Scaling Static Whole-Program Analysis to Modern C and C++ Software Development

Statically Analyzing C and C++ Software With PhASAR

Philipp Dominik Schubert

Doctoral Thesis

Submitted in partial fulfillment of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisors
Prof. Dr. Eric Bodden
JProf. Dr. Ben Hermann

Paderborn University
Faculty of Computer Science, Electrical Engineering and Mathematics
Bielefeld & Paderborn, July 24, 2024

Abstract

While static analysis originates in optimizing compilers, it is nowadays also heavily used to automatically detect bugs and security breaches.

Static analyses that aim at detecting bugs and security breaches have to be precise and inter-procedural, i.e., must span the whole program to compute results that are actually useful to developers. Analyses run in a whole-program manner, however, oftentimes lead to unsatisfactory performance. Unfortunately, traditional whole-program analysis also does not match modern software development that is characterized by extensive library usage and continuous integration/continuous deployment, which compromises performance to the point at which it becomes practically infeasible. Analyzing C and C++ programs complicates matters further, since these languages are notoriously hard to analyze statically. Still, they are the de facto standard for embedded systems and various performance, safety and security critical domains, which makes them a desirable target to analyze.

To address the challenging problem of making precise static whole-program analysis scale, we exploit the fact that virtually all serious software projects are organized with help of version control systems such as Git, Mercurial or Apache Subversion. Starting from a "blank" repository that contains only the target source code, we enrich the repository with additional, persisted static analysis information to make whole-program analysis actually feasible in practice.

The first problem that needs to be addressed when analyzing C and C++ software is that every non-trivial program, due to the preprocessor, is written in a mixed language and represents a software product line that—in the worst case—comprises exponentially many software products—analyzing them one by one thus cannot scale. We therefore present VarAlyzer, a variability-aware static analysis approach that analyzes software product lines as a whole.

To avoid unnecessary repeated reanalysis of library components that do not change from one analysis run to another, we introduce ModAlyzer, a compositional analysis approach that analyzes and summarizes library components. ModAlyzer persists analysis summaries such they can be checked into the target project's code repository, allowing anyone to check out a specific commit together with a library's up-to-date analysis summaries. These summaries can then be employed while analyzing the actual application code to significantly reduce analysis times.

To further avoid unnecessary reanalysis for parts of the code that do undergo frequent changes, but whose static analysis results computed in a previous analysis run are still valid, we present IncAlyzer. IncAlyzer exploits commit information provided by the target project's version control system to recompute analysis information only for the parts of the code that actually changed. Since individual commits typically only introduce small and local code changes, large portions of previously computed analysis results are still

valid for a new commit, making this approach very effective. IncAlyzer, too, persists analysis information through ModAlyzer, allowing one to keep the project's code base in synchronization with up-to-date static analysis information. This reduces analysis times—even for deep, semantic analysis—to a minium.

We have implemented our approaches within PhASAR, our novel static analysis framework that aims at making the analysis of real-world C and C++ programs feasible in practice by matching precise whole-program analysis to modern software development. We show that PhASAR—the host of VarAlyzer, ModAlyzer, and IncAlyzer—not only looks good on paper, but is actually useful and helps solving real problems by discussing its applications in interesting projects from academia and its application in a large industry project with a leading telecommunications company that shows how our infrastructure can solve real problems such as software debloating, program comprehension and bug finding that could not be addressed before.

Zusammenfassung

Statische Programmanalyse hat seine Ursprünge in optimierenden Compilern, wird aber heutzutage auch immer mehr zum automatisierten Auffinden von Bugs und Sicherheitslücken verwendet.

Statische Analysen mit dem Ziel Bugs und Sicherheitslücken zu finden, müssen präzise und inter-prozedural, d.h. das gesamte Programm umfassend, sein, um Ergebnisse berechnen zu können, die Entwickler:innen tatsächlich helfen. Analysen, die das gesamte Programm überspannen, führen jedoch leider oftmals zu unzureichender Performanz. Traditionelle inter-prozedurale Analysen passen leider auch nicht mit den modernen Arbeitsabläufen in der Softwareentwicklung zusammen, welche insbesondere durch die häufige Verwendung von Bibliotheken und den Einsatz von continuous integration/continuous deployment Ansätzen gekennzeichnet sind. Hierdurch wird die Performanz soweit reduziert, dass solche Whole-Program-Analysen praktisch undurchführbar werden. Die Analyse von C und C++ Programmen verkompliziert die Sache weiter, da diese Sprachen dafür bekannt sind statisch schwierig analysierbar zu sein. Dennoch sind diese Sprachen der de facto Standard für eingebettete Systeme und die verschiedensten Performanz und sicherheitskritischen Domänen, was sie zu attraktiven Analysezielen macht.

Um das herausfordernde Problem der Skalierbarkeit von präziser, statischer Whole-Program-Analyse zu adressieren, machen wir uns den Umstand zu nutze, dass nahezu jedes ernsthafte Softwareprojekt mit Hilfe eines Systems zur Versionsverwaltung wie etwa Git, Mercurial oder Apache Subversion organisiert wird. Ausgehend von einem “rohen” Repository, welches nur den zu analysierenden Source Code enthält, reichern wir dieses mit zusätzlichen persistierten statischen Analyseinformation an, um Whole-Program-Analysen in der Praxis doch zu ermöglichen.

Das erste Problem, welches bei der Analyse von C und C++ Software gelöst werden muss, ist, dass jedes nicht-triviale Programm auf Grund des Präprozessors in einer gemischten Sprache geschrieben ist und dadurch eine Software-Produktlinie darstellt. Im schlimmsten Fall enthält eine Software-Produktlinie exponentiell viele Softwareprodukte. Diese einzeln, eine nach der anderen zu analysieren, kann nicht skalieren. Aus diesem Grund präsentieren wir VarAlyzer, einen varabilitäts-bewussten statischen Analyseansatz, der Software-Produktlinien stattdessen als Ganzes analysiert.

Um die unnötige wiederholte Analyse von Bibliothekskomponenten, die sich von einem Analyselauf zum nächsten nicht ändern, zu vermeiden, führen wir ModAlyzer ein. ModAlyzer ist ein kompositionaler Analyseansatz, der Bibliothekskomponenten analysiert und zusammenfasst. ModAlyzer persistiert die Analysezusammenfassung, sodass diese in das Code Repository des Zielprojekts eingecheckt werden können. Dies erlaubt es einen spezifischen Commit des Projekts mit den dazugehörigen, aktuellen Analysezusammenfassung

auszuchecken. Die Analysezusammenfassungen können dann eingesetzt werden, wenn die eigentliche Anwendung analysiert wird, um die Analysezeiten siginifikant zu reduzieren.

Um weitere unnötige, wiederholte Analysen für Programmteile zu vermeiden, die häufigen Änderungen unterliegen, aber dessen statische Analyseinformationen, die in einem vorherigen Analyselauf berechnet wurden, immer noch valide sind, stellen wir IncAlyzer vor. IncAlyzer nutzt die Informationen zu den Commits eines Projekts aus, die durch die Versions-Kontrollsysteme der Zielprojekte bereit gestellt werden. Mit diesen Informationen werden Analysen lediglich für die Programmteile erneut berechnet, die sich tatsächlich geändert haben. Da einzelne Commits typischerweise nur kleine und lokale Änderungen beinhalten, sind große Teile von zuvor berechneten Analyseinformationen immer noch gültig, was diesen Ansatz sehr effektiv macht. IncAlyzer persistiert die Analyseinformationen ebenfalls mit Hilfe von ModAlyzer und erlaubt damit die aktuellen Analyseinformationenen mit der Codebasis des Projekts zu synchronisieren. Dies reduziert die Laufzeiten auch für tiefe, semantische Programmanalysen auf ein Minimum.

Wir haben unsere Analyseansätze in PhASAR implementiert. PhASAR ist unser neues statisches Analyserahmenwerk, welches auf die Analyse von real-welt C und C++ Programmen abzielt und dies ermöglicht, indem es präzise Gesammtprogrammanalysen auf moderne Softwareentwicklung abstimmt. Wir zeigen, dass PhASAR—der Host von VaRAlyzer, ModAlyzer und IncAlyzer—nicht nur auf dem Papier gut assieht, sondern einen tatsächlichen Nutzen aufweist und echte Probleme lösen kann. Dazu diskutieren wir PhASAR's Anwendungen in zwei interessanten Projekten aus dem akademischen Umfeld und einem großen Industrieprojekt mit einem führenden Telekommunikationskonzern. Dies zeigt, dass unsere Infrastruktur echte Probleme wie etwa Software Debloating, Programmverstehen und das Auffinden von komplexen Bugs und Sicherheitslücken lösen kann, die zuvor nicht adressiert werden konnten.

Acknowledgements

First, I would like to thank my beloved wife Irina, and my two daughters Amalia and Nina for always supporting me. You have been my greatest motivators; without you, this document would not exist. You always believe in my abilities even during times in which I did not.

I would also like to express immense gratitude to my two advisors Eric Bodden and Ben Hermann. Without them and their tremendous support, all of this would not have been possible. Eric gave me all the academic freedom that one could have ever asked for. He was always approachable and provided me with invaluable feedback on research ideas and papers. Eric's high standards allowed me to make a huge step forward in my professional development. Similarly, I would like to thank Ben for our numerous discussions and his helpful input on the various topics one needs to deal with as a researcher.

I, of course, also thank my colleagues from Paderborn University and Fraunhofer IEM. In particular, I would like to mention my former colleagues with whom I shared an office for several years: Johannes Geismann and Martin Mory, and for a brief period of time Richard Leer. You were always open to discussing research problems and also listened when I shared frustration about rejected papers and other academic endeavors that did not go as planned. I think we had great fun and great chemistry. Then, I would like to thank Stefan Krüger for patiently enduring my sometimes interesting humor and for all of our satirical discussions on all kinds of matters. Huge kudos also to Lisa Nguyen Quang Do who really helped to set me up for professional writing in academia.

Many thanks to Vera Meyer, Jürgen (Sammy) Maniera, and Nicole Graskamp for taking care of all the organizational and technical matters that a research group is concerned with. You have been always helpful when I faced an issue and helped me to address and resolve it effectively and promptly.

Of course, I would like to thank the entire team of the Secure Software Engineering research group that I have been a part of for six and a half years for the great time. It really has been a pleasure to be part of this research group.


PhASAR

Contents

Abstract i

1 Introduction 1
1.1 A Motivating Example . . . 4
1.2 A Broader Perspective . . . 6
1.3 Contributions of This Thesis . . . 9
1.4 Structure of This Thesis . . . 11
Prior Publications . . . 12

2 Background 13
2.1 The Idea of Static Data-Flow Analysis . . . 13
2.1.1 Procedure Boundaries and Context Sensitivity . . . 15
2.1.2 The Zoo of Sensitivities . . . 16
Context Sensitivity . . . 16
Object Sensitivity . . . 16
Flow Sensitivity . . . 16
Field Sensitivity . . . 16
Path Sensitivity . . . 16
2.1.3 Distributive Data-Flow Analysis Problems . . . 17
2.2 The IFDS and IDE Frameworks . . . 19
2.3 Helper Analyses for Precise Whole-Program Data-Flow Analysis . . . 22
2.3.1 Control Flow and Callgraph Information . . . 24
2.3.2 Points-to and Alias Information . . . 25
2.3.3 Type Hierarchy Information . . . 25
2.3.4 Data-Flow Information and Client Analyses . . . 25
2.4 Soundness and Completeness . . . 27
2.5 Precision and Performance . . . 29
2.6 Static Versus Dynamic Analysis . . . 29
2.7 The LLVM Compiler Infrastructure . . . 30

3 PhASAR 33
3.1 Introduction . . . 33
3.2 Related Work . . . 34
3.3 Architecture . . . 35
3.4 PhASAR's Implementation . . . 38
3.4.1 Encoding an IFDS Analysis . . . 38
3.4.2 Encoding an IDE Analysis . . . 40

