technique, and produce different amounts of overhead. Hence, we cannot generalize our results to all profilers. However, our evaluation demonstrates that Walrus is able to make state-of-practice profilers configuration aware with reasonable additional overhead, even when the profilers use different implementation and measurement techniques.

5.6 Discussion

In this section, we illustrate the advantages of combining state-of-practice profilers through Walrus with configuration knowledge along with potential limitations.

5.6.1 Configuration-Aware Performance Analysis

Configuration-wise profiling. In order to determine the performance impact of individual configuration options, previous configuration-focused performance-analysis approaches have either used state-of-practice black-box profilers, with the disadvantage of requiring measurements for a large set of configurations [114, 213], or white-box localization approaches together with handwritten measurement code on a few configurations [240, 241]. Walrus combines the best of both worlds, integrating state-of-practice white-box profilers with white-box localization approaches (i.e., Walrus solves the attribution problem and relays this information to the profiler). Our evaluation of RQ_1 shows that precise configuration-specific performance measurements can be extracted directly from the execution of a configuration and used to detect configuration-specific performance regressions.

Profiler reuse. Configurability in software system is orthogonal to performance analysis, as all performance aspects can be influenced by configuration choices. Furthermore, as profiling different performance aspects or in special environments requires different state-of-practice performance profilers, conceptually combining configurability with different profilers is important.

In our evaluation, we demonstrate that Walrus is able to conceptually and practically combine state-of-practice profilers with configuration information. By decoupling the concrete profilers through lightweight adapter code from the localization approaches, Walrus is able to support a wide range of different profilers and profiling scenarios (see Section 5.3.2) together with different implementation strategies for configurability. Furthermore, RQ_2 shows that Walrus enabled profilers do not produce unreasonable amount of additional overhead. This way, Walrus gives users the freedom to select the best profiler for their use case but also enables them to contextualize the profilers measurements with regard to configurability.

5.6.2 Limitations

External performance regressions. Walrus currently focuses on attributing program-specific configuration options and does not instrument operating system code. Hence, in a standard setup, Walrus cannot attribute OS-specific configuration options to performance regressions.

Short-circuiting alternatives. Logical OR expressions (e.g., `a || b`) in conditionals short-circuit. When the first part of the condition is true, the other one does not get evaluated. In such a case, we cannot unambiguously identify through static analysis to which configuration options a measurement should be attributed. In general, a single execution of such a conditional cannot determine, without tracking the actual values of the configuration options through an additional dynamic analysis, which configuration options where involved in executing the controlled block. This currently limits Walrus to attribute the performance measurements related to such conditionals to specific configuration options.

Implicit flows. Walrus's load-time location approach currently does not handle implicit flows, which would model indirect dependence on configuration options [122]. This practically limits us in detecting performance regressions that arise in a part of the program that is indirectly influenced by a configuration option. However, this is only a practical limitation of our current load-time location implementation and does not influence the core approach behind Walrus, as Walrus is agnostic to the concrete localization approach used.

Dynamic Language Features Currently, Walrus does not handle dynamic program features, such as exceptions and reflection, explicitly. This is a practical limitation of Walrus's implementation that could be improved in the future by adding logic to the dynamic analysis side (i.e., the profilers) but this was not necessary for our synthetic subject systems and our real-world subject system dune.

5.7 Related Work

Configuration-dependent code localization. There already exists a large body of work from the recent years that introduces varying approaches to localize and analyze configuration-dependent code. Compile-time configurability can be analyzed through tools, such as TypeChef [118], SuperC [71], cppstats [144, 146], or Morpheus [145]. With Lotrack, Lillack et al. [147] demonstrate a localization approach that tracks load-time configuration options within Android apps. Walrus's localization approach is conceptually modeled after Lotrack, using a similar taint analysis. The work from Rhein et al. [195] unifies both compile and load-time configurability by encoding compile-time configuration options into load-time configuration options, creating a 150% program. Furthermore, work from Garvin and Cohen [70] analyzes interaction faults and their dependence on feature-dependent specific code regions. Their work introduces the term variability region, a similar abstraction to our configuration-dependent regions.

With Walrus, we build and incorporate previous research, introducing a common abstraction for compile and load-time configurability with the goal of making configuration-dependent code accessible in the compiler, enabling configuration-aware performance analyzes.

Configuration-aware performance analysis. The area of configuration-aware performance analysis is conceptually subdivided into black-box and white-box approaches, where only the later take into account the source code of a program. Previous work from Siegmund et al. [212] learned performance-influence models based on black-box measurements, which was later extended to white-box profilers by Weber et al. [247]. Furthermore, Velez et al. [240] developed ConfigCrusher, an approach that utilizes a static data-flow analysis to run white-box performance measurements for building performance-influence models. Walrus's approach is in the same vain but tries to solve the problem of measurement overhead not by relaxing a regions scope but by utilizing state-of-practice performance profilers. Comprex, another white-box approach to build performance-influence models, utilizes a dynamic taint analysis with the goal of reducing measurement cost. ConfProf [84], an iterative white-box performance profiling approach, automatically determines configuration-dependent performance-critical code regions and the most influential configuration options. Walrus differs as our goal is to measure all specified configuration-options and enhance existing state-of-practice performance profilers to visualize their information in a configuration-aware manner. Hence, a developer selects the configuration options to analyze and the performance tool that should be used for the analysis. So, where ConfProf aims at a specific use case, Walrus is flexible by enabling a multitude of different profilers. Furthermore, Chen et al. [34] detect inconsistent configuration-specific regressions in software histories. They demonstrate that configuration-specific performance influences change throughout the development of the software system, an interesting approach that starts to combine the revision history of a project with configurations-focused performance analysis. Our evaluation shows that Walrus can enable profiler to detect regressions between two revisions. That is, through Walrus, state-of-practice profilers can now be combined with their approach to build full historical analyses.

Reducing measurement overhead. Another approach to reduce the measurement cost of configuration-aware performance profiling, called Performance-Detective, was proposed by Schmid et al. [206]. Performance-Detective improves the experiment design by deducing insights about program parameters (i.e., configuration options, avoiding unnecessary measurements and repressions). So, where Walrus focuses on reducing measurement overhead by enabling state-of-practice performance profilers, their work focuses on overall reducing measurement cost by selecting a better experiment setup.

5.8 Summary

In what follows, we contextualize the contributions of the work presented in this chapter to the field of configurable software systems, and with regard to the general contribution that is makes to our overall thesis.

State-of-the-art performance profilers have the drawback that they cannot incorporate configuration knowledge. This means, they cannot attribute their measurements to configuration choices. Hence, detecting performance bugs in configurable software system difficult, as developers have no direct information about *which* and *how* configuration options are involved in a performance bug.

Walrus resolves this problem by detecting configuration-dependent code and passing configuration knowledge through its instrumentation interface onto profilers. This way, profilers can use configuration knowledge to collect and interpret their measurement data, enabling configuration-aware performance measurements.

Through our evaluation that analyzed the two real-world subject systems used in high-performance computing (dune and HyTeG) and 16 additional synthetic subject systems, we have demonstrated that Walrus can enable three state-of-practice profilers to detect configuration-specific performance regressions.

Contribution to the field of configurable software systems. Our evaluation demonstrates that, with Walrus, we can analyze the performance of configurable software systems using state-of-practice performance profilers. Our results show that through Walrus multiple performance profilers where able to identify configuration-specific performance regressions with reasonable amounts of overhead. This demonstrates that Walrus can combined different state-of-the-art configurability localization approaches with state-of-practice performance profilers, enabling configuration-specific performance analysis.

Walrus introduces a new way of empowering industry-strength performance profilers with configuration knowledge. This way, we make a step into the direction of better understanding the performance landscape of configurable software systems.

General contribution to this thesis. The work presented in this chapter employs code regions in two different ways. First, to reuse an already existing static taint analysis to detect terminator instructions that are influenced by configuration variables (Section 5.2.2). This application shows how code regions enable the reuse of advanced program analyzes that initially targeted a different problem. Second, to model configuration-dependent code in a program (Section 5.2.1). This application shows that configuration variability (*space*), encoded through configuration options, can be modeled through code regions. Afterwards, in Section 5.3, we demonstrate how code regions enable dynamic analyses—performance profilers in our case—to capture code region-specific runtime information.

6 Concluding Remarks

In the following, we conclude this dissertation. Afterwards, as the research presented in this dissertation lays the foundation for better integrating state-of-the-art program analysis into variability research, we outline future research directions and ideas that can now be realized through our work.

6.1 Summary of Contributions

Over the last decades, the complexity of modern software systems has been growing rapidly through faster development, larger teams, and more configurability. As software systems become ever more prominent in everyday lives and society, we expect this trend to accelerate in the future. However, analyzing and understanding the evolution of software systems in the context of configurability is difficult, not only because *time* and *space* dimensions interact and influence each other, but also because state-of-the-art program analyses cannot be easily applied to such systems.

To improve the current situation, we developed a conceptual abstraction of code regions that unifies variability information from both dimensions and that presents it to state-of-the-art program analyses. Code regions decouple domain-specific variability information from the program analysis through a uniform interface and separate the analysis semantics from the variability semantics. We developed a practical implementation of our uniform abstraction of code regions that is able to apply advanced state-of-the-art program analyses to solve previously challenging research problems. Through the innovation of abstracting variability information and combining it with state-of-the-art program analyses, we were able to make contributions to different research fields. In particular, the research presented in this dissertation made the following contributions to the fields of socio-technical analysis, software repository mining, the analysis of configurable software systems, and program analysis in general:

Software repository mining. State-of-the-art software repository analyses often do not have precise information about a program's operational semantics at their disposal. However, such information is essential for understanding how software behaves, for example, to infer developer connections or understand the impact of

code changes. To provide such information and to bridge the gap between repository information and program semantics, we have developed SEAL.

With SEAL, we have presented an approach that combines historical development data, mined from software repositories, with a program's operational semantics. This way, we can now, for example, infer previously hidden connections between developers. The combination of high-level information from software repositories with low-level program semantics opens up new research avenues to understand software evolution, by incorporating the inner workings of a program. Our work contributes to the field of software repository mining by introducing a new *method* to extract a new *kind* of data from software repositories that combines repository information with a program's operational semantics.

Socio-technical analysis. Results from state-of-the-art program analyses can be hard to interpret and act upon, especially in a development context that encompasses several teams and a many developers, as understanding the results or building a needed fix can require the involvement of multiple different developers. Hence, it is important to incorporate the socio-technical context of a software project into analysis findings. However, combining both is a non-trivial task, as socio-technical information is not present on the abstraction level that program analyses typically operate on.

To address this shortcoming, we have built SEAL in such a way that the computed repository information can be freely attributed to existing analyses results. This way, the analysis can automatically be contextualized with socio-technical information extracted from the project repository. Our work contributes to the field of socio-technical analysis by providing a new information source that incorporates a program's operational semantics into analyses, but also by enabling program analyses to contextualize their results with socio-technical information.

Configurable software systems. The configurability of modern software systems can lead hard to diagnose and resolve performance bugs. A major reason for the difficulties in debugging configuration-related performance bugs is the inability of current state-of-practice performance profilers to account for configurability explicitly. That is, state-of-practice profilers cannot attribute their measurements to configuration choices and leave it up to the developer to contextualize the measurements with configuration knowledge.

To bridge this gap, we have developed Walrus, a code region-based dynamic analysis framework that automatically combines configuration knowledge with state-of-practice performance profilers. Embedded into a modern compiler, Walrus detects configuration-dependent code and weaves profiler-specific measurement code into it, to contextualize the performance profiler's measurements with configuration knowledge. Our work contributes to the field of configurable software systems by conceptually combining compile-time and load-time configurability with state-of-practice performance analysis tools. This way, bringing together a decade of

configurability localization research with modern industrial strength performance analysis tools.

Program analysis. Advancements in the area of program analysis often do not get carried over to users in the wider research community quickly. This is due to the inherent theoretical and practical complexities modern program analysis entail. As a result, software engineering researchers cannot capitalize on the most modern analysis techniques.

Through our abstraction of code regions and the integration of state-of-the-art program analyses frameworks, such as PHASAR, we enable high-level research approaches to leverage novel and up-to-date program analyses. Furthermore, through the decoupling between domain-specific variability information and analyses, we make analyses freely combinable and reusable, by that, reducing the burden on software engineering researchers. This way, we make a small step into the direction of better integrating modern state-of-the-art program analysis into software engineering research.

The conceptual abstraction of code regions that we developed through this dissertation enables reusable and combinable program analysis that can be applied to variability but also potentially to a multitude of other domain-specific concepts. Through our work, we are now able to analyze different types of variability in concert with state-of-the-art program analyses.

6.2 Future Work

The work presented in this dissertation lays the foundation for a static- and dynamic-analysis-driven investigation of different types of variability and more program-analysis-driven research, in general. In what follows, we lay out the four most promising directions in which our work could be extended.

6.2.1 Extending Code Regions to Other Domains

Code regions abstract from specific high-level information, variability information in our case. Throughout our work, it became clear that code regions are also suitable to represent information from other domains in a uniform way. For example, for architectural analysis we can model architectural abstracting as code regions, or for security we can implement common information leakage analysis by modeling private data locations as well as user facing APIs as code regions. We give a more detailed overview of potential information that could be modeled through code regions in Section 3.7. When looking into additional applications of code regions, we noticed that there are two categories of additional code regions that could be

conceived: Primary code regions that are in the direct focus of an investigation, for example, security-critical regions that indicate where private data are stored. But there are also complementary code regions that inject relevant information into the analysis (i.e., code regions that, when combined with primary code regions, enhance the statements one can make). Take, for example, test-specific coverage data, modeled as a code region. This additional information could be valuable to many analyses and complement the interpretation of their results. Clearly, what defines a primary or complementary code region depends on the research question one wants to address. However, seen from a pragmatic point of view, adding complementary code regions for information that is already present or can be integrated easily into LLVM could be beneficial for multiple research questions. As by adding a new type of code region, multiple new questions can be addressed through the combinatorics with other code regions and different analyses (i.e., as multiple different types of code regions can be used simultaneously in integrated analyses, adding a new one enables other approaches to incorporate them). In general, code regions counteract the loss of domain-specific information, which is incurred during the lowering to an intermediate representation, by preserving or recovering the lost information and relating it to the intermediate representation.

6.2.2 Enhancing Code-Region Interactions Through Data-Flow Path Information

In Section 3.3, we introduced code-region interactions capturing when two code regions affect each other with regard to a specific interaction relation. In Chapter 4, we applied code regions to capture data-flow-based commit interactions that encode if there is a data flow from code belonging to one commit to code belonging to another. What is currently missing in this picture is the data-flow path that actually connects the two commits or, described from an analysis point of view, the data-flow path that connects the specific instructions inside the commit-based code regions. Data-flow path information can be highly valuable to understand of *why* certain interactions happen and, furthermore, to understand *what* lies on the path.

Gaining an understanding *why* an interaction happens is valuable in many cases. For example, if an interaction is unexpected, path information helps to find the location in a program that cause the interaction (e.g., if two developers from previously unconnected teams now have a connection, path information can be used to locate the code that causes this interaction).

Furthermore, determining *what* lies on the data-flow path of a specific interaction can be used to infer additional information about an interaction. For example, through the data-flow path, one can collect all configuration options from the configuration-dependent code regions that are on the path. In another example that uses architecture code regions (i.e., regions that encode to which architectural layer a

specific piece of code belongs), one can collect in which sequence architectural code regions lie on the path and from that infer if an architectural violation occurred.

By extending code-region interactions with path information, we can infer more information about *why* interactions happen and *what* lies on the paths of an interaction, enabling researchers to investigate a variety of new questions about the evolution of configurable software systems through code regions.

6.2.3 Analyzing Code Regions in a Historical Context

Currently, code regions and information derived from them, such as commit interactions, are computed for a specific revision of a software system. However, up to now, they have not been analyzed in the historical context of the software system, meaning, how they themselves change over time. The historical context opens up two research problems: First, we need to analyze how the information derived from code regions, such as commit interactions, changes over time, as understanding changes in such data could contain valuable information. Second, we need to find a way to trace code regions over time to understand how specific regions change (i.e., we need to identify the same region in multiple revisions).