3.4.3 Encoding an Analysis Within the Monotone Framework 41
3.4.4 Use PhASAR as a Library 42
3.4.5 Handling of Intrinsic Functions and `libc` 42
3.4.6 A Note on PhASAR's Soundness 43
3.5 Scalability 43
3.6 The Need for Dedicated Debugging Capabilities 44
3.6.1 Instrumenting Static Analysis 45
3.6.2 Analysis Development Process 47
3.6.3 Implementation 48
3.6.4 Experience Report 48
Bug Finding and Detection of Anomalies 48
Performance Benchmarking for Optimizations 50
3.6.5 Related Work 54
3.6.6 Conclusions 54
3.7 The Burden of Correctly Handling Global Variables 55
3.7.1 Framework Support for Global Variables 55
3.7.2 Background and Problem Description 57
Globals in C and C++ 57
Built-in Typed Global Variables 57
Class/Struct Typed Global Variables 57
Global Con-/Destructors 59
Representation in LLVM IR 59
3.7.3 Modeling the Effects of Globals 60
Status Quo 60
Control Flows 61
Data Flows 62
3.7.4 Implementation 63
3.7.5 Case Study: Constant Propagation 63
An Analysis Writer's Perspective 64
Global Variables in Real-World Programs 64
Experimental Setup 64
RQ_1: Usages of Global Variables 65
RQ_2: Precision 66
RQ_3: Performance 66
3.7.6 Related Work 66
3.7.7 Conclusions 67
3.8 A Few Years Later: Designing Static Analysis Implementations 67
3.8.1 Experiences From Building a Static Analysis Framework 68
3.8.2 Background 69
Parametrization and Configurations 69
Analysis Styles 69
3.8.3 Lessons Learned 70
Modularity and Encapsulation 70
Accessing Information 71

Bugs and Debugging . . . 72
Parametrization, Configuration and Usability . . . 72
Flexible Usage Modes . . . 73
Analyzing C, C++, and LLVM IR . . . 74
Build Systems . . . 75
LLVM IR Generation . . . 75
Contributing Guidelines . . . 76
3.8.4 Related Work . . . 76
3.8.5 Conclusions . . . 77
3.9 Future Work . . . 77
3.10 Conclusions . . . 78

4 Variability **79**
4.1 Introduction . . . 79
4.2 Motivating Example . . . 82
4.3 Analysis . . . 83
4.3.1 Transforming Preprocessor Directives . . . 85
Phases of the Desugarer . . . 86
Parsing. . . . 86
Type checking. . . . 88
Rewriting. . . . 90
Desugaring C Type Specifications . . . 91
Desugaring Function Definitions . . . 92
Limitations of the Transformation . . . 93
4.3.2 Variational Data-flow Analysis . . . 94
Automated Lifting of Edge Functions . . . 95
Operations on Lifted Edge Functions . . . 96
Join. . . . 97
Composition. . . . 97
Equality. . . . 98
Evaluation. . . . 99
Why IDE Is the Ideal Framework of Choice . . . 99
4.4 Implementation . . . 99
4.5 Experiments . . . 100
4.5.1 Experimental Setup . . . 101
4.5.2 RQ_4: Analysis Correctness . . . 102
4.5.3 RQ_5: Analysis Efficiency . . . 104
4.5.4 RQ_6: Analysis Precision . . . 105
4.6 Related Work . . . 106
4.7 Conclusions . . . 108

5 Modularity **109**
5.1 Introduction . . . 109
5.2 Motivating Example . . . 112

5.3 Strategy . . . 114
5.3.1 Idea of the Algorithm . . . 114
5.3.2 Summary Generation . . . 116
Type Hierarchies . . . 116
Intra-Procedural Points-To Information . . . 116
Callgraphs and Inter-Procedural Points-To Information . . . 117
Data-Flow Information . . . 121
5.3.3 Merging Analysis Summaries . . . 123
Type Hierarchies . . . 124
Callgraphs and Points-To Information . . . 124
Fixed-Point Iteration for Callgraph and Points-To Graph . . . 124
Data-Flow Information . . . 126
Analyzing the Main Application . . . 127
5.3.4 Removing Dependencies Ahead of Time . . . 127
5.4 Implementation . . . 128
5.5 Experiments . . . 132
5.5.1 Experimental Setup . . . 132
5.5.2 RQ_7: Precision . . . 134
5.5.3 RQ_8: Performance . . . 135
5.5.4 RQ_9: Shortcuts . . . 136
5.6 Limitations . . . 137
5.7 Related Work . . . 137
5.8 Conclusions . . . 139

6 Incrementality **141**
6.1 Introduction . . . 141
6.2 Motivating Example . . . 144
6.3 Terminology and Notation . . . 147
6.3.1 Model of a Version Control System . . . 147
6.3.2 Model of Analysis Information . . . 147
6.4 Incremental Update Analysis . . . 147
6.4.1 Preparing Commit Metadata . . . 148
6.4.2 Change Scenarios . . . 148
6.4.3 Compute Whole Program Information . . . 151
6.4.4 Compute Incremental Updates . . . 151
Type Hierarchy Information . . . 152
Points-to Information . . . 152
Callgraph Information . . . 152
Data-Flow Information . . . 153
Answering the Client Analysis . . . 154
6.5 Implementation . . . 154
6.6 Evaluation . . . 158
6.6.1 Research Questions . . . 159

6.6.2 Experimental Setup . . . 159
Client analyses. . . . 159
Experimental process. . . . 160
Target Subject Selection. . . . 160
Execution environment. . . . 161
6.6.3 RQ_{10}: Performance . . . 161
6.6.4 RQ_{13}: Correctness . . . 162
6.6.5 RQ_{11}: Change Characteristics . . . 162
6.6.6 RQ_{12}: Helper Analyses . . . 163
6.7 Threats to Validity . . . 164
Internal Validity. . . . 164
External Validity. . . . 164
6.8 Related Work . . . 165
6.9 Conclusions . . . 166
6.10 Incrementality: Data . . . 166

7 Applications of PhASAR **177**
7.1 Combining Repository Mining and Static Code Analysis . . . 177
7.2 Static Configuration-Logic Identification . . . 178
7.3 White-Box Penetration Testing . . . 181
7.3.1 Running Example . . . 183
7.3.2 Overview of the Static Analysis Engine . . . 187
Pre-Processing . . . 187
Taint Analysis . . . 188
Symbolic Execution . . . 189
7.3.3 Design and Implementation . . . 190
Taint Configurations . . . 190
Taint Analysis . . . 191
Path Sensitivity and Performance Optimizations . . . 193
Symbolic Execution . . . 196
Path Constraints . . . 196
Symbolic Loop Finiteness Check . . . 198
Symbolic Out-Of-Bound Buffer Access Check . . . 199
Implementing Custom Symbolic Checks . . . 200
7.3.4 Results and How to Access Them . . . 200
Path Collection . . . 201
Emitting the Exploded Super-Graph . . . 201
Emitting Analysis Coverage . . . 202
Emitting Full JSON Reports . . . 202
7.3.5 Insights and Lessons Learned . . . 204
7.4 Conclusions . . . 206

8 Conclusions and Future Work **209**

Bibliography **213**

1 Introduction

Bugs and security vulnerabilities are a constant threat to all companies that produce their own software products. Not only can they become financially expensive—even risking a company's success—but they can harm people, too. Examples for bugs and vulnerabilities causing serious problems can be found plenty in the news. Apple's FileVaultBug caused the user's password to be stored on hard drive in clear text [Pro12]. The iOS unicode bug [iOS15] allowed one to crash iPhones by sending a carefully crafted text message which caused system crashes and reboot errors due to errors in iOS' unicode handling code. OpenSSL's heartbleed bug [Hea14] enabled anyone on the internet to read the memory of systems "protected" by the vulnerable OpenSSL version. Attackers could craft packets that trigger a buffer over-read and allowed them to read arbitrary sensitive materials such as names and passwords of users, secret keys, actual content, etc. To disable a warning message in valgrind [Ope08], a developer accidentally broke a random number generator in a particular version of OpenSSL with what was though to be a fix. NASA's mars climate orbiter [Har23] caused a crater on Mars worth around 190 million USD due to confusing the units in the computations for its trajectory. Boeing's 737 Max software flaws [Tra19] caused it to reject manual overrides of steering commands from the pilots for its automated counteracting of "pitch up", which has become necessary due to an engineering hack that has been made to fit larger engines, leading to a disaster with 346 deaths. This list could be extended virtually indefinitely.

Software tends to become increasingly larger and more complex [TG17, BBC+10]. Yet, developers are expected to keep software quality and correctness up—an almost impossible endeavor. While manual software walkthroughs, detailed code reviews or security audits can be employed, they can only be applied to the most critical parts of an application as they are too expensive to cover larger parts of an (existing) application, let alone the whole application. Tests can certainly uncover undesired behaviors and help debugging and hardening an application, and virtually all serious software projects employ (unit and integration) tests to verify the correctness of individual functionalities. However, since testing is a from of dynamic program analysis, it can only show behaviors emitted along the paths that the tests actually cover.

Static program analysis, on the other hand, analyzes all possible execution paths of the target program to make statements on a program's properties. It therefore cannot "overlook" possible program behavior. The majority of larger companies that develop their own software products not only employ code reviews, extensive testing and other dynamic program analysis techniques, but also employ their own static analyzers that are tailored to their specific needs. Google, for instance, utilizes its Tricorder infrastructure [SvGJ+15], Meta (Facebook) develops and employs Infer [DFLO19], Oracle uses its Parfait project [CKLS09], Amazon offers and uses CodeGuru [AWS22] for its AWS cloud

services, and IBM maintains and uses its WALA infrastructure [Wal19]. Other companies built complete businesses around static analysis and offer static program analyzers as a product. Synopsys offers Coverity(-SAST) [CS18], GrammaTech offers CodeSonar [Cod18] and Sonar (SonarSource) provides products like SonarQube [son23b] or SonarCloud [Son23a], for instance. These static analysis tools are tremendously useful and help developers to cope with large code bases by providing automated means to uncover software defects, bugs and vulnerabilities, and eventually help to improve the software under analysis [BBC+10].

But even automated approaches such as precise whole-program analysis quickly reach their limits when it comes to scalability, if being used in an unadept manner. Whereas lightweight, syntax-based static analysis and simple program checkers can provide useful results to developers in minutes, even for million-line programs, more heavyweight analyses that reason about semantic properties of programs are hard to scale. The detailed abstractions that are required to solve difficult semantic problems require large amounts of memory and computational resources. In addition, statically predicting program behavior, i.e., making statements on a program's properties is generally undecidable unfortunately and can be reduced to the halting problem [Ric53]. Every static analysis is thus necessarily incomplete, unsound or undecidable, or multiple of these. Static analysis writers are therefore constantly "working around" an undecidable problem by finding adequate analysis abstractions and algorithms that allow them to still compute meaningful information for a given target program.

Targeting programs written in C or C++ complicates matters further. Not only must analysis developers find adequate abstractions to deal with the oftentimes undecidable problems of static analysis in general, but, in addition, deal with the unique challenges that programming languages of the C family provide. In contrast to many other languages, C and C++ come with a separate preprocessor, deliberately unsafe type system, unrestricted use of pointers, *address-of* operator, and (for C++) virtual dispatch, making them notoriously hard to analyze. Even though those languages are around and in widespread use for decades, it took compiler writers many years to iron out most of the bugs in their implementations. The C programming language has now been around for more than half a century (first appeared 1972) and C++ is in its late thirties now (first appeared 1985) and still, these languages are the de facto standard for embedded systems and a variety of other performance, safety and security critical domains [Str18, Pro18], which makes them a desirable target to analyze statically to find bugs and vulnerabilities. Yet, scalable analysis frameworks that allow for deep, semantic static analysis targeting these languages are practically non-existent.

The main concern of this thesis is thus to improve the scalability of inter-procedural, semantic program analysis for the C and C++ programming languages. This will increase the amount of more complex analyses that can be solved on larger software projects and help finding bugs and vulnerabilities much earlier in the software development lifecycle.

In this thesis, we exploit the characteristics of modern software development that—at a first glance—seem to make program analysis particularly difficult to scale and instead, use them to make static program analysis indeed more scalable. Modern software development

is characterized by its extensive library usage [Sof18]. The vast majority of software projects also nowadays uses continuous integration and continuous deployment (CI/CD) techniques and the code bases of virtually all serious software projects are therefore organized with help of version control systems such as Git, Mercurial or Apache Subversion. The "unit" that drives software development is commits. The LLVM [LA04] project, for instance, usually receives more than 500 commits per week [LLV19]. Extrapolating the number of per-week commits for the LLVM project results in 72 commits per day, on average. Thus, a program analysis may not exceed a timeframe of 20 minutes if all commits of a day shall undergo an analysis. A new whole-program analysis would be triggered for every change made to the code that ignores all previously computed results. Depending on the size of the program under analysis, it is hard to impossible to run a precise, heavyweight static analysis within a few minutes. Instead, whole-program analyses are oftentimes postponed to the end of the day and are run in batch style during the nightly build. In such a scenario, they may not exceed the timeframe of eight hours [BBC+10]. Facebook recently reported that their code basis changes so frequently that they have trouble keeping their analyses up with the code changes such that the results are still relevant when being reported [HO18]. Several companies including Google started following the approach of *trunk-based development*, an approach that aims at getting the code to build and run correctly in the face of changing assumptions and requirements [Win17]. Since the approach treats the latest revision of a software project as the source of truth, it is highly advisable to obtain its analysis results as quickly as possible.