With regard to the first problem, we have seen that, from a data-flow perspective, code changes can have varying effects on the rest of the software system. However, until now, we have only analyzed these changes in isolation, that is, we only have analyzed a specific revision without considering the order in which they are developed. This gives us a snapshot view on the system, but it does not tell us how the system evolved into this state. Take, for example, central code, where we can locate changes for a revision that modify or affect code that currently is central in the data-flow dependency structure of the system. What we do not know is how and when the code became central. For example, was a function introduced as central code or did changes to the system that used the function let it become central over time? We expect that change patterns in the history of configurable software systems can be used as valuable signals to determine new metrics for assessing the potential impact a change could have (e.g., to determine fault-prone artifacts or improve maintenance efficiency [129]).

For the second problem, to be able to trace code regions over multiple revisions, we need to find a way to uniquely identify a code region. Being able to trace code regions through multiple revisions would enable us to analyze how a region changes over time, which allows analyses of various kinds to better incorporate software evolution, for example, detecting when the set of configuration options that control a configuration-dependent region change, or comparing the time spent in a configuration-dependent region with the previous time spent in the same region. Currently, we have no reliable way of matching a code region to it's counterpart in a previous revision. Note, due to the fact that practical programming languages have no inherent concept of program revisions, such a matching will be a heuristic [209]. Even for language concepts such as function names there is no guarantee that a

function with the same name in another revision identifies the same function [139]. For many practical purposes matching function through their name is good enough. Developing a similar heuristic to match code region from different revisions would enable us to analyze how code regions change over time. For example, we are currently limited to compare the time spent on a configuration-option level instead of a region level, which would be interesting as we would be able to locate configuration-specific performance regression more precisely.

Overall, we expect that considering code regions and the information derived from them in the historical context of a software system can provide new and unique insights into the evolution of configurable software systems.

6.2.4 Applying Commit-Interaction Data to Socio-Technical Network Analysis

Our work introduces the new concept of commit interactions to analyze the dependencies between code changes and their developers. We have shown that we can infer interesting new connections between developers that previous state-of-the-art approaches (e.g., Joblin et al. [105] or Mauerer et al. [151]) that are based on co-occuring changes or call-graph interactions where not able to find. Current approaches in socio-technical network analyses base their coupling metrics on co-changes, determined on a file or function level, or on semantic coupling between words, and do not incorporate data-flow information [75, 101, 104, 105, 154]. However, we did dig deeper into the facts that can be gained from commit interactions for socio-technical network analyses. We argue that data-flow-based commit interactions carry valuable information for state-of-the-art approaches. However, it needs to be determined experimentally whether and how much valuable information is carried in this new type of interactions. Furthermore, the value gained depends on the specific socio-technical approach, meaning, for some approaches the additionally discovered developer interactions might be highly relevant, where for others they do not provide relevant additional information. For example, we expect that commit interactions can inform socio-technical congruence studies [106, 107, 151] by revealing previously hidden developer connections. Overall, more research in this direction is required to understand the value of data-flow-based commit interactions.

Bibliography

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. "42 variability bugs in the linux kernel: a qualitative analysis." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2014, pp. 421–432.

[2] Jonathan Aldrich. "Using Types to Enforce Architectural Structure." In: *Proc. Working IEEE / IFIP Conf. Software Architecture (WICSA)*. IEEE, 2008, pp. 211–220.

[3] Sven Amann and Elmar Jürgens. "Change-Driven Testing." In: *The Future of Software Quality Assurance*. Springer-Verlag, 2020, pp. 1–14.

[4] Lars Andersen. "Program Analysis and Specialization for the C Programming Language." PhD thesis. University of Copenhagen, 1994.

[5] Sven Apel. "The New Feature Interaction Challenge." In: *Proc. Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 2017, p. 1.

[6] Sven Apel, Joanne M. Atlee, Luciano Baresi, and Pamela Zave. "Feature Interactions: The Next Generation (Dagstuhl Seminar 14281)." In: *Dagstuhl Reports* 4.7 (2014), pp. 1–24.

[7] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer-Verlag, 2013.

[8] Sven Apel, Christian Kästner, and Eunsuk Kang. "Feature Interactions on Steroids: On the Composition of ML Models." In: *IEEE Softw.* 39.3 (2022), pp. 120–124.

[9] Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. "Exploring feature interactions in the wild: the new feature-interaction challenge." In: *Proc. Int. Workshop on Feature-Oriented Software Development (FOSD)*. ACM, 2013, pp. 1–8.

[10] Sven Apel, Thomas Leich, and Gunter Saake. "Aspectual Feature Modules." In: *IEEE Trans. Softw. Eng.* 34.2 (2008), pp. 162–180.

[11] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. "Strategies for Product-Line Verification: Case Studies and Experiments." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2013, pp. 482–491.

[12] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. "Detection of Feature Interactions Using Feature-Aware Verification." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2011, pp. 372–375.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps." In: *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2014, pp. 259–269.

[14] Mona Attariyan, Michael Chow, and Jason Flinn. "X-Ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software." In: *Proc. Symp. Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2012, pp. 307–320.

[15] Denis Bakhvalov. *Performance Analysis and Tuning on Modern CPUs*. Independently Published, 2020.

[16] Peter Bastian, Markus Blatt, Andreas Dedner, Nils-Arne Dreier, Christian Engwer, René Fritze, Carsten Gräser, Christoph Grüninger, Dominic Kempf, Robert Klöfkorn, Mario Ohlberger, and Oliver Sander. "The Dune Framework: Basic Concepts and Recent Developments." In: *Comput. Math. Appl.* 81 (2021), pp. 75–112.

[17] Cédric Bastoul. "Improving Data Locality in Static Control Programs." PhD thesis. University Paris 6, Pierre et Marie Curie, France, Dec. 2004.

[18] Kent L. Beck and Cynthia Andres. *Extreme Programming Explained - Embrace Change*. The XP series. Addison-Wesley, 2005.

[19] Thorsten Berger, Steven She, Rafael Lotufo, Krzysztof Czarnecki, and Andrzej Wasowski. "Feature-to-Code Mapping in Two Large Product Lines." In: *Proc. Int. Software Product Line Conf. (SPLC)*. Vol. 6287. Springer-Verlag, 2010, pp. 498–499.

[20] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs In the Real World." In: *Commun. ACM* 53.2 (2010), pp. 66–75.

[21] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, Laszlo Lango, and Tibor Gyimóthy. "Code Coverage-Based Regression Test Selection and Prioritization in WebKit." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2012, pp. 46–55.

[22] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. "The Promises and Perils of Mining Git." In: *Proc. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2009, pp. 1–10.

[23] Thomas Bock, Claus Hunsen, Mitchell Joblin, and Sven Apel. "Synchronous Development in Open-Source Projects: A Higher-Level Perspective." In: *Proc. Int. Conf. Automated Software Engineering (ASE)* 29.1 (2022), p. 3.

[24] Thomas Bock, Angelika Schmid, and Sven Apel. "Measuring and Modeling Group Dynamics in Open-Source Software Development: A Tensor Decomposition Approach." In: *ACM Trans. Softw. Eng. Methodol.* 31.2 (2022), 19:1–19:50.

[25] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. "SPL$^{\text{LIFT}}$: Statically Analyzing Software Product Lines in Minutes Instead of Years." In: *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2013, pp. 355–364.

[26] Shawn Bohner and Robert Arnold. *Software Change Impact Analysis*. IEEE, 1996.

[27] Amiangshu Bosu, Fang Liu, Danfeng (Daphne) Yao, and Gang Wang. "Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications." In: *Proc. Symp. Information, Computer and Communications Security (ASIACCS)*. ACM, 2017, pp. 71–85.

[28] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, Johnni Winther, and Paulo Borba. "Intraprocedural Dataflow Analysis for Software Product Lines." In: *LNCS Trans. Aspect Oriented Softw. Dev.* 10 (2013), pp. 73–108.

[29] Rolando Brondolin and Marco D. Santambrogio. "A Black-box Monitoring Approach to Measure Microservices Runtime Performance." In: *ACM Trans. Archit. Code Optim.* 17.4 (2020), 34:1–34:26.

[30] Marcelo Cataldo and James D. Herbsleb. "Coordination Breakdowns and Their Impact on Development Productivity and Software Failures." In: *IEEE Trans. Softw. Eng.* 39.3 (2013), pp. 343–360.

[31] Marcelo Cataldo, James D. Herbsleb, and Kathleen M. Carley. "Socio-Technical Congruence: A Framework For Assessing the Impact of Technical and Work Dependencies on Software Development Productivity." In: *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2008, pp. 2–11.

[32] Marcelo Cataldo, Patrick Wagstrom, James D. Herbsleb, and Kathleen M. Carley. "Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools." In: *Proc. Conf. Computer Supported Cooperative Work (CSCW)*. ACM, 2006, pp. 353–362.

[33] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. "An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2017, pp. 597–608.

[34] Jinfu Chen, Zishuo Ding, Yiming Tang, Mohammed Sayagh, Heng Li, Bram Adams, and Weiyi Shang. "IoPV: On Inconsistent Option Performance Variations." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM. 2023.

[35] Jane Cleland-Huang, Olly Gotel, and Andrea Zisman, eds. *Software and Systems Traceability*. Springer, 2012.

[36] Alistair Cockburn and Jim Highsmith. "Agile Software Development: The People Factor." In: *Computer* 34.11 (2001), pp. 131–133.

[37] Lyra J Colfer and Carliss Y Baldwin. "The Mirroring Hypothesis: Theory, Evidence, and Exceptions." In: *Industrial and Corporate Change* 25.5 (Sept. 2016), pp. 709–738.

[38] Michael Collard and Jonathan Maletic. "srcML 1.0: Explore, Analyze, and Manipulate Source Code." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2016, p. 649.

[39] Melvin E Conway. "How do committees invent?" In: *Datamation* 14.4 (1968), pp. 28–31.

[40] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures." In: *Proc. Int. Symp. Theory of Computing (STOC)*. ACM, 1971, pp. 151–158.

[41] Kevin Crowston and James Howison. "The Social Structure of Free and Open Source Software Development." In: *First Monday* 10.2 (2005).

[42] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. "Core and Periphery in Free/Libre and Open Source Software Team Communications." In: *Proc. Hawaii Int. Conf. Systems Science (HICSS)*. IEEE, 2006.

[43] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[44] Krzysztof Czarnecki and Ulrich W. Eisenecker. "Synthesizing Objects." In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Vol. 1628. Springer-Verlag, 1999, pp. 18–42.

[45] Krzysztof Czarnecki and Krzysztof Pietroszek. "Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints." In: *Proc. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2006, pp. 211–220.

[46] Marco D'Ambros, Harald C. Gall, Michele Lanza, and Martin Pinzger. "Analysing Software Repositories to Understand Software Evolution." In: *Software Evolution*. Springer-Verlag, 2008, pp. 37–67.

[47] Marco D'Ambros, Michele Lanza, and Romain Robbes. "An Extensive Comparison of Bug Prediction Approaches." In: *Proc. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2010, pp. 31–41.

[48] Marco D'Ambros and Romain Robbes. "Effective Mining of Software Repositories." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2011, p. 598.

[49] David Daly. "Creating a Virtuous Cycle in Performance Testing at MongoDB." In: *Proc. Int. Conf. on Performance Engineering (ICPE)*. ACM, 2021, pp. 33–41.

[50] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. "The Use of Change Point Detection to Identify Software Performance Regressions in a Continuous Integration System." In: *Proc. Int. Conf. on Performance Engineering (ICPE)*. ACM, 2020, pp. 67–75.

[51] Arnab De and Deepak D'Souza. "Scalable Flow-Sensitive Pointer Analysis for Java with Strong Updates." In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Vol. 7313. Springer-Verlag, 2012, pp. 665–687.

[52] Daniel Joseph Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. "PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures." In: *Proceedings of the ACM Symp. on Cloud Computing*. ACM, 2014, 8:1–8:13.

[53] N. Deepa, Boopathy Prabadevi, L.B. Krithika, and B. Deepa. "An Analysis on Version Control Systems." In: *Int. Conf. Emerging Trends in Information Technology and Engineering (ic-ETITE)*. 2020, pp. 1–9.

[54] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Jürgens. "Flexible Architecture Conformance Assessment with ConQAT." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 247–250.

[55] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[56] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. "Causality in Configurable Software Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2022, pp. 325–337.

[57] Slawomir Duszynski, Jens Knodel, and Mikael Lindvall. "SAVE: Software Architecture Visualization and Evaluation." In: *Proc. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 323–324.

[58] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. "Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2013, pp. 422–431.

[59] Chris Edwards. "Moore's Law: What Comes Next?" In: *Commun. ACM* 64.2 (2021), pp. 12–14.

[60] Domenik Eichhorn, Tobias Pett, Nils Przigoda, Jessica Kindsvater, Christoph Seidl, and Ina Schaefer. "Coverage-Driven Test Automation for Highly-Configurable Railway Systems." In: *Proc. Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 2023, pp. 23–30.

[61] Pär Emanuelsson and Ulf Nilsson. "A Comparative Study of Industrial Static Analysis Tools." In: *Electron. Notes Theor. Comput. Sci.* 217 (2008), pp. 5–21.

[62] Eric Evans. *Domain-Driven Design – Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[63] Jean-Marie Favre. "Understanding-In-The-Large." In: *Int. Workshop on Program Comprehension (WPC)*. IEEE, 1997, pp. 29–38.

[64] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. "Do Background Colors Improve Program Comprehension in the #ifdef Hell?" In: *Empir. Softw. Eng.* 18.4 (2013), pp. 699–745.

[65] Norman Fenton and Shari Pfleeger. *Software Metrics - A Practical and Rigorous Approach*. International Thomson, 1996.

[66] Gabriel Ferreira, Momin Malik, Christian Kästner, Jürgen Pfeffer, and Sven Apel. "Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel." In: *Proc. Int. Software Product Line Conf. (SPLC)*. ACM, 2016, pp. 65–73.

[67] Andreas Fischer, Benny Fuhry, Jörn Kußmaul, Jonas Janneck, Florian Kerschbaum, and Eric Bodden. "Computation on Encrypted Data Using Dataflow Authentication." In: *ACM Tans. Priv. Secur.* 25.3 (2022), 21:1–21:36.

[68] Stefan Fischer, Lukas Linsbauer, Alexander Egyed, and Roberto Erick Lopez-Herrejon. "Predicting Higher Order Structural Feature Interactions in Variable Systems." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 252–263.

[69] Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities." In: *Int. Computer Software and Applications Conference (COMPSAC)*. IEEE, 2007, pp. 87–96.

[70] Brady J. Garvin and Myra B. Cohen. "Feature Interaction Faults Revisited: An Exploratory Study." In: *Proc. Int. Symp. Software Reliability Engineering (ISSRE)*. IEEE, 2011, pp. 90–99.

[71] Paul Gazzillo and Robert Grimm. "SuperC: Parsing All of C by Taming the Preprocessor." In: *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 323–334.

[72] Daniel M. Germán, Bram Adams, and Kate Stewart. "cregit: Token-Level Blame Information in Git Version Control Repositories." In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2725–2763.

[73] Daniel Germán, Ahmed Hassan, and Gregorio Robles. "Change Impact Graphs: Determining the Impact of Prior Codechanges." In: *Inf. Softw. Technol.* 51 (2009), pp. 1394–1408.

[74] Philipp Gnoyke, Sandro Schulze, and Jacob Krüger. "An Evolutionary Analysis of Software-Architecture Smells." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 413–424.

[75] Christoph Gote, Ingo Scholtes, and Frank Schweitzer. "Analysing Time-Stamped Co-Editing Networks in Software Development Teams using git2net." In: *Empir. Softw. Eng.* 26.4 (2021).

[76] Alexander Grebhahn, Christian Kaltenecker, Christian Engwer, Norbert Siegmund, and Sven Apel. "Lightweight, Semi-Automatic Variability Extraction: A Case Study on Scientific Computing." In: *Empir. Softw. Eng.* 26.2 (2021), p. 23.

[77] Brendan Gregg. *BPF Performance Tools*. Pearson Education, 2019.