Our approach assumes that the target project is organized with help of a version control system. Initially, the target repository contains only the target source code and is "blank" w.r.t. static analysis information. Our approach makes C and C++ programs analyzable and transform the repository by amending it with static analysis information that is directly checked into the target project's code repository to make analysis information accessible to other developers and to improve scalability of subsequent analyses.

Surprisingly, the target C or C++ code within the target repository is not actually analyzable as is if the *complete* code shall undergo analysis. This is because every non-trivial C or C++ program represents an entire software product line that comprises exponentially many individual software products. Programs written in C or C++ are, in fact, written in a mixed (preprocessor and pure C or C++) language and each preprocessor `#ifdef` directive introduces two software variants. It is clear that generating and analyzing all of the exponentially many software products one by one cannot scale. VarAlyzer copes with the variability by first transforming the code into pure C or C++ by replacing all preprocessor constructs with semantically equivalent constructs in pure C or C++, effectively transforming compile-time variability into runtime variability. It then solves any given distributive data-flow problem in a variability-aware manner in a single analysis pass on the transformed code. This is a big step towards making the analysis of full software product lines a reality.

To cope with the analysis of libraries whose unnecessary repeated re-analysis, due to stability w.r.t. code changes, not only waste a lot of time, but also plenty of electrical energy—questioning even environmental aspects—we developed ModAlyzer. ModAlyzer allows one to analyze, summarize and persist parts of the code in a separate offline phase. Parts of the code that change seldomly, i.e., libraries components, can hence be summarized

and their persisted summaries can be directly checked into the targets project's repository for later use. Whenever a developer checks out a certain commit, they automatically obtain the precomputed library summaries which can then be employed while analyzing the actual application code that uses the respective libraries to massively reduce analysis times.

On top of ModAlyzer, we built IncAlyzer, an approach that is applicable to parts of the target project that change frequently. It uses information on code deltas from one commit to another provided by the target program's version control system to make an analysis incremental. IncAlyzer reduces analysis times by exploiting the fact that individual commits typically only change small parts of the target code. It only recomputes analysis information for parts of the code that have actually changed and updates the analysis results as necessary and too persists them directly within the associated commit using the target project's version control system.

In the following, we will present an example that highlights the difficulties of analyzing C and C++ programs, and explains the steps that we undertake to reduce analysis times to a minium. We will see that even the analysis of seemingly small and simple C and C++ programs can become complex very quickly.

1.1 A Motivating Example

```
1  #include <stdio.h>
2
3  int add(int i, int j);
4  int identity(int i);
5  void magic_initialization(int *i);
6
7  int (*f)(int) = &identity;
8
9  int add(int i, int j) { return i + j; }
10
11 int identity(int i) { return i; }
12
13 void magic_initialization(int *i) {}
14
15 int main(int argc, char **argv) {
16   printf("Hello, World!\n");
17   int i;
18   magic_initialization(&i);
19   int j = add(i, 42);
20   int k = f(j);
21   return k;
22 }
```

Listing 1.1: A *Hello, World!* program in C. The program comprises undefined behavior.

The *Hello, World!* program in Listing 1.1 prints the string `Hello, World!` to the command line and performs simple integer arithmetic. The program comprises undefined behavior as the uninitialized local variable `i` is used to compute `k`'s value that is returned to the operating system at the end of `main`.

This small example allows one to elaborate on numerous interesting facts. First, the severity of static program analysis representing an undecidable problem is impressively shown by the fact that even modern C compilers do not issue a warning or an error for the obvious bug. This is because `i`'s address is taken using the *address-of* operator (`&`) and passed to the `magic_initialization` function. To correctly reason about the status of `i`'s initialization would require the compiler to compute precise points-to information which, again, is generally undecidable and very expensive when precise points-to information is desired. The compiler is thus not even trying to prove any properties on `i`'s initialization. This shows that precise static analysis requires a multitude of information computed by additional helper analyses such as points-to and call information to reason about a program.

Second, the compiler community and the program analysis community have very different motivations and incentives. Whereas compiler writers would, of course, also like to detect these kinds of bugs, they need to ensure that a program still compiles sufficiently fast to the desired target language. Compilation of larger C and especially C++ projects already requires a substantial amount of time. More sophisticated static program analyses would exceed any acceptable time frame for compilation; they thus cannot be integrated into the compiler [Bab18]. The static analysis community on the other hand, can generally accept longer analysis times as long as an analysis is capable of finding real, actionable bugs and vulnerabilities. And while static analysis that is used as a basis for compiler optimization has to be sound, analyses that are used for finding bugs and vulnerabilities may be unsound. But even this extended time frame and the relaxation of soundness has its limits due to modern software development workflows. Users of CI/CD pipelines, for instance, crucially require their pipelines to successfully pass, which typically includes static analysis tools that need to approve the current state of the software, to continue with the software development, e.g. merging of feature branches, deployment, etc. Facebook, for instance, faces a real challenge to keep up with the analysis and report the results such that they are still relevant and actionable when they are available [HO18]. Google has given up on running expensive inter-procedural, i.e., whole-program analysis on every commit that is made to their code base [Bab18]. Instead, they collect a whole bunch of code changes and then run an expensive whole-program analysis in batch style at the end of the day.

Yet another, less obvious problem lurks hidden within the small *Hello, World!* program. The program in Listing 1.1 does not actually show one program, but many programs. This is due to the *Hello, World!* program's include directives and becomes evident when looking at the (partially) preprocessed version of the program. A brief inspection with the Clang compiler uncovers that the "program" comprises 622 preprocessor symbols[1], 92 `#ifdef` directives[2] and (transitively) includes 16 header files[3]. Virtually every realistic C or C++ program is actually a software product line that comprises various different software products that can be generated. This is because C and C++ compilers are language processing systems, and C and C++ programs are actually written in a mixed language that comprises the preprocessor language and pure C or C++. If one wishes to analyze a

[1] `$ clang -dM -E code.c`
[2] `$ clang -M -E code.c|grep -o -E "\.h"|wc -l`
[3] `$ clang -M -E code.c|grep -o -E "/.*\.h"|tr ' ' '\n'|xargs grep -E "\#ifdef"|wc -l`

complete C or C++ software product line, one would first need to generate each and every possible software product and then conduct a static analysis on each generated product. Since the number of software products grows exponentially in the number of preprocessor `#ifdef` directives, it is clear that this cannot scale.

Even the single `stdio.h` include introduces a large number of (preprocessor) symbols and various functionalities from the C standard library `libc`. Realistic programs include hundreds of header files for C's `libc` or C++'s standard template library (STL) and potentially various other libraries. However, since the functionalities that these libraries provide are only applied while developing the application, but not modified, analyzing them over and over for each analysis run does waste lots of computing resources. Instead, it is desirable to analyze these libraries only once and summarize their effects on the analysis. When analyzing the actual application code, those summaries can then be integrated into the analysis avoiding reanalysis of the unchanged library code. The amount of analysis results that can be reused—rather than expensively recomputed—can be further increased when one assumes that the *Hello, World!* program is developed using CI/CD. After running an initial, inter-procedural, flow- and context-sensitive, alias-aware *uninitialized variables* analysis that can easily uncover the missing initialization of variable `i`, a developer may commit a fix that correctly initializes `i` in `magic_initialization` to `42`. This change, recognized and recorded by the project's version control system, only affects a very small part of the data-flow and points-to information required to answer an uninitialized variables analysis. Information on control flows, type hierarchy, virtual function tables, and callgraph is not affected at all. Besides avoiding unnecessary and potential expensive reanalysis for parts of the code for which developers know that they do not change, it is thus advisable to further make an analysis incremental to reduce the necessary computations for answering a concrete client analysis to a minimum. Incrementalizing static analysis, however, requires one to solve the challening problem of keeping the incremental analysis information in synchronization with the code changes such that the computed results are equivalent to a full reanalysis.

One can clearly see that even though traditional, precise whole-program analysis is tremendously useful, it quickly runs into massive scalability issues even for small programs and does not match modern software development that is characterized by an extensive use of libraries and continuous integration/continuous deployment (CI/CD) when being run naively for each change made to the target code.

Next, we describe important, existing analysis approaches that address scalability.

1.2 A Broader Perspective

Over the past decades, a vast amount of research has been conducted to improve the scalability and precision of static whole-program analysis and to deal with the especially complex languages from the C family.

General algorithmic data-flow analysis frameworks have been developed and evolved from simple, yet powerful, concepts for intra-procedural analysis [KU77] to more and more sophisticated algorithms that allow for inter-procedural analysis [PK13, SP78] and

ones that exploit certain properties of the analysis problem to be solved [RHS95, SRH96, RSJ03, Bod18] allowing for solving analysis problems more efficiently. Yet more advanced approaches that build on top of these combine multiple instances of those frameworks to further increase precision [SAB19a, LTKR08] and to allow for solving problems in a flow-, fully context- *and* fully field-sensitive manner while avoiding undecidability [Rep00] for many realistic programs.

Yet, finding infrastructure that implements the above concepts and allows for the analysis of C and C++ programs can only be found sparsely. The SVF [SX16b] framework allows one to conduct inter-procedural static analysis targeting LLVM [LA04] intermediate representation using spare value flow. GDFA [KSK09] is a generic data-flow solver for GCC that implements traditional data-flow analysis [KU77] as a bit-vector framework. LLVM's front-end Clang for C-like languages offers infrastructure that allows one to formulate lightweight syntactic analyses based on the abstract syntax tree (AST) of the target program. Clang's static analyzer [Cla18b] additionally offers checks that employ symbolic execution to find common bugs in the target code. However, it struggles with scalability when conducting inter-procedural analysis, since many checks require in the worst case exponential time to find bugs. It therefore can only be applied on a compilation-unit level and not in a whole-program setting. And while LLVM, of course, offers general infrastructure to formulate custom analysis and transformation (or optimization) passes, it does not offer generic data-flow solvers. Instead, compiler passes that come with LLVM are individually implemented and target program optimizations. The following quote from 2009, pointed out by Scott Meyers, shows the severity of the problem:

> Does anybody know a [...] refactoring tool for C++ that works ... with large code bases [...]? I tried [...] again and again over the last years: [none] were at all usable. [ADA09]

The question has been updated in March 2015:

> By my opinion the answer to this question still is 'NO'. [ADA09]

A commenter of the Stack Overflow post "Clang Static Analyzer doesn't find the most basic problems" from March 2017 states:

> I haven't used this particular static analyser, but many others. As a rule of thumb, always assume they are broken beyond repair. [Cod17]

Dozens of works try to deal with C and C++'s preprocessor and the software product lines that the preprocessor's directives effectively establish [KGR+11, GG12, KATS12, CEW12, BRTB12, BTR+13, MDBW15, CCS+13, Dim16, GJ05, KOE12, KKHL10, Her20]. And yet, the only available data-flow analysis capable of analyzing software product lines encoded in C and C++ is intra-procedural only [LvRK+13, RLJ+18]. SPLlift [BTR+13] presents a family-based approach that allows one to efficiently conduct inter-procedural data-flow analysis. This approach, however, is restricted to analysis problems that can be expressed using distributive flow functions and require a finite—and ideally small—analysis domain, and is only available as a prototypically implementation for a seldom-used Java-dialect.

Bugs such as Apple's file vault bug [Pro12] show that maintaining and analyzing software variability is still an issue—even for large companies of the computing industry.