[78] Brendan Gregg. *Systems Performance, 2nd Edition*. Pearson, 2020.

[79] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly – Performing Polyhedral Optimizations on a Low-Level Intermediate Representation." In: *Parallel Process. Lett.* 22.4 (2012).

[80] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wasowski. "Variability-Aware Performance Prediction: A Statistical Learning Approach." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[81] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. Chapman and Hall / CRC computational science series. CRC Press, 2011.

[82] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. "Performance Debugging in the Large via Mining Millions of Stack Traces." In: *34th International Conference on Software Engineering*. IEEE, 2012, pp. 145–155.

[83] Xue Han and Tingting Yu. "An Empirical Study on Performance Bugs for Highly Configurable Software Systems." In: *Proc. Int. Symp. Empirical Software Engineering and Measurement (ESEM)*. ACM, 2016, 23:1–23:10.

[84] Xue Han, Tingting Yu, and Michael Pradel. "ConfProf: White-Box Performance Profiling of Configuration Options." In: *Proc. Int. Conf. on Performance Engineering (ICPE)*. ACM, 2021, pp. 1–8.

[85] Quinn Hanam, Ali Mesbah, and Reid Holmes. "Aiding Code Change Understanding with Semantic Change Impact Analysis." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 202–212.

[86] Mark Harman and Peter W. O'Hearn. "From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis." In: *Proc. Int. Conf. Source Code Analysis and Manipulation (SCAM)*. IEEE, 2018, pp. 1–23.

[87] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, Steven Alexander Spoon, and Ashish Gujarathi. "Regression Test Selection for Java Software." In: *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2001, pp. 312–326.

[88] Lile Hattori, Gilson Pereira dos Santos Jr., Fernando Cardoso, and Marcus Sampaio. "Mining Software Repositories for Software Change Impact Analysis: A Case Study." In: *Proc. Brazilian Symp. Databases*. SBC, 2008, pp. 210–223.

[89] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. "Deriving Code Coverage Information from Profiling Data Recorded for a Trace-Based Just-In-Time Compiler." In: *Proc. Int. Conf. Principles and Practices of Programming on the Java Platform (PPPJ)*. ACM, 2013, pp. 1–12.

[90] Jincheng He, Sitao Min, Kelechi Ogudu, Michael Shoga, Alex Polak, Iordanis Fostiropoulos, Barry Boehm, and Pooyan Behnamghader. "The Characteristics and Impact of Uncompilable Code Changes on Software Quality Evolution." In: *Proc. Int. Conf. Software Quality, Reliability and Security (QRS)*. IEEE, 2020, pp. 418–429.

[91] Thomas A. Henzinger and Joseph Sifakis. "The Discipline of Embedded Systems Design." In: *Computer* 40.10 (2007), pp. 32–40.

[92] Thomas Henzinger. "Two Challenges in Embedded Systems Design: Predictability and Robustness." In: *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences* 366 (2008), pp. 3727–36.

[93] James D. Herbsleb. "Socio-Technical Coordination (Keynote)." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2014, p. 1.

[94] James D. Herbsleb. "Building a Socio-Technical Theory of Coordination: Why and How (Outstanding Research Award)." In: *Proc. Int. Symp. Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 2–10.

[95] James D. Herbsleb and Rebecca E. Grinter. "Architectures, Coordination, and Distance: Conway's Law and Beyond." In: *IEEE Softw.* 16.5 (1999), pp. 63–70.

[96] John E. Hopcroft and Robert Endre Tarjan. "Efficient Algorithms for Graph Manipulation [H] (Algorithm 447)." In: *ACM* 16.6 (1973), pp. 372–378.

[97] Claus Hunsen, Janet Siegmund, and Sven Apel. "On the Fulfillment of Coordination Requirements in Open-Source Software Projects: An Exploratory Study." In: *Empir. Softw. Eng.* 25.6 (2020), pp. 4379–4426.

[98] Timea Illes-Seifert and Barbara Paech. "Exploring the Relationship of History Characteristics and Defect Count: An Empirical Study." In: *Proc. ISSTA Workshop on Defects in Large Software Systems*. ACM, 2008, pp. 11–15.

[99] Pooyan Jamshidi and Giuliano Casale. "An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems." In: *Proc. Int. Symp. Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2016, pp. 39–48.

[100] Andrea Janes and Barbara Russo. "Automatic Performance Monitoring and Regression Testing During the Transition from Monolith to Microservices." In: *IEEE International Symp. on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*. IEEE, 2019, pp. 163–168.

[101] Andrejs Jermakovics, Alberto Sillitti, and Giancarlo Succi. "Mining and Visualizing Developer Networks from Version Control Systems." In: *Proc. Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. ACM, 2011, pp. 24–31.

[102] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and Detecting Real-World Performance Bugs." In: *Proc. Conf. Programming Language Design and Implementation (PLDI)*. ACM, 2012, pp. 77–88.

[103] Mitchell Joblin and Sven Apel. "How Do Successful and Failed Projects Differ? A Socio-Technical Analysis." In: *ACM Trans. Softw. Eng. Methodol.* 31.4 (2022), 67:1–67:24.

[104] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. "Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2017, pp. 164–174.

[105] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. "Evolutionary Trends of Developer Coordination: A Network Approach." In: *Empir. Softw. Eng.* 22.4 (2017), pp. 2050–2094.

[106] Mitchell Joblin, Barbara Eckl-Ganser, Thomas Bock, Angelika Schmid, Janet Siegmund, and Sven Apel. "Hierarchical and Hybrid Organizational Structures in Open-Source Software Projects: A Longitudinal Study." In: *ACM Trans. Softw. Eng. Methodol.* (2022).

[107] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. "From Developer Networks to Verified Communities: A Fine-Grained Approach." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 563–573.

[108] Frederick P. Brooks Jr. *The Mythical Man-Month - Essays on Software Engineering (2. ed.)* Addison-Wesley, 1995.

[109] Huzefa H. Kagdi and Jonathan I. Maletic. "Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction." In: *Proc. Working Conf. Mining Software Repositories (MSR)*. IEEE, 2007, p. 17.

[110] Huzefa Kagdi, Michael Collard, and Jonathan Maletic. "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution." In: *J. Softw. Maint. and Evol.: Research and Practice* 19.2 (2007), pp. 77–131.

[111] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael Collard. "Blending Conceptual and Evolutionary Couplings to Support Change Impact Analysis in Source Code." In: *Proc. Working Conf. Reverse Engineering (WCRE)*. IEEE, 2010, pp. 119–128.

[112] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "The Interplay of Sampling and Machine Learning for Software Performance Prediction." In: *IEEE Softw.* 37.4 (2020), pp. 58–66.

[113] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2019, pp. 1084–1094.

[114] Christian Kaltenecker, Stefan Mühlbauer, Alexander Grebhahn, Norbert Siegmund, and Sven Apel. "Performance Evolution of Configurable Software Systems: An Empirical Study." In: *Empir. Softw. Eng.* 1 (2023).

[115] Teemu Kanstrén. "Towards a Deeper Understanding of Test Coverage." In: *J. Softw. Maint. and Evol.: Research and Practice* 20.1 (2008), pp. 59–76.

[116] Vini Kanvar and Uday P. Khedker. "Heap Abstractions for Static Analysis." In: *ACM Comput. Surv.* 49.2 (2016), 29:1–29:47.

[117] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. "Type Checking Annotation-Based Product Lines." In: *ACM Trans. Softw. Eng. Methodol.* 21.3 (2012), 14:1–14:39.

[118] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation." In: *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2011, pp. 805–824.

[119] Yogita Khatri and Sandeep Singh. "Cross Project Defect Prediction: A Comprehensive Survey With its SWOT Analysis." In: *Innovations in Systems and Software Engineering* (2021). Online First.

[120] Hassan Khosravi and Recep Colak. "Exploratory Analysis of Co-Change Graphs for Code Refactoring." In: *Proc. Int. Conf. Advances in Artificial Intelligence (AI)*. Springer-Verlag, 2009, pp. 219–223.

[121] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming." In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Vol. 1241. Springer-Verlag, 1997, pp. 220–242.