The literature offers several approaches to make static program analysis compositional [OPS92, CDG93, RR01, Dwy97, HR96, RRL99, GRS00, RMR04, XHN05, TWX+17, CCP17]. Existing techniques for compositional static analysis typically focus on either data-flow or points-to analysis only. Rountev et al. presented an approach to pre-analyze Java libraries using *Interprocedural Distributive Environments* (IDE) [SRH96] to generate reusable data-flow summaries. StubDroid [AB16] generates precise library models for *Interprocedural Finite Distributive Subset* (IFDS) [RHS95]-based taint-analysis problems. Both approaches, however, assume the existence of whole-program points-to and callgraph information. Several works use partial summaries for points-to information computed using context-free language (CFL-)reachability [WR99, LSXX13, SXX12]. IDEal [SAB17] is an alias-aware extension to IDE [SRH96] that embeds the alias analysis Boomerang [SNAB16] into the IDE solver implementation Heros [Bod12]. Averros [AL13] uses the *separate compilation assumption* and Java's constant pool [Jav18] to generate sound and precise callgraphs without actually analyzing library code in order to generate a placeholder library. Klohs et al. described a situation that allows one to remove data-flow dependencies without compromising precision [Klo08]. Tree-adjoining languages [TWZ+15] and Dyck context-free language reachability [TWX+17, CCP17] can be used to increase the effective library summarization by computing reasonable conditional summaries that potentially enable greater summary reuse. Early versions of Facebook's Infer [CD11] used separation logic to allow for the compositional analysis of heap-based programs. The approach computed bottom-up summaries using bi-abductive inference [CDOY09, BGS18], which could then be used in different calling contexts.

Yet other work attempts to make static analysis incremental. The line of research conducted by Szabó et al. employs a declarative approach to make static data-flow analysis incremental [SEV16, SVE17, Sza21, SBEV18]. These approaches, however, are currently limited to intra-procedural analysis. Reviser [AB14] presents an incremental version of the conceptual *Interprocedural Finite Distributive* (IFDS) [RHS95] and *Interprocedural Distributive Environments* (IDE) [SRH96] frameworks that enables one to conduct flow- and *fully* context-sensitive inter-procedural data-flow analysis. The Reviser approach, however, is limited to incrementally compute data flows only. In particular, it cannot incrementally compute information such as the (inter-procedural) control-flow graph that are required by the IFDS/IDE frameworks. Other approaches aim at computing concrete analysis problems like pointer analysis incrementally [LHR19, CNDE05].

This thesis takes a different approach at making precise, semantic static analysis scale to large, real-world C and C++ programs. Instead of focussing on scaling individual static analyses, we aim at scaling the whole static analysis stack that is required to answer a concrete client analysis (such as the *uninitialized variables* analysis presented in Section 1.1) for finding bugs and security vulnerabilities using an *integrated approach*. This includes—besides data-flow information—information on pointers and aliasing, control flows, callgraph, type hierarchy and virtual function tables. To achieve scalable, integrated static analysis, we are using *composition* and *distributivity*. In summary, this thesis follows the goal:

Precise, inter-procedural static data-flow analysis can be scaled to large programs and matched to modern software development through composition and summarization of analysis information.

1.3 Contributions of This Thesis

In this thesis, we present PhASAR, a novel, open-source static program analysis framework. PhASAR has been built on top of the LLVM compiler infrastructure [LA04] and allows its users to solve arbitrary data-flow problems in a fully automated manner. It provides various algorithms to compute all helper analyses that are required to conduct concrete client analyses. PhASAR comprises, among others, a type hierarchy and virtual function table as well as various pointer and callgraph analyses. On top, it provides various generic data-flow solver implementations. The framework follows a clean structure and the various different components of the static analysis stack are all modeled as interfaces. This allows individual components to be easily replaced in a concrete analysis setup. A user only needs to provide a description of the data-flow problem they wish to solve. This description corresponds to a concrete implementation of an analysis problem interface, which is handed over to the desired solver for computing data flows to answer a client analysis.

PhASAR's functionalities can be accessed through its command-line tool `phasar-llvm`. However, the framework can also be used as a library which allows its users to quickly build or prototype new program analyzers on top of it. The framework is used to implement and evaluate the new program analysis strategies and algorithms presented in this thesis. Meanwhile, our open-source PhASAR framework, which is available on Github under the permissive MIT license at `https://github.com/secure-software-engineering/phasar`, has received quite some attention from other researches as well as practitioners as its 922 stars and 140 forks[4] on Github demonstrate.[5]

To deal with the "program explosion" caused by the infamous preprocessor `#ifdef` directives that developers use to encode multiple software variants in a common code base, this thesis contributes VarAlyzer. The VarAlyzer approach lets its users conduct arbitrary inter-procedural, distributive data-flow analyses on entire software product lines (SPLs). Compared to the naive *product-based* approach, in which one would first generate each and every software variant and then analyze each such variant individually, VarAlyzer is a *family-based* approach. It avoids generating the exponentially many software products and instead first transforms the software product line, replacing preprocessor directives with semantically equivalent constructs in plain C, and then conducts a variability-aware data-flow analysis that computes the results for all software products in a single analysis run. VarAlyzer's results are annotated with the presence conditions (feature constraints) under which they hold. VarAlyzer for the first time allows one to conduct complex, inter-procedural data-flow analysis with infinite lattices on SPLs written in the C programming language. It avoids exponential runtime by design and its runtime requirements are not

[4] Probably by many other poor PhD students that fight the complexity dragon of static analysis for C and C++.
[5] As of 21pm July 23rd, 2024.

affected by the number of preprocessor `#ifdef`s. VarAlyzer's results coincide with those produced by the *product-based* approach.

This thesis further contributes ModAlyzer, an approach that avoids repeated re-analysis for parts of the target program that do not change frequently. While modern software development heavily relies on libraries, these libraries would be unnecessarily re-analyzed for each analysis run on a target program that uses them. ModAlyzer allows one to independently analyze certain parts of the code in a separate offline phase. In contrast to previous works that mainly focus on making individual static analysis compositional, ModAlyzer computes summaries for the complete static analysis stack that is required to solve a concrete client analysis. This includes type hierarchy, points-to, control-flow and call-graph and data-flow information. The approach then persists these summaries for later use. While analyzing the actual application code, ModAlyzer consults the previously computed summaries to avoid unnecessary reanalysis of the application's library components. As our evaluation shows, this strategy is quite effective and allows the analysis to finish 72% faster, on average, while providing the same results as a matching whole program analysis.

To make whole program analysis fit modern continuous integration/continuous deployment (CI/CD) workflows, this thesis contributes IncAlyzer. The IncAlyzer approach enables one to compute information (similar to ModAlyzer) for the complete static analysis stack in an incremental manner. Individual commits typically only introduce small changes to a code base making running an immediate whole-program analysis seem wasteful. IncAlyzer exploits the fact that large parts of the analysis information computed on a revision C_i of a program is still valid in the next revision C_{i+1}. It hence uses commit information provided by the target project's version control system to determine changes made to the code from one revision to another to invalidate and recompute analysis information only for the parts of the program that actually changed. It then applies a final repropagation step and as a result produces static analysis information that is equivalent to those of a matching whole program analysis conducted on the latest revision of the target project.

Combining the approaches presented in this thesis enables interesting software development workflows as the shown in Figure 1.1. We briefly explain this overview figure from top to bottom. In this example, we assume that the target project comprises the actual application code and a few additional libraries. The target project is made analyzable with help of the VarAlyzer approach that enables analysis of software product lines as a whole. Starting at a certain commit of the target project under version control, PhASAR can be used to compute whole-program static analysis information. Any libraries used by the project are summarized in this process using ModAlyzer. The resulting analysis information can be persisted and directly checked into the project under analysis. Another developer can check out this new commit together with the persisted analysis information. PhASAR can parse the persisted information to instantly obtain the analysis information of the code. Any interesting analysis findings such as bugs or potential (security) vulnerabilities are therefore also instantly available and can be addressed by a developer. The fixed code can then be incrementally reanalyzed by IncAlyzer and checked into the software repository. Due to ModAlyzer, libraries require no reanalysis whatsoever as long as they are not changed. Another developer might have worked on a separate feature branch to add new functionalities to the project. The commits of the feature branch also come with their

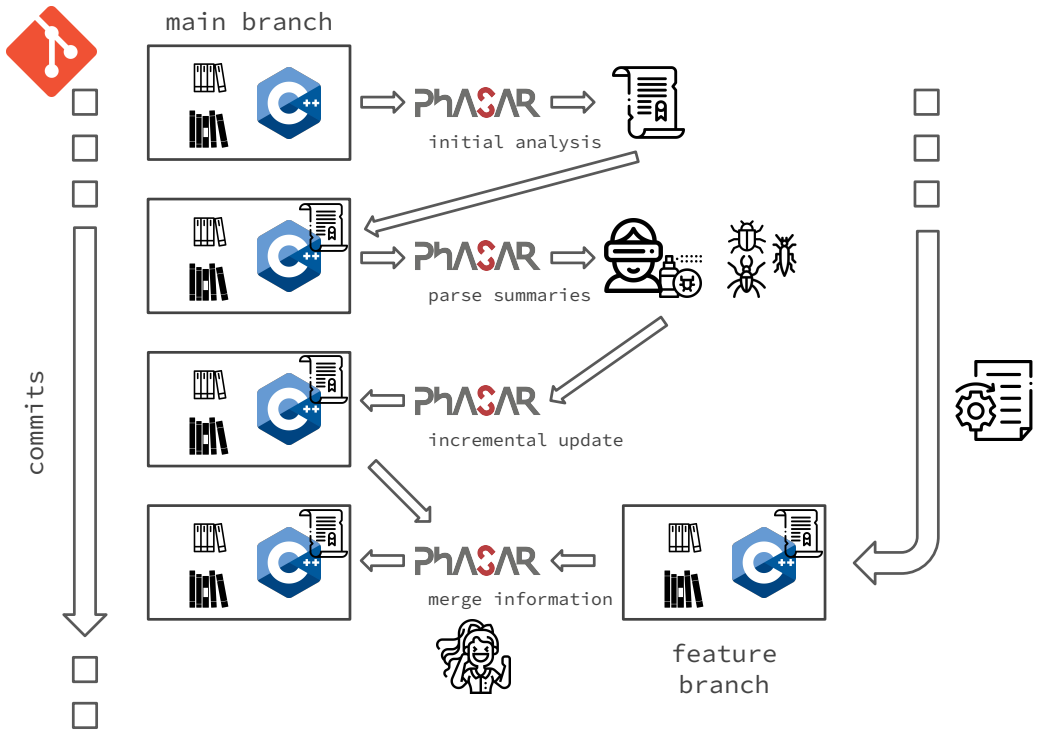


Figure 1.1: An overview of an exemplary workflow that the analysis approaches presented in this thesis enable.

corresponding persisted analysis information. When merging the feature branch back to the main branch of the repository, IncAlyzer allows for efficiently merging analysis information and recomputes outdated analysis information as necessary based on the delta in the code that it obtains from the version control system of the target project.

In our final contribution, we present some of PhASAR's applications. We first present PhASAR's applications in two larger academic projects to show some interesting problems that our framework could help to address. We then further underline its usefulness by presenting a large industry project that we conducted with one of the worlds leading telecommunications companies to highlight the real-world challenges that we have been facing while applying PhASAR and state-of-the-art static analysis to solve demanding problems in today's software development.

1.4 Structure of This Thesis

The remainder of this thesis is structured as follows: In Chapter 2, we give an introduction to the necessary concepts that are used throughout the remainder of this thesis, including the different forms of static (data-flow) analysis. Chapter 3 presents PhASAR, the analysis infrastructure that we use to implement and evaluate our approaches that we describe in the chapters thereafter. It also presents some interesting technical challenges and how we

address them, and shares the experience that we gained from implementing static analysis over the last years. In Chapter 4, we present our VarAlyzer approach that allows for statically analyzing software product lines written in C in an inter-procedural manner. We then describe how we scale precise static analysis to large and realistic C and C++ program by making static analysis compositional in Chapter 5. In Chapter 6, we present how to make an analysis fit for modern software development in which software undergoes frequent changes. Both approaches, ModAlyzer and IncAlyzer improve an analysis's scalability and can be enabled together. Chapter 7 presents actual applications of our static analysis framework PhASAR and the approaches we have built on top of it. This chapter presents academic works that successfully utilize our infrastructure to obtain novel insights that could have been hardly produced without PhASAR. We also present valuable insights that we gained from applying PhASAR in industry and discuss areas in which data-flow analysis is already quite successful and others which still need more attention from academia to allow it to solve important and realistic problems that occur in practice. Lastly, we conclude in Chapter 8.

Prior Publications The work on our analysis infrastructure described in Chapter 3 has been originally published at the regular tool paper track of the *25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS 2019) [SHB19]. The experiences and shortcomings of initial versions of PhASAR have been shared, discussed and addressed in the *8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (SOAP) [SLHB19] and at the engineering tack of the *21st IEEE International Working Conference on Source Code Analysis and Manipulation* (SCAM) in [SLHB21] and [SSS+21], respectively. The work on variability-aware data-flow analysis presented in Chapter 4 has been previously published in *Automated Software Engineering – An International Journal, Volume 29, Springer Nature* (AUSE) [SGP+22] and presented at *Feature-Oriented Software Development* (FOSD) meeting 2022 and as part of the journal-first track at the *37th IEEE/ACM International Conference on Automated Software Engineering* (ASE 2022). Our ModAlyzer approach to compositional static analysis presented in Chapter 5 has been previously published at the main research track of the *35th European Conference on Object-Oriented Programming* (ECOOP 2021) [SHB21]. This work has been awarded a distinguished paper award. The IncAlyzer approach that is concerned with making static analysis incremental presented in Chapter 6 is currently being prepared for submission. The author of this thesis has made significant contributions to the applications that are presented in Section 7.1 and Section 7.2; the work on integrating program analysis and repository mining presented in Section 7.1 has been published in *ACM Transactions on Software Engineering and Methodology* (TOSEM) [SBS+23]. The research on static configuration-logic identification has been published on *arXiv.org* and is available at [ASR+23]. The experiences and insights from applying PhASAR in an industrial context shared in Section 7.3 have been gained in a 15 months project with one of the leading telecommunications corporations. In this industry project, the author of this thesis was the project lead.

2 Background

In this chapter, we introduce the fundamentals of static analysis and present its aspects that are relevant for this thesis.

2.1 The Idea of Static Data-Flow Analysis

In this section, we briefly discuss the fundamental idea of static data-flow analysis and then present complementary aspects that are required to conduct precise, semantic data-flow analysis.

Static data-flow analysis is—among symbolic execution, abstract interpretation, and type and effect systems—one of the various static program analysis techniques that reasons about a program's properties without actually executing it. The goal of static data-flow analysis is to make statements about a property of interest for a given target program. A concrete data-flow analysis (implementation) answers a concrete question on a property of interest by providing a *client* analysis with statically computed data flows. Information on the property of interest can be used as a basis for compiler optimizations or to decide whether a program possibly contains bugs or is vulnerable to certain attacks, for instance.

Static data-flow analysis propagates data-flow facts that capture information on the property of interest through the target program's control-flow graph. A program's control-flow graph that is amended with data-flow facts is sometimes also referred to as *exploded graph*. Flow functions capture the effects of each program instruction on the data-flow facts and therefore, the property of interest. In the simplest case, sets of data-flow facts of a data-flow domain D are propagated through the target program's control-flow graph. Figure 2.1 depicts this situation. A set of data-flow facts X is handed as input to a flow function f for an instruction i which produces as an output a new set X', thereby describing i's effect on the property of interest.

Each analysis needs to organize its data-flow domain D using an underlying lattice $\mathcal{L}$ that describes the "kind of information". $\mathcal{L}$ can be represented graphically as a Hasse

Figure 2.1: A flow function f for an instruction i and its effect on data-flow facts.

$$X \sqcap X' := X''$$

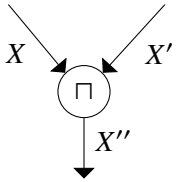


Figure 2.2: A merge operator and its effect on data-flow facts.

diagram. There are two special, commonly used lattice elements: bottom $\bot$ and top $\top$ that specify "no information" and "all information" (or "statically undecidable"), respectively. A concrete analysis requires at least a semi-lattice, i.e., one that specifies a $\top$ element.

When analyzing a program that comprises branching, there are situations in which control-flow edges of two (or more) different branches lead to a common successor instruction. In this situation, depicted in Figure 2.2, the possibly varying information computed along different control-flow paths must be merged before the analysis can continue propagating the resulting data-flow information further along the program's control-flow graph. Inputs of the merge operator are elements of the analysis' lattice and its output is a single lattice element that describes how information is merged. Whenever the analysis' merge operator is applied, the amount of analysis information described by the operation's output must grow monotonically, i.e., must be transferred higher up in the lattice—growing towards $\top$, the most imprecise element of the lattice. This is to ensure termination.

The data-flow information is propagated through the control-flow graph and the flow functions and merge operator are (re)applied in a traditional worklist algorithm until the data-flow information stabilizes and is no longer changed. Once the analysis' fixed-point is reached, the solution can be read off by checking the data-flow facts at each instruction that one is interested in.

This natural idea of basic data-flow analysis is known as *Monotone Framework* and has been formalized by Kam and Ullman in 1977 in [KU77]. The monotone framework describes a generic algorithm to solve any given data-flow problem that is specified with the following five parameters:

i analysis direction $\uparrow\downarrow$

ii analysis domain D

iii flow function(s) f

iv merge operator $\sqcap$

v lattice $\mathcal{L}$

The monotone framework has been designed to solve data-flow problems on individual procedures. It is not suited for solving data-flow problems in an inter-procedural setting, in particular when it comes to context-sensitivity. In the following, we therefore detail on how to make static data-flow analysis inter-procedural.

2.1.1 Procedure Boundaries and Context Sensitivity

While most compiler optimization passes such as LLVM's constant propagation implement a concrete instance of the monotone framework, they analyze functions only intra-procedurally, i.e., they do not follow function calls within the analyzed function nor analyze across the function's procedure boundaries. This is partially because of speed, but also because of *precision* and *soundness*. More complex inter-procedural analyses are typically too expensive to be integrated into the compiler [Bab18]. But even though LLVM implements several inter-procedural analysis and optimization passes that aim at de-virtualizing function calls or merging (global) constants, for instance, these analyses are very shallow and need to back off as soon as the code comprises more complex constructs or uses dynamic features. We discuss soundness separately in Section 2.4. When applying the monotone framework in an inter-procedural setting, the set of data-flow facts can be mapped and propagated into potential call targets at any given call site within the function under analysis. However, at a callee's exit points, i.e., return or throw instructions, an analysis formulated in the (unmodified) monotone framework does not know to which of the potentially many return sites of the callee the data-flow information has to be propagated, since it does not maintain information on calling contexts. An analysis would need to propagate the information to all possible return sites, which includes program paths that are infeasible at runtime. This introduces so much imprecision that prevents the analysis from computing any meaningful results.

To cope with the problem of distinguishing calling contexts, the individual data-flow facts can be amended with so-called *call-strings* as presented in [SP78], for instance. The call-strings approach amends each data-flow fact with a call stack. Whenever a data-flow fact is mapped into a callee, the call site's respective return site is pushed onto the fact's respective stack. When mapping a data-flow fact back to the caller at a callee's exit point, the stack is consulted to ensure that data-flow facts are only propagated to the matching return site. To cope with recursive functions that would otherwise cause non-termination, however, the call-strings must be limited in length. In practice, the call-strings approach is typically k-limited with $k \in \{1, 2, 3\}$, since larger ks massively impede scalability for larger target programs. This is often referred to as k-context sensitivity. Propagating data-flow facts down the program's call stack deeper than k causes the most recent return site to be put onto the call stack while the oldest return site is discarded. When propagating a data-flow fact upward the call stack, the flow facts are propagated to the return site on top of the call stack before the return site is then removed from the stack. Once the call stack is empty and a flow fact still needs to propagated further up the call stack, it is propagated to all possible return sites context-insensitively.

Besides (calling-)context sensitivity, a static data-flow analysis can exhibit a variety of sensitivities as discussed in the following.

2.1.2 The Zoo of Sensitivities

In this section, we briefly explain the various sensitivities that a data-flow analysis can exhibit. Generally, it holds that the more sensitivities an analysis exhibits, the more precise data flows it can compute.

Context Sensitivity A context-sensitive analysis maintains and matches information on calling contexts. Context-insensitive analyses do not distinguish between calling contexts and hence propagate data-flow information back to all potential return sites after analyzing a given callee function. An instantiation of the unmodified monotone framework in an inter-procedural setting serves as an example for a context-insensitive analysis as described in the above. Analyses formulated this way are typically too imprecise to compute useful information for realistic data-flow problems.

Object Sensitivity An object-sensitive analysis distinguishes between different objects of the same type and is relevant for call resolution and precise modelling of field accesses. Object sensitivity can also be seen as an alternative to (calling-)context sensitivity. It attaches information on the creation of data-flow facts rather than call sites at which they have been passed into a call target to make an inter-procedural analysis context sensitive. This is achieved by reusing information that has been computed for previous calls under the same context. This approach to context-sensitive inter-procedural analysis is known as *value-based* approach [PK13]. However, object-sensitive analysis too needs to be k-limiting to avoid infinite field-access paths for recursive data structures such as linked lists.

Flow Sensitivity A flow-sensitive analysis respects the order of program instructions. Each program instruction i has its own input and output sets of data-flow facts that it is associated with, i.e., hold before and after instruction i. Data flows that have been computed in a flow-insensitive manner hold at all instructions of the program (or function, in an intra-procedural setting) under analysis.

Field Sensitivity An analysis that is able to distinguish different fields of a user-defined type typed or array typed variable is field sensitive, whereas analyses that merge all of a variable's fields are field insensitive. Similar to context sensitivity, the depth of fields managed, i.e., distinguished by an analysis must be limited to avoid running into infinite loops for recursive data structures such as linked lists.

Path Sensitivity The results of a data-flow analysis, as described in Section 2.1, is a control-flow graph for which each node representing a program instruction i is associated with two sets of data-flow facts: one that holds before and one that holds after instruction i. Information on control-flow paths along which a certain data-flow fact has reached a particular node in the control-flow graph is typically not recorded due to high computational costs. Analyses that do record control-flow paths between the program point at which a data-flow fact has been generated and the program points it has been propagated to until it is

killed, if it is killed at all, are path sensitive. Although path sensitivity is rather expensive, it can yield very useful information as we will further examine in Section 7.3.

2.1.3 Distributive Data-Flow Analysis Problems

Next, we elaborate on data-flow problems whose flow functions distribute over the merge operator. This property can be used to solve static data-flow analysis problems in an inter-procedural manner using the *summary-based* approach [SP78]. This approach allows for infinite ($k = \infty$) context sensitivity and was proposed by Sharir and Pnueli in [SP78] as an alternative to the call-strings approach. It enables the efficient solving of data-flow problems by making them compositional.

For analyses formulated in the monotone framework the following monotonicity holds

$$f(x) \sqcap f(y) \sqsubseteq f(x \sqcap y), \text{ with } x, y \in D \tag{2.1}$$

When solving data-flow problems, one would ideally like to compute the optimal, i.e., most-precise so-called *meet-over-all-paths* (MOP) solution. The MOP solution is computed by analyzing each control-flow path of the program separately and merging information obtained along all possible program paths at the end (cf. $f(x) \sqcap f(y)$). The MOP solution can be formally written as

$$\forall i \in P : \text{MOP}(i) = \sqcap\{f_p(v_{init}) \mid p \text{ is a path from } i_0 \text{ to } i\} \tag{2.2}$$

where f_p is a composed "flow function" for path p from program P's entry point i_0 to instruction i and $v_{init} \in D$ is an initial value.

The MOP solution is, however, generally uncomputable, since every program whose control-flow graph contains non-trivial loops comprises an infinite number of program paths. Analyses formulated in the monotone framework thus compute the so-called *maximal fixed-point* (MFP) (or *merge-first*) solution, instead. To obtain the MFP solution, data-flow information is merged *first* (sometimes called "early") at control-flow merge points, before it is further propagated along the program's control-flow graph and consumed by the subsequent flow functions (cf. $f(x \sqcap y)$). The MFP solution can be formally written as

$$\text{MFP}(i_0) = v_{init} \tag{2.3}$$

$$\text{MFP}(i) = \sqcap\{f_h(\text{MFP}(h)) \mid h \in preds(i)\} \tag{2.4}$$

The MFP solution is expressed in terms of predecessor relationships in the control-flow graph and the merge of information takes place at control-flow merge points. The MFP solution is a sound (and computable) overapproximation of the MOP solution, i.e., $\forall i \in P,\ \text{MOP}(i) \sqsubseteq \text{MFP}(i)$. This is because because Equation (2.1) holds, if f is monotone.