[122] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. "Implicit Flows: Can't Live with 'Em, Can't Live without 'Em." In: *Proc. Int. Conf. on Information Systems Security (ICISS)*. Vol. 5352. Lecture Notes in Computer Science. Springer-Verlag, 2008, pp. 56–70.

[123] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. "Static Evaluation of Software Architectures." In: *Proc. Conf. Software Maintenance and Reengineering (CSMR)*. IEEE, 2006, pp. 279–294.

[124] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Sven Apel. "On the Relation of Control-Flow and Performance Feature Interactions: A Case Study." In: *Empir. Softw. Eng.* 24.4 (2019), pp. 2410–2437.

[125] Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, Alexander Grebhahn, and Sven Apel. "Tradeoffs in Modeling Performance of Highly Configurable Software Systems." In: *Softw. Syst. Model.* 18.3 (2019), pp. 2265–2283.

[126] Samuel Kounev, Klaus-Dieter Lange, and Jóakim von Kistowski. *Systems Benchmarking - For Scientists and Engineers*. Springer, 2020.

[127] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. "TEEMon: A Continuous Performance Monitoring Framework for TEEs." In: *Proc. Int. Middleware Conf. (MIDDLEWARE)*. ACM, 2020, pp. 178–192.

[128] Robert E. Kraut and Lynn A. Streeter. "Coordination in Software Development." In: *Commun. ACM* 38.3 (1995), pp. 69–81.

[129] Maria Kretsou, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Ignatios S. Deligiannis, and Vassilis C. Gerogiannis. "Change Impact Analysis: A Systematic Mapping Study." In: *J. Syst. Softw.* 174 (2021), p. 110892.

[130] Rahul Krishna, Vivek Nair, Pooyan Jamshidi, and Tim Menzies. "Whence to Learn? Transferring Knowledge in Configurable Systems Using BEETLE." In: *IEEE Trans. Softw. Eng.* 47.12 (2021), pp. 2956–2972.

[131] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. "Can Method Data Dependencies Support the Assessment of Traceability between Requirements and Source Code?" In: *J. Softw. Evol. Process* 27.11 (2015), pp. 838–866.

[132] Irwin Kwan, Marcelo Cataldo, and Daniela E. Damian. "Conway's Law Revisited: The Evidence for a Task-Based Perspective." In: *IEEE Softw.* 29.1 (2012), pp. 90–93.

[133] Irwin Kwan, Adrian Schröter, and Daniela E. Damian. "Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project." In: *IEEE Trans. Softw. Eng.* 37.3 (2011), pp. 307–324.

[134] Chris Lattner and Vikram S. Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proc. Int. Symp. Code Generation and Optimization (CGO)*. IEEE, 2004, pp. 75–88.

[135] Duc Minh Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. "An Empirical Study of Architectural Decay in Open-Source Software." In: *Proc. Int. Conf. Software Architecture (ICSA)*. IEEE, 2018, pp. 176–185.

[136] Duc Le, Eric Walkingshaw, and Martin Erwig. "#ifdef Confirmed Harmful: Promoting Understandable Software Variation." In: *Proc. Int. Symp. Visual Languages and Human-Centric Computing (VLHCC)*. IEEE, 2011, pp. 143–150.

[137] Steffen Lehnert. *A Review of Software Change Impact Analysis*. Citeseer, 2011.

[138] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. "There's Plenty of Room at the Top: What Will Drive Computer Performance after Moore's Law?" In: *Science* 368.6495 (2020), eaam9744.

[139] Olaf Leßenich, Sven Apel, Christian Kästner, Georg Seibt, and Janet Siegmund. "Renaming and Shifted Code in Structured Merging: Looking Ahead for Precision and Performance." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2017, pp. 543–553.

[140] Olaf Leßenich, Sven Apel, and Christian Lengauer. "Balancing Precision and Performance in Structured Merge." In: *Proc. Int. Conf. Automated Software Engineering (ASE)* 22.3 (2015), pp. 367–397.

[141] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. "A Survey of Code-Based Change Impact Analysis Techniques." In: *Softw. Test. Verification Reliab.* 23.8 (2013), pp. 613–646.

[142] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick D. McDaniel. "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2015, pp. 280–291.

[143] Xiaoli Lian, Ahmed Fakhry, Li Zhang, and Jane Cleland-Huang. "Leveraging Traceability to Reveal the Tapestry of Quality Concerns in Source Code." In: *Int. Symp. Software and Systems Traceability (SST)*. IEEE, 2015, pp. 50–56.

[144] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2010, pp. 105–114.

[145] Jörg Liebig, Andreas Janker, Florian Garbe, Sven Apel, and Christian Lengauer. "Morpheus: Variability-Aware Refactoring in the Wild." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 380–391.

[146] Jörg Liebig, Christian Kästner, and Sven Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code." In: *Proc. Int. Conf. Aspect-Oriented Software Development (AOSD)*. ACM, 2011, pp. 191–202.

[147] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking Load-Time Configuration Options." In: *IEEE Trans. Softw. Eng.* 44.12 (2018), pp. 1269–1291.

[148] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Amaral, Bor-Yuh Chang, Samuel Guyer, Uday Khedker, Anders Møller, and Dimitrios Vardoulakis. "In Defense of Soundiness: A Manifesto." In: *Commun. ACM* 58.2 (2015), pp. 44–46.

[149] Patrick Mäder and Alexander Egyed. "Do Developers Benefit from Requirements Traceability when Evolving and Maintaining a Software System?" In: *Empir. Softw. Eng.* 20.2 (2015), pp. 413–441.

[150] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, 2003.

[151] Wolfgang Mauerer, Mitchell Joblin, Damian A. Tamburri, Carlos V. Paradis, Rick Kazman, and Sven Apel. "In Search of Socio-Technical Congruence: A Large-Scale Longitudinal Study." In: *IEEE Trans. Softw. Eng.* 48.8 (2022), pp. 3159–3184.

[152] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. "A Comparison of 10 Sampling Algorithms for Configurable Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2016, pp. 643–654.

[153] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. "On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. ACM, 2016, pp. 483–494.

[154] Andrew Meneely and Laurie Williams. "Socio-Technical Developer Networks: Should we Trust our Measurements?" In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2011, pp. 281–290.

[155] Tom Mens and Serge Demeyer, eds. *Software Evolution*. Springer-Verlag, 2008.

[156] Tim Menzies and Thomas Zimmermann. "Software Analytics: So What?" In: *IEEE Softw.* 30 (2013), pp. 31–37.

[157] Mariam El Mezouar, Feng Zhang, and Ying Zou. "An empirical study on the teams structures in social coding using GitHub projects." In: *Empir. Softw. Eng.* 24.6 (2019), pp. 3790–3823.

[158] Mehdi Mirakhorli and Jane Cleland-Huang. "Using Tactic Traceability Information Models to Reduce the Risk of Architectural Degradation during System Maintenance." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2011, pp. 123–132.

[159] Mehdi Mirakhorli, Yonghee Shin, Jane Cleland-Huang, and Murat Çinar. "A Tactic-Centric Approach for Automating Traceability of Quality Concerns." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2012, pp. 639–649.

[160] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, and Qiong Feng. "Architecture Anti-Patterns: Automatically Detectable Violations of Design Principles." In: *IEEE Trans. Softw. Eng.* 47.5 (2021), pp. 1008–1028.

[161] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two Case studies of Open-Source Software Development: Apache and Mozilla." In: *ACM Trans. Softw. Eng. Methodol.* 11.3 (2002), pp. 309–346.

[162] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University, https://cs.au.dk/~amoeller/spa/spa.pdf. 2023.

[163] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Accurate Modeling of Performance Histories for Evolving Software Systems." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2019, pp. 640–652.

[164] Stefan Mühlbauer, Sven Apel, and Norbert Siegmund. "Identifying Software Performance Changes Across Variants and Versions." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2020, pp. 611–622.

[165] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. "Explaining Static Analysis - A Perspective." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2019, pp. 29–32.

[166] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. "A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools." In: *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*. ACM, 2022, pp. 532–543.

[167] Stefan Nagy and Matthew Hicks. "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing." In: *Int. Symp. Security and Privacy (SP)*. IEEE, 2019, pp. 787–802.