For distributive analysis problems it holds that

$$f(x) \sqcap f(y) = f(x \sqcap y), \text{ with } x, y \in D \tag{2.5}$$

For distributive analysis problems the MFP solution hence coincides with the most precise and theoretically optimal MOP solution.

Full constant propagation is a textbook example for an analysis that is not distributive. Computing the maximal fixed-point solution for a full constant propagation on the program shown in Listing 2.1 using the monotone framework [KU77] produces the exploded graph shown in Figure 2.3. One can easily see that information obtained for the program variables `x` and `y` is merged to $\top$ at the control-flow merge point to indicate that these variables no longer carry constant values after the control-flow merge point. Therefore, `z`'s value cannot be determined precisely and it is associated with $\top$ at instruction `z = x + y`. Computing the meet-of-all-paths solution on the program shown in Listing 2.1 is decidable, since it's control-flow graph does not contain loops and yields the data-flow information `x` $\mapsto \top$, `y` $\mapsto \top$ and `z` $\mapsto 1$ at instruction `z = x + y`. This example shows that for full constant propagation the MOP solution is more precise than the MFP solution and is able to precisely determine `z`'s value at the end of the function. This is because information is merged at the end after data-flow information has been computed for each program path in separate. For our example program, `z`'s value can be precisely determined as 1 at instruction `z = x + y` in both program paths and the merge thus yields its associated value 1.

This example shows that Equation (2.5) is not satisfied and full constant propagation is indeed not distributive.

Linear constant propagation, a variant of constant propagation that computes and propagates variables that depend on constant values or linearly ($y = ax + b$, with a and b constant literals) depend on constant variables can, however, be expressed using distributive flow functions as we show in the next section.

```
1  int main(int argc, char **argv) {
2    int x;
3    int y;
4    int z;
5    if (argc > 1) {
6      x = 1;
7      y = 0;
8    } else {
9      x = 0;
10     y = 1;
11   }
12   z = x + y;
13   return z;
14 }
```

Listing 2.1: An example program that requires non-distributive flow functions for solving a constant propagation without loss of precision.

We will next show how distributive data-flow problems can be solved efficiently and in a compositional manner using the algorithmic IFDS [RHS95] and IDE [SRH96] frameworks. We will use these frameworks to compute data flows in our analysis approaches presented in this thesis.

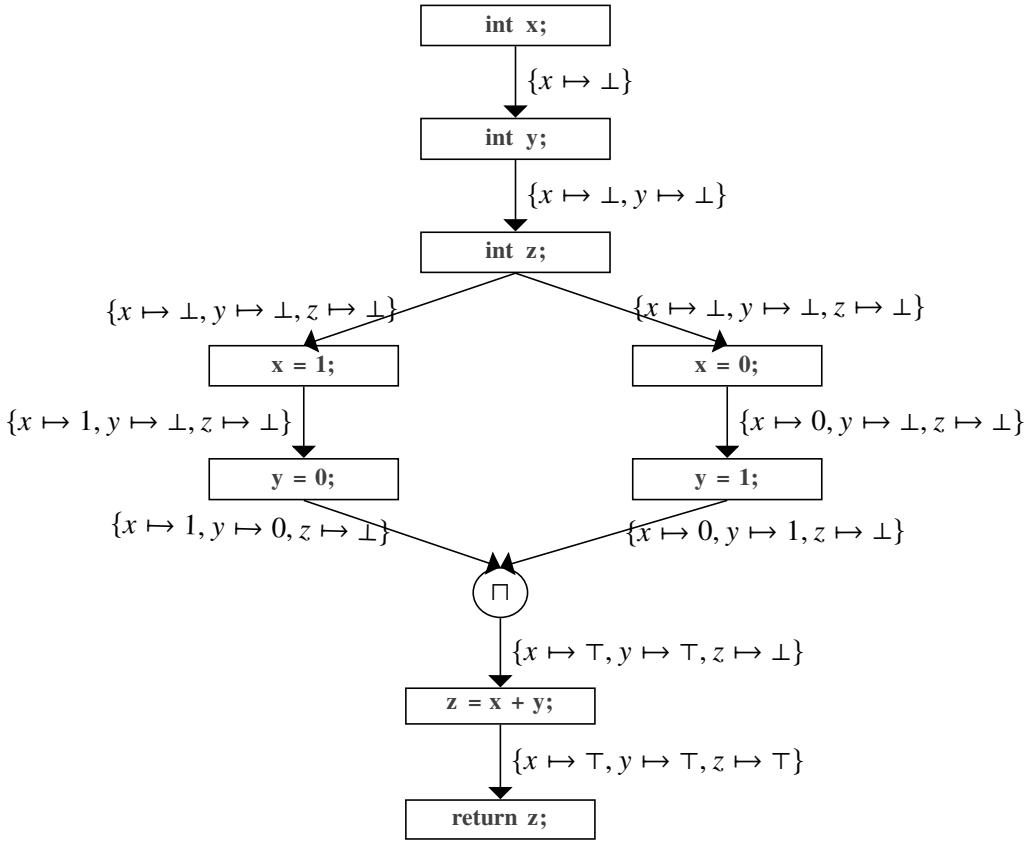
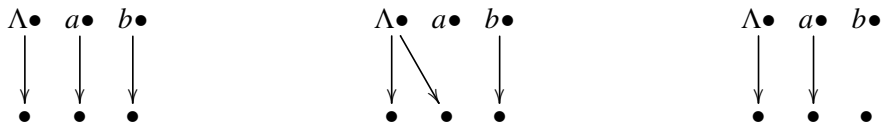


Figure 2.3: Exploded graph for a full-constant propagation conducted on the program shown in Listing 2.1.

2.2 The IFDS and IDE Frameworks

The algorithmic *Interprocedural Finite Distributive Subset* (IFDS) [RHS95] and *Interprocedural Distributive Environments* (IDE) [SRH96] frameworks both follow the *functional* approach [SP78, Bod18] to achieve flow- and fully context-sensitive, inter-procedural data-flow analysis. IFDS and its generalization IDE compute fine-grained, per-fact, re-usable procedure summaries, which allows them to solve data-flow problems efficiently and elegantly.

Both frameworks solve distributive data-flow problems by constructing a so-called *exploded super-graph* (ESG) and solving a graph-reachability problem over that graph. If a data-flow fact d holds at instruction i, the ESG node (i, d) in the ESG is reachable from a special, tautological node Λ. The ESG is constructed for a given target program by replacing every node in its inter-procedural control-flow graph (ICFG) (sometimes referred to as *super-graph*) with the bipartite graph representation of the respective flow function. For distributive data-flow problems, every flow function can be represented as a bipartite graph without loss of precision. Bipartite graphs for the common flow functions *identity*, *gen*

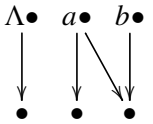


(a) Identity (b) Generate (c) Kill

Figure 2.4: Distributive flow functions and their bipartite graph representations.

$$out(i) = \begin{cases} in(i) \cup \{b\} & \text{if } f \in in(i) \\ in(i) & \text{otherwise} \end{cases}$$

(a) An exemplary flow function. $in(i)$ and $out(i)$ are the sets of data-flow facts that hold before and after instruction i.



(b) The flow function's respective bipartite graph representation.

Figure 2.5: An exemplary flow function and its bipartite graph representation that shows how IFDS/IDE allows one to conditionally generate data-flow facts.

(generate) and *kill* (remove) are presented in Figure 2.4. Hence all *gen/kill* problems such as taint analysis, uninitialized variables, available expressions, etc. can be expressed within IFDS/IDE. But not only those. In particular, IFDS/IDE allow one to *conditionally* generate data-flow facts. The flow function shown in Figure 2.5a, for instance, can be represented by the bipartite graph shown in Figure 2.5b. In this example, every fact is reachable if and only if it was previously reachable, and b is reachable if a was reachable before.

An exemplary ESG for a taint analysis encoded in IFDS that showcases how bipartite graphs can be used to represent flow functions is shown in Figure 2.6. A taint analysis tracks *tainted* variables generated by so-called *source* functions through the program and reports a potential security vulnerability whenever a tainted variable reaches a call to a *sink* function. The function **getPwd** acts as a *source* in our example as it retrieves sensitive user information and the **print** function presents a *sink* as sensitive information must not leak. The taint analysis detects the potential leak at Line 7 in the program since the ESG node $(inst : 7, \mathbf{p})$ is reachable from the tautological Λ fact.

To achieve fully context-sensitive, inter-procedural analysis, IFDS and IDE follow the summary-based approach [SP78] and create procedure summaries that can be reused and instantiated in subsequent calling contexts. Summaries are created by composing the flow functions of adjacent instructions. 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. The graph of h can be produced by merging the nodes of g with the corresponding nodes of the domain of f. Once a summary ψ for a complete procedure p has been constructed,

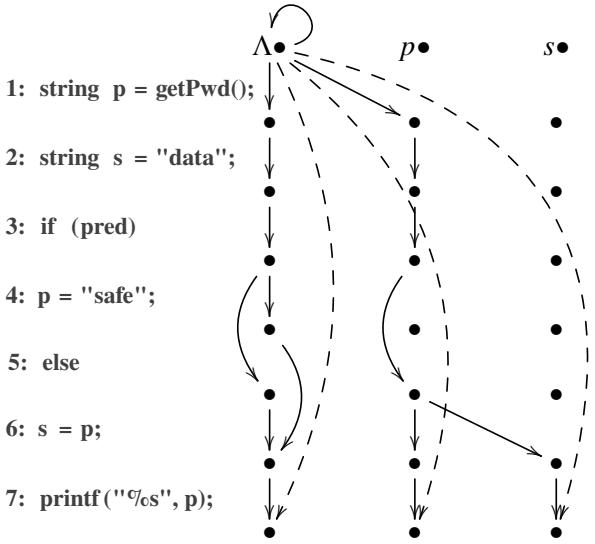


Figure 2.6: An exemplary exploded super-graph for a taint analysis encoded in IFDS [RHS95]. Individual flow functions are indicated with solid edges (→) and flow function summaries (also known as jump functions) are indicated with dashed edges (⇢).

it can be re-applied in any other context in which the procedure p is called. Jump functions are indicated using dashed arrows in Figure 2.6.

In IDE, the generalization of IFDS, the ESG edges carry additional distributive functions. These so-called *edge functions* can be used to describe an additional *value computation problem* over a value domain V that is solved while performing the reachability check on the ESG. The time complexity of both algorithms is $O(|N| \cdot |D|^3)$, where N is the set of program instructions and D is the data-flow domain, i.e., the set of data-flow facts. Importantly, the complexity is independent of V, which allows IDE to conduct efficient computations using large or even infinite value domains (e.g., sets of states of larger state machines in a typestate analysis or the set of natural numbers as required for constant propagation). Attempts to encode such problems in IFDS will lead to state-space explosion or even non-termination. While one can generally encode a linear constant propagation in IFDS using $D = (v, c)$, where $v \in \mathcal{V}$ is the set of program variables and $c \in \mathbb{Z}$, i.e., with tuples of program variables and associated integer values, this encoding drastically impedes performance and without widening, will not even terminate. (Widening is a techniques that causes the analysis to converge on a fixed-point that is a safe approximation.) This is because IFDS was built to solve problems with finite domains but $\mathbb{Z}$ is infinite. Even in cases where one bounds its size artificially, solving performance will be bad. A linear constant propagation can be encoded much more efficiently instead in IDE, using $D = \mathcal{V}$ and $V = \mathbb{Z}$, such as to reduce the size of the data-flow domain and to utilize the edge

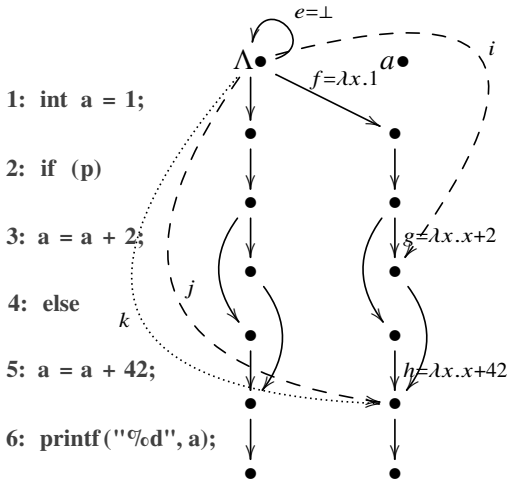


Figure 2.7: An exemplary exploded super-graph for a linear constant analysis encoded in IDE [SRH96]. The ESG highlights the various operations that edge functions must support. Those operations are described, in detail, in Section 4.3.2. Identity edge functions have been omitted to avoid cluttering. Individual flow functions have been indicated by solid edges (⟶) and jump functions have been indicated by dotted (⇢) and dashed (⇢) edges.

functions' value domain V—computing a variable's value using the context-independent edge functions. Since the complexity of IDE's solving algorithm depends only on the size of D and not V and therefore is independent of the infinite size of $\mathbb{Z}$, such an encoding will scale [SRH96]. An exemplary ESG for a linear constant propagation encoded in IDE is shown in Figure 2.7. We explain this ESG in detail in Section 4.3.2.