[168] Reena Nair and Tony Field. "GAPP: A Fast Profiler for Detecting Serialization Bottlenecks in Parallel Linux Applications." In: *Proc. Int. Conf. on Performance Engineering (ICPE)*. ACM, 2020, pp. 257–264.

[169] D. Nesteruk. *Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design*. Apress, 2018.

[170] Christian Newman, Jonathan Maletic, and Michael Collard. "srcType: A Tool for Efficient Static Type Resolution." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2016, pp. 604–606.

[171] Christian Newman, Tessandra Sage, Michael Collard, Hakam Alomari, and Jonathan Maletic. "srcSlice: A Tool for Efficient Static Forward Slicing." In: *Companium Volume ICSE*. ACM, 2016, pp. 621–624.

[172] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[173] Andrzej Nowak and Georgios Bitzes. *The Overhead of Profiling using PMU Hardware Counters*. 2014.

[174] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. "Finding Near-Optimal Configurations in Product Lines by Random Sampling." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 61–71.

[175] Marco Ortu, Tracy Hall, Michele Marchesi, Roberto Tonelli, David Bowes, and Giuseppe Destefanis. "Mining Communication Patterns in Software Development: A GitHub Analysis." In: *Proc. Int. Conf. Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 2018, pp. 70–79.

[176] Adrian Ostrowski and Piotr Gaczkowski. *Software Architecture with C++: Design Modern Systems using Effective Architecture Concepts, Design patterns, and Techniques with C++20*. Packt Publishing, 2021.

[177] Sebastiano Panichella, Gabriele Bavota, Massimiliano Di Penta, Gerardo Canfora, and Giuliano Antoniol. "How Developers' Collaborations Identified from Different Sources Tell Us about Code Changes." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 251–260.

[178] Sebastiano Panichella, Gerardo Canfora, Massimiliano Di Penta, and Rocco Oliveto. "How the Evolution of Emerging Collaborations Relates to Code Changes: An Empirical Study." In: *Proc. Int. Conf. Program Comprehension (ICPC)*. ACM, 2014, pp. 177–188.

[179] Austin Parker, Daniel Spoonhower, Jonathan Mace, and Rebecca Isaacs. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated, 2020.

[180] David L. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules." In: *Commun. ACM* 15.12 (1972), pp. 1053–1058.

[181] David L. Parnas. "On the Design and Development of Program Families." In: *IEEE Trans. Softw. Eng.* 2.1 (1976), pp. 1–9.

[182] Leonardo Teixeira Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wasowski, Christian Kästner, and Jianmei Guo. "Feature-Oriented Software Evolution." In: *Proc. Int. Workshop Variability Modelling of Software-intensive Systems (VaMoS)*. ACM, 2013, 17:1–17:8.

[183] Leonardo Teixeira Passos, Leopoldo Teixeira, Nicolas Dintzner, Sven Apel, Andrzej Wasowski, Krzysztof Czarnecki, Paulo Borba, and Jianmei Guo. "Coevolution of Variability Models and Related Software Artifacts - A Fresh Look at Evolution Patterns in the Linux Kernel." In: *Empir. Softw. Eng.* 21.4 (2016), pp. 1744–1793.

[184] Benjamin C. Pierce. *Types and programming languages*. MIT Press, 2002.

[185] Reese T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams." In: *Papers presented at the 1959 eastern joint IRE-AIEE-ACM computer conference*. ACM, 1959, pp. 133–138.

[186] Mona Rahimi and Jane Cleland-Huang. "Evolving Software Trace Links between Requirements and Source Code." In: *Empir. Softw. Eng.* 23.4 (2018), pp. 2198–2231.

[187] Michael Rath, David Lo, and Patrick Mäder. "Analyzing Requirements and Traceability Information to Improve Bug Localization." In: *Proc. Working Conf. Mining Software Repositories (MSR)*. ACM, 2018, pp. 442–453.

[188] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. "VUzzer: Application-aware Evolutionary Fuzzing." In: *Proc. Int. Symp. Network and Distributed System Security (NDSS)*. The Internet Society, 2017.

[189] Sanjai Rayadurgam and Mats Per Erik Heimdahl. "Coverage Based Test-Case Generation Using Model Checkers." In: *Proc. Int. Conf. Engineering of Computer-Based Systems (ECBS)*. IEEE, 2001, p. 83.

[190] Chris Reade. *Elements of Functional Programming*. International computer science series. Addison-Wesley, 1989.

[191] James Reinders. *Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[192] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability." In: *Proc. Symp. Principles of Programming Languages (POPL)*. ACM Press, 1995, pp. 49–61.

[193] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. "Presence-Condition Simplification in Highly Configurable Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2015, pp. 178–188.

[194] Alexander von Rhein, Jörg Liebig, Andreas Janker, Christian Kästner, and Sven Apel. "Variability-Aware Static Analysis at Scale: An Empirical Study." In: *ACM Trans. Softw. Eng. Methodol.* 27.4 (2018), 18:1–18:33.

[195] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. "Variability Encoding: From Compile-Time to Load-Time Variability." In: *J. Log. Algebraic Methods Program.* 85.1 (2016), pp. 125–145.

[196] H. Gordon Rice. "Classes of Recursively Enumerable Sets and their Decision Problems." In: *Trans. of the American Mathematical Society* 74 (1953), pp. 358–366.

[197] Arch D. Robison. "Intel® Threading Building Blocks (TBB)." In: *Encyclopedia of Parallel Computing*. Springer-Verlag, 2011, pp. 955–964.

[198] Marko Rosenmüller, Norbert Siegmund, Sven Apel, and Gunter Saake. "Flexible Feature Binding in Software Product Lines." In: *Autom. Softw. Eng.* 18.2 (2011), pp. 163–197.

[199] Gregg Rothermel and Mary Jean Harrold. "A Safe, Efficient Regression Test Selection Technique." In: *ACM Trans. Softw. Eng. Methodol.* 6.2 (1997), pp. 173–210.

[200] W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 1987, pp. 328–339.

[201] Shmuel Sagiv, Thomas Reps, and Susan Horwitz. "Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation." In: *Theor. Comput. Sci.* 167.1&2 (1996), pp. 131–170.

[202] Atrisha Sarkar, Jianmei Guo, Norbert Siegmund, Sven Apel, and Krzysztof Czarnecki. "Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)." In: *Proc. Int. Conf. Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[203] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Trans. Softw. Eng. Methodol.* (2023).

[204] Florian Sattler, Alexander von Rhein, Thorsten Berger, Niklas Schalck Johansson, Mikael Mark Hardø, and Sven Apel. "Lifting Inter-App Data-Flow Analysis to Large App Sets." In: *Autom. Softw. Eng.* 25.2 (2018), pp. 315–346.

[205] Fabian Schiebel. "Evaluating Techniques for Flow-, Context-, and Field-Sensitive Dataflow Analysis in C/C++." MA thesis. Paderborn University, 2021.

[206] Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Dominik Werle, Andreas Reiter, Michael Selzer, Anne Koziolek, and Torsten Hoefler. "Performance-Detective: Automatic Deduction of Cheap and Accurate Performance Models." In: *Int. Conf. Supercomputing (ICS)*. ACM, 2022, 3:1–3:13.

[207] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis." In: *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Vol. 194. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 2:1–2:31.

[208] Philipp Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Interprocedural Static Analysis Framework for C/C++." In: *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer-Verlag, 2019, pp. 393–410.

[209] Georg Seibt, Florian Heck, Guilherme Cavalcanti, Paulo Borba, and Sven Apel. "Leveraging Structure in Software Merge: An Empirical Study." In: *IEEE Trans. Softw. Eng.* 48.11 (2022), pp. 4590–4610.

[210] Micha Sharir and Amir Pnueli. *Two Approaches to Interprocedural Data Flow Analysis*. Department of Computer Science, New York University, 1978.

[211] Pankajeshwara N. Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. "Boundary Spanners in Open Source Software Development: A Study of Python Email Archives." In: *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 2017, pp. 308–317.

[212] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. "Performance-Influence Models for Highly Configurable Systems." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.

[213] Norbert Siegmund, Sergiy S. Kolesnikov, Christian Kästner, Sven Apel, Don S. Batory, Marko Rosenmüller, and Gunter Saake. "Predicting Performance via Automated Feature-Interaction Detection." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2012, pp. 167–177.

[214] Norbert Siegmund, Alexander von Rhein, and Sven Apel. "Family-Based Performance Measurement." In: *Proc. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2013, pp. 95–104.

[215] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. "Scalable Prediction of Non-functional Properties in Software Product Lines." In: *Proc. Int. Software Product Line Conf. (SPLC)*. IEEE, 2011, pp. 160–169.

[216] Norbert Siegmund, Marko Rosenmüller, Christian Kästner, Paolo G. Giarrusso, Sven Apel, and Sergiy S. Kolesnikov. "Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption." In: *Inf. Softw. Technol.* 55.3 (2013), pp. 491–507.

[217] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. "Pick Your Contexts Well: Understanding Object-Sensitivity." In: *Proc. Symp. Principles of Programming Languages (POPL)*. ACM, 2011, pp. 17–30.

[218] Linhai Song and Shan Lu. "Statistical Debugging for Real-World Performance Problems." In: *Proc. Int. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, 2014, pp. 561–578.

[219] Linhai Song and Shan Lu. "Performance Diagnosis for Inefficient Loops." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2017, pp. 370–380.

[220] Johannes Späth. "Synchronized Pushdown Systems for Pointer and Data-Flow Analysis." PhD thesis. University of Paderborn, Germany, 2019.

[221] Johannes Späth, Karim Ali, and Eric Bodden. "Context-, Flow-, and Field-Sensitive Data-Flow Analysis using Synchronized Pushdown Systems." In: *ACM Prog. Lang.* 3.Proc. Symp. Principles of Programming Languages (POPL) (2019), 48:1–48:29.

[222] Henry Spencer and Geoff Collyer. "#ifdef Considered Harmful, or Portability Experience with C News." In: *Proc. USENIX Technical Conf.* USENIX Association, 1992.

[223] Margaret-Anne Storey, Neil Ernst, Courtney Williams, and Eirini Kalliamvakou. "The Who, What, How of Software Engineering Research: A Socio-Technical Framework." In: *Empir. Softw. Eng.* 25 (2020), pp. 4097–4129.

[224] Diane E. Strode, Sid L. Huff, Beverley G. Hope, and Sebastian Link. "Coordination in Co-located Agile Software Development Projects." In: *J. Syst. Softw.* 85.6 (2012), pp. 1222–1238.

[225] Nassim Nicholas Taleb. *The Black Swan: The Impact of the Highly Improbable*. Random House Group, 2007.

[226] Damian A. Tamburri, Patricia Lago, and Hans van Vliet. "Organizational Social Structures for Software Engineering." In: *ACM Comput. Surv.* 46.1 (2013), 3:1–3:35.

[227] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. "*N* Degrees of Separation: Multi-Dimensional Separation of Concerns." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 1999, pp. 107–119.

[228] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10, 000 Feature Problem." In: *Proc. Europ. Conf. Computer Systems (EuroSys)*. ACM, 2011, pp. 47–60.

[229] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software Architecture - Foundations, Theory, and Practice*. Wiley, 2010.

[230] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. "Safe Composition of Product Lines." In: *Proc. Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2007, pp. 95–104.

[231] Dominik Thönnes and Ulrich Rüde. "Model-Based Performance Analysis of the HyTeG Finite Element Framework." In: *Proc. of the Platform for Advanced Scientific Computing Conference (PASC)*. ACM, 2023, 23:1–23:12.

[232] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A Classification and Survey of Analysis Strategies for Software Product Lines." In: *ACM Comput. Surv.* 47.1 (2014), 6:1–6:45.

[233] Thomas Thüm, André van Hoorn, Sven Apel, Johannes Bürdek, Sinem Getir, Robert Heinrich, Reiner Jung, Matthias Kowal, Malte Lochau, Ina Schaefer, and Jürgen Walter. "Performance Analysis Strategies for Software Variants and Versions." In: *Managed Software Evolution*. Springer-Verlag, 2019, pp. 175–206.

[234] John Toman and Dan Grossman. "Taming the Static Analysis Beast." In: *Summit on Advances in Programming Languages (SNAPL)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, 18:1–18:14.

[235] Adam Tornhill. *Software Design X-Rays*. Pragmatic Bookshelf, 2018.

[236] Jason Tsay, Laura Dabbish, and James D. Herbsleb. "Influence of Social and Technical Factors for Evaluating Contribution in GitHub." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2014, pp. 356–366.

[237] Alan M. Turing. "On Computable Numbers, With an Application to the Entscheidungsproblem." In: *Proc. London Mathematical Society* s2-42.1 (1937), pp. 230–265.

[238] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates: The Complete Guide*. 2nd. Addison-Wesley, 2017.

[239] Alberto Varela, Héctor Pérez-González, Francisco Martínez-Pérez, and Carlos Soubervielle-Montalvo. "Source Code Metrics: A Systematic Mapping Study." In: *J. Syst. Softw.* 128 (2017), pp. 164–197.

[240] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "ConfigCrusher: Towards White-Box Performance Analysis for Configurable Systems." In: *Autom. Softw. Eng.* 27.3 (2020), pp. 265–300.

[241] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-Box Analysis over Machine Learning: Modeling Performance of Configurable Systems." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2021, pp. 1072–1084.

[242] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "On Debugging the Performance of Configurable Software Systems: Developer Needs and Tailored Tool Support." In: *Proc. Int. Conf. Software Engineering (ICSE)*. ACM, 2022, pp. 1571–1583.

[243] Ashwin Prasad Shivarpatna Venkatesh, Jiawei Wang, Li Li, and Eric Bodden. "Enhancing Comprehension and Navigation in Jupyter Notebooks with Static Analysis." In: *Proc. Int. Conf. Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 391–401.

[244] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool." In: Jan. 2012.

[245] Markos Viggiato, Johnatan Oliveira, Eduardo Figueiredo, Pooyan Jamshidi, and Christian Kästner. "How Do Code Changes Evolve in Different Platforms? A Mining-Based Investigation." In: *Proc. Int. Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 218–222.

[246] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. "Variational Data Structures: Exploring Tradeoffs in Computing with Variability." In: *Proc. Int. Symp. New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)* ACM, 2014, pp. 213–226.

[247] Max Weber, Sven Apel, and Norbert Siegmund. "White-Box Performance-Influence Models: A Profiling and Learning Approach." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE, 2021, pp. 1059–1071.

[248] Lingmei Weng, Yigong Hu, Peng Huang, Jason Nieh, and Junfeng Yang. "Effective Performance Issue Diagnosis with Value-Assisted Cost Profiling." In: *Proc. Europ. Conf. Computer Systems (EuroSys)*. ACM, 2023, pp. 1–17.

[249] Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly, 2020.

[250] Niklaus Wirth. "Program Development by Stepwise Refinement." In: *Commun. ACM* 14.4 (1971), pp. 221–227.

[251] Guowu Xie, Jianbo Chen, and Iulian Neamtiu. "Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software." In: *Proc. Int. Conf. Software Maintenance (ICSM)*. IEEE, 2009, pp. 51–60.

[252] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. "Hey, You Have Given Me Too Many Knobs!: Understanding and Dealing With Over-Designed Configuration in System Software." In: *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 307–319.

[253] Hong Zhu, Patrick A. V. Hall, and John H. R. May. "Software Unit Test Coverage and Adequacy." In: *ACM Comput. Surv.* 29.4 (1997), pp. 366–427.

[254] Thomas Zimmermann and Nachiappan Nagappan. "Predicting Defects Using Network Analysis on Dependency Graphs." In: *Proc. Int. Conf. Software Engineering (ICSE)*. IEEE / ACM, 2008, pp. 531–540.

[255] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. "Mining Version Histories to Guide Software Changes." In: *IEEE Trans. Softw. Eng.* 31.6 (2005), pp. 429–445.

[256] Vlas Zyrianov, Christian Newman, Drew Guarnera, Michael Collard, and Jonathan Maletic. "srcPtr: A Framework for Implementing Static Pointer Analysis Approaches." In: *Proc. Int. Conf. Program Comprehension (ICPC)*. IEEE / ACM, 2019, pp. 144–147.