2.3 Helper Analyses for Precise Whole-Program Data-Flow Analysis

As pointed out in Section 1.1, to solve data-flow problems precisely, a variety of additional information is required that must be computed by static helper analyses on the target program. Even when solving the most basic intra-procedural, flow-sensitive data-flow problems, information on control flows is required to guide the data-flow solver through the target code. Client analyses that require information on inter-procedural data flows require inter-procedural control-flow information, i.e., control-flow and callgraph information.

An overview on the complete analysis stack that is required to solve inter-procedural data-flow problems precisely and its dependencies are shown in Figure 2.8. These dependencies are imposed by many useful client analyses such that we assume them in our work in Chapter 3, Chapter 5, Chapter 6 and Chapter 7.

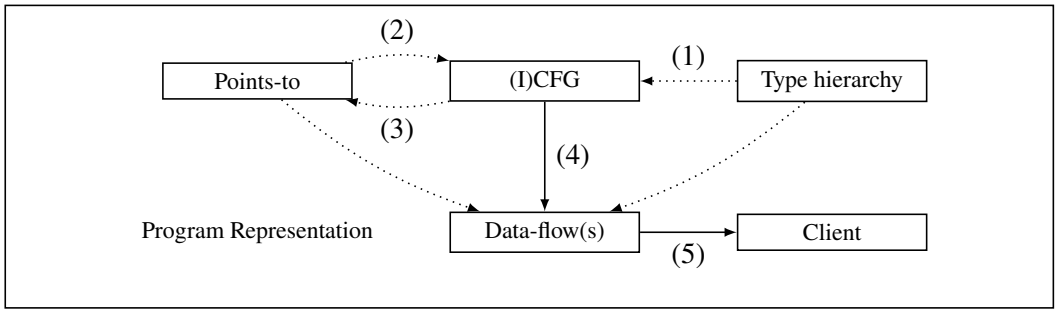


Figure 2.8: Dependency model of a concrete client analysis. Numbered edges indicate in which order information has to be computed on the target program's representation.

The IFDS and IDE solvers are always depending on the target program's inter-procedural control-flow graph (ICFG) that guides the solvers through the program, indicated by a solid edge in Figure 2.8. In addition, they may depend on points-to and type hierarchy information in case variables of pointer types are encountered which is indicated by dotted edges in Figure 2.8. Depending on the desired precision, the ICFG, in turn, may depend on points-to-, type-hierarchy, and virtual-function-table information. A cyclic dependency is introduced due to the fact that precise points-to information also depends on ICFG information [Bod18].

Information on the type hierarchy (and virtual function tables in object oriented languages) have no further dependencies. A client analysis is likely to transitively depend on all of the above information.

Listing 2.2 shows a piece of code that contains a data leak which can be found with help of a taint analysis. The information on control flows, pointers, type hierarchy and data flows that are required to detect the undesired data flow are presented in the following sections.

```
1  #include <iostream>
2
3  [[clang::annotate("psr.source")]] int secret();
4
5  void logInteger([[clang::annotate("psr.sink")]] int i) {
6    std::cout << i << '\n';
7  }
8
9  int identity(int i, int j) { return j; }
10
11 struct A {
12   virtual ~A() = default;
13   virtual int foo() { return 13; }
14 };
15
16 struct B : A {
17   ~B() override = default;
18   int foo() override { return secret(); }
19 };
```

```
20
21 int main(int argc, char **argv) {
22   int i = 0;
23   int j = 0;
24   A *a;
25   A *b;
26   if (argc > 1) {
27     a = new A; // allocation_site_1
28   } else {
29     a = new B; // allocation_site_2
30   }
31   b = a;
32   i = b->foo();
33   j = identity(13, i);
34   logInteger(j);
35   delete a;
36   return 0;
37 }
```

Listing 2.2: An example program that contains a data leak to showcase the various static helper analyses. We will discuss the purpose of the additional attribute specifier sequences ([[]]) in Section 7.3.

2.3.1 Control Flow and Callgraph Information

The inter-procedural control-flow graph of the program presented in Listing 2.2, is shown in Figure 2.9. Besides providing trivial information on intra-procedural control flows, the ICFG also offers callgraph information on inter-procedural caller-callee relations. It connects call sites with starting points of potential call targets as well as a callee's exit instructions with the callee's respective return sites. While the targets of *direct* function calls can be directly read off the code under analysis, an algorithm for callgraph construction has to resolve *indirect* calls to function pointers or virtual function members whose values are dynamically determined at runtime. The result of a callgraph construction algorithm is a set of possible call targets for each indirect call site.

To resolve the virtual call at Line 32 in Listing 2.2, a callgraph algorithm may apply various strategies. In the following, we detail on a few possible approaches with increasing precision.

i A callgraph algorithm can use an (unsound) underapproximation and state that no function is called, effectively ignoring the potential call targets of a call site.

ii It can treat all functions as potential call targets whose signature matches the one at the call site.

iii It can use the type hierarchy and information on virtual function tables to determine that an implementation of `foo` is being called and assume the implementations of the declared type of the receiver object (variable `b`) or any of its subtypes can be called.

iv It can additionally use points-to information to determine possible allocation sites of the pointer (or reference) to the receiver object to determine potential call targets.

2.3.2 Points-to and Alias Information

Points-to information describes to which abstract memory location(s) a pointer-typed variable (or reference-typed variable in case of C++) can potentially point to. Pointer analysis is often a component of more complex analyses and is desired to resolve indirect call sites during callgraph construction and to model loads and stores in data-flow analysis. Alias information answers the question whether an abstract memory location may be accessed in more than one way. Two pointer variables do alias, if they point to the same memory location.

Information on points-to relationships can be maintained using multiple different data structures. The points-to sets for the program in Listing 2.2 at the end of the **main** function look like follows:

```
pts(a) : {obj_1, obj_2}
pts(b) : {a, obj_1, obj_2}
```

However, pointer information can also be stored as graphs. The flow-insensitive pointer assignment graph for the program in Listing 2.2 is shown in Figure 2.11. A pointer assignment graph is constructed with help of the rules stated in Figure 2.10. We will use pointer assignment graphs later on to maintain point-to information in a compositional manner.

2.3.3 Type Hierarchy Information

In static analysis, information on a program's type hierarchy and virtual function tables is required to resolve virtual function calls during callgraph construction. This information can be statically retrieved from the program under analysis and does not require special analysis algorithms.

The type hierarchy and virtual function tables for the program shown in Listing 2.2 are depicted in Figure 2.12. The program comprises two user-defined types that we denote as $\tau_{\mathbf{A}}$ and $\tau_{\mathbf{B}}$. While in programming languages such as Java or C#, all function members are virtual by default and hence potentially emit polymorphic behavior, C++ requires function members to be explicitly marked using the **virtual** keyword to allow subtypes to override their implementations. The virtual function tables for $\tau_{\mathbf{A}}$ and $\tau_{\mathbf{B}}$ both contain two entries: one for their destructor implementation and one for their implementation of **foo**.

2.3.4 Data-Flow Information and Client Analyses

Based on the information provided by the helper analyses described in the previous sections, precise data-flow information can be computed to answer a given client analysis.

While the taint flows required to detect the data leak in the program shown in Listing 2.2 can be computed in an inter-procedural manner using the call-strings approach, it can be

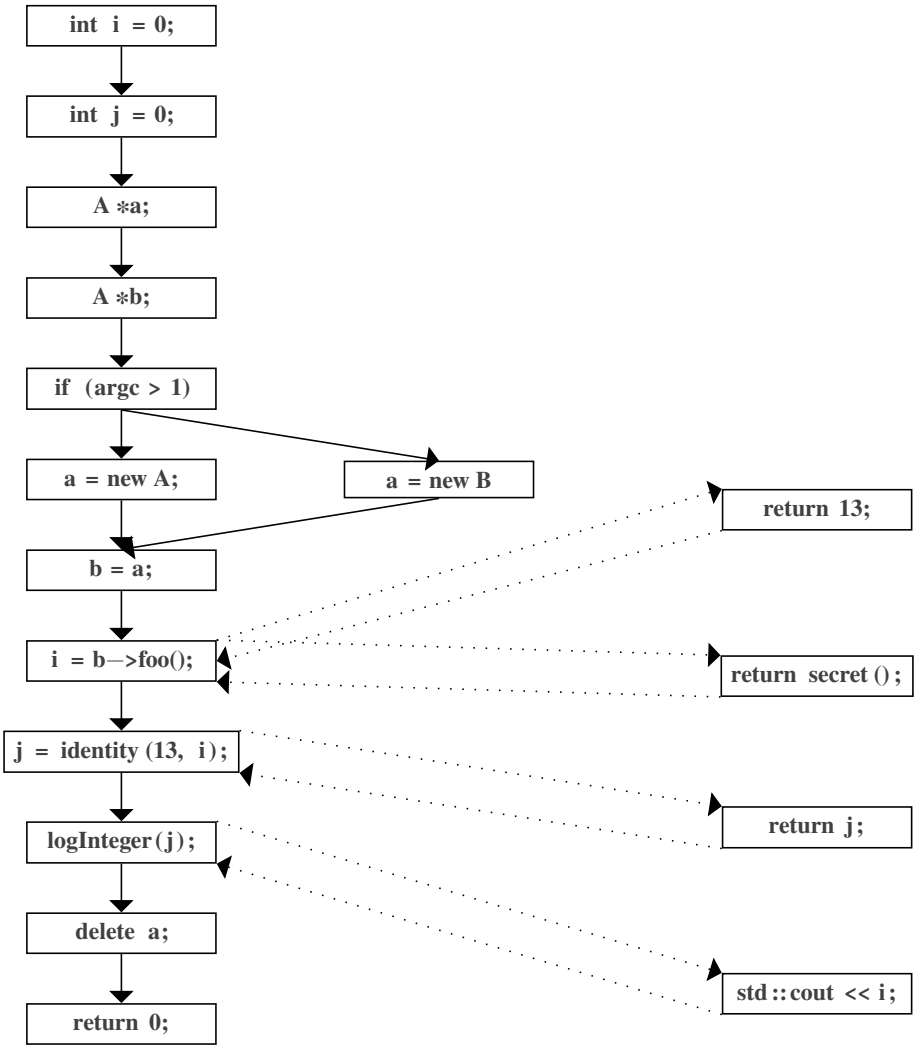


Figure 2.9: Inter-procedural control-flow graph of the program shown in Listing 2.2. Intra-procedural control flows are denoted with solid edges (⟶) and inter-procedural control flows are denoted with dotted edges (⇢).

$$\frac{A1 \rightarrow x}{\{A1\} \subseteq pts(x)}$$

(a) Memory allocation: `x = new Obj; // A1`

$$\frac{y \rightarrow x}{pts(y) \subseteq pts(x)}$$

(b) Assignment: `x = y;`

$$\frac{x \rightarrow y.f,\ o \in pts(y)}{pts(x) \subseteq pts(o.f)}$$

(c) Store: `y.f = x;`

$$\frac{y.f \rightarrow x,\ o \in pts(y)}{pts(o.f) \subseteq pts(x)}$$

(d) Load: `x = y.f;`

Figure 2.10: Rules for the construction of pointer-assignment graphs.

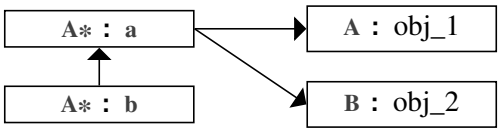


Figure 2.11: Pointer assignment graph of Listing 2.2.

computed much more efficiently using IFDS [RHS95], since taint analysis can be expressed using distributive flow functions.

The parts of the exploded super-graph that are relevant to correctly detect the undesired taint flow are shown in Figure 2.13. The ESG carries the necessary data-flow information to correctly identify the potential leak of sensitive information through the program Lines 32 – 34.

2.4 Soundness and Completeness

Static program analysis can be *sound* or *unsound*. Static analysis that is sound does not "overlook" possible program behavior and instead, overapproximates situations in which it can not compute information precisely.

Our analysis approaches presented in this thesis are *unsound*, since implementing an analysis that computes a more complex property on realistic C and 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 practically unusable [LSS+15]. Instead, our analysis approaches aim at *soundiness* [LSS+15], a well-known term in static analysis. Soundy analyses apply sensible underapproximations to

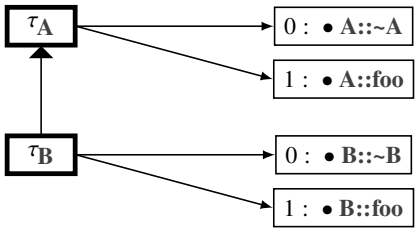


Figure 2.12: Type hierarchy graph and respective virtual function tables of Listing 2.2.

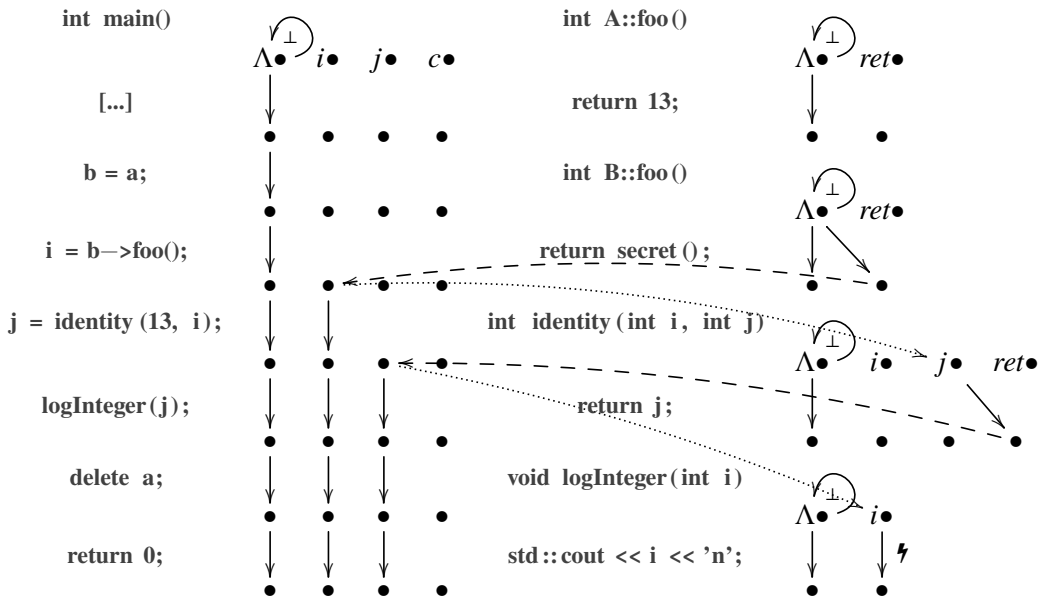


Figure 2.13: Excerpt of an exploded super-graph computed for an IFDS-based taint analysis conducted on the program shown in Listing 2.2. Jump functions and inter-procedural flow functions of Λ have been omitted for brevity. The leak of potentially sensitive information is indicated with the bolt symbol (ϟ).

compute meaningful results in an inter-procedural analysis setting and are widely accepted in the static analysis community [LSS+15, TG17]. 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 virtually all static analyses used for compiler optimization that aim at computing more complex properties are intra-procedural only.

2.5 Precision and Performance

The previous sections show that answering a concrete client analysis not only requires a lot of different analyses but these, in addition, can and must be heavily parameterized, too.

Each of the involved analyses can be typically parameterized to trade off precision versus performance. For instance, limiting the amount of context-sensitivity in the call-strings approach or the depth of field sensitivity are direct trade offs. But also the chosen callgraph and points-to algorithms have a large impact on precision and performance. The situation becomes even more complex, since the analyses oftentimes influence each other; oftentimes in non-trivial manners [Bod18, SLHB21].

Whereas a less precise callgraph is generally faster to compute, the set of potentially call targets at an indirect call site will grow (assuming that the analysis aims at soundness or soundiness, i.e., overapproximates), which, in turn, requires many more propagations of data-flow facts by the subsequent data-flow analysis. Spending more time on computing precise information can actually make the overall client analysis faster, too. This, however, is oftentimes unknown and needs to be evaluated for each analysis setup and target program, given the fact that each target program oftentimes has slightly different characteristics.

2.6 Static Versus Dynamic Analysis

In contrast to static analysis that only "looks" at a given program and emulates an expert analyzing the target code, dynamic program analysis analyzes a given program by executing (parts of) it.

While (sound) static program analysis computes an overapproximation of the possible behavior of the program under analysis, dynamic program analysis computes an under-approximation, since it can only make statements based on the program behavior for the program paths that have been executed. Static program analysis hence often suffers from false positives, whereas dynamic program analysis suffers from false negatives. A diagram that shows the relationship between the actual program behavior and the behavior computed by a static and dynamic analysis is shown in Figure 2.14.

Combining static and dynamic program analysis to create hybrid analysis that enables one to automatically verify static analysis findings by creating appropriate test cases for dynamic validation to cope with false positives in an industrial context is subject of Section 7.3.

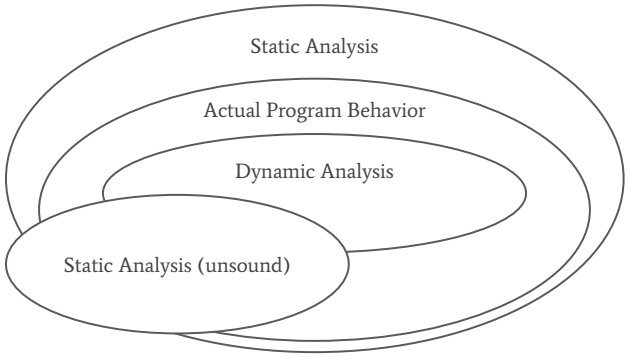


Figure 2.14: Relationship between the actual program behavior and the behaviors computed by static and dynamic program analysis.

2.7 The LLVM Compiler Infrastructure

The LLVM compiler infrastructure [LA04] provides a modular and reusable library and tool chain that contains all necessary parts to build compilers and associated tools. LLVM is a very active open-source project with dozens of related projects. It is also used as the production compiler infrastructure by major companies like Google, Oracle, Apple, Facebook (Meta), Sony and many more [TL:18b].

One essential part of the LLVM infrastructure is its intermediate representation (IR). The LLVM IR is a low-level, typed, three-address, assembly-like language in static single assignment form. Control flows as well as data flows are explicit in the LLVM IR. The IR is—in theory—expressive enough to encode arbitrary source languages. In a typical compiler related workflow, a compiler front-end (i) compiles an input source language into LLVM IR that is then (ii) analyzed and optimized, and finally (iii) a compiler back-end may generate machine code. As we are currently only interested in analyzing the IR, we can ignore step (iii). Clang is LLVM's front-end for C-like languages (including C and C++). Listing 2.3 shows Listing 2.2's corresponding unoptimized LLVM IR produced by the Clang compiler. LLVM IR is the program representation that will be eventually analyzed by the automated analysis approaches presented in this thesis. Although LLVM's IR consists of 65 different instructions and around 315 LLVM intrinsic functions [TL:18a], this small piece of code already shows some of the most important ones.[1]

```
1 %struct.A = type { i32 (...)** }
2 %struct.B = type { %struct.A }
3
4 ; Function Attrs: noinline norecurse optnone uwtable mustprogress
5 define dso_local i32 @main(i32 %argc, i8** %argv) #7 {
6 entry:
7   %retval = alloca i32, align 4
8   %argc.addr = alloca i32, align 4
9   %argv.addr = alloca i8**, align 8
```

[1] As of November 3rd, 2022.

```
10    %i = alloca i32, align 4
11    %j = alloca i32, align 4
12    %a = alloca %struct.A*, align 8
13    %b = alloca %struct.A*, align 8
14     store i32 0, i32* %retval, align 4
15     store i32 %argc, i32* %argc.addr, align 4
16     store i8** %argv, i8*** %argv.addr, align 8
17     store i32 0, i32* %i, align 4
18     store i32 0, i32* %j, align 4
19    %0 = load i32, i32* %argc.addr, align 4
20    %cmp = icmp sgt i32 %0, 1
21     br i1 %cmp, label %if.then, label %if.else
22
23  if.then:      ; preds = %entry
24    %call = call noalias nonnull i8* @_Znwm(i64 8) #11
25    %1 = bitcast i8* %call to %struct.A*
26     call void @_ZN1AC2Ev(%struct.A* nonnull dereferenceable(8) %1) #3
27     store %struct.A* %1, %struct.A** %a, align 8
28     br label %if.end
29
30  if.else:      ; preds = %entry
31    %call1 = call noalias nonnull i8* @_Znwm(i64 8) #11
32    %2 = bitcast i8* %call1 to %struct.B*
33     call void @_ZN1BC2Ev(%struct.B* nonnull dereferenceable(8) %2) #3
34    %3 = bitcast %struct.B* %2 to %struct.A*
35     store %struct.A* %3, %struct.A** %a, align 8
36     br label %if.end
37
38  if.end:       ; preds = %if.else, %if.then
39    %4 = load %struct.A*, %struct.A** %a, align 8
40     store %struct.A* %4, %struct.A** %b, align 8
41    %5 = load %struct.A*, %struct.A** %b, align 8
42    %6 = bitcast %struct.A* %5 to i32 (%struct.A*)***
43    %vtable = load i32 (%struct.A*)**, i32 (%struct.A*)*** %6, align 8
44    %vfn = getelementptr inbounds i32 (%struct.A*)*, i32 (%struct.A*)
          ** %vtable, i64 2
45    %7 = load i32 (%struct.A*)*, i32 (%struct.A*)** %vfn, align 8
46    %call2 = call i32 %7(%struct.A* nonnull dereferenceable(8) %5)
47     store i32 %call2, i32* %i, align 4
48    %8 = load i32, i32* %i, align 4
49    %call3 = call i32 @_Z8identityii(i32 13, i32 %8)
50     store i32 %call3, i32* %j, align 4
51    %9 = load i32, i32* %j, align 4
52     call void @_Z10logIntegeri(i32 %9)
53    %10 = load %struct.A*, %struct.A** %a, align 8
54    %isnull = icmp eq %struct.A* %10, null
55     br i1 %isnull, label %delete.end, label %delete.notnull
56
57  delete.notnull:       ; preds = %if.end
58    %11 = bitcast %struct.A* %10 to void (%struct.A*)***
59    %vtable4 = load void (%struct.A*)**, void (%struct.A*)*** %11,
          align 8
60    %vfn5 = getelementptr inbounds void (%struct.A*)*, void (%struct.A
          *)** %vtable4, i64 1
```

```
61    %12 = load void (%struct.A*)*, void (%struct.A*)** %vfn5, align 8
62    call void %12(%struct.A* nonnull dereferenceable(8) %10) #3
63    br label %delete.end
64
65  delete.end:     ; preds = %delete.notnull, %if.end
66    ret i32 0
67  }
```

Listing 2.3: An excerpt of the LLVM intermediate representation of the program shown in Listing 2.2 produced by the Clang compiler.

Figure 2.15 shows the different scopes or building blocks in which the IR is organized. A **Module** is able to represent a complete C or C++ compilation unit (or multiple linked compilation units). This is what the Clang compiler produces when compiling a single C or C++ file into the LLVM IR. The **Module** contains **GlobalVariable**s and **Function** definitions (or declarations). A **Function**, in turn, consists of one or more **BasicBlock**s which contain one or more **Instruction**s. Each scope of the aforementioned is modeled as a data type within LLVM. One important thing to note is that LLVM follows a highly hierarchical structure. One very important super-type is the LLVM **Value** that, for instance, the **Function**, **BasicBlock** and **Instruction** subtype. Another relevant super-type for static analysis is the **Instruction** type. All of the 65 different LLVM instructions are specializations of this type. A type relation or cast can be checked or performed using LLVM's custom runtime type information (RTTI) system. We need to make use of the RTTI system when writing a concrete data-flow analysis for the LLVM IR in order to inspect the different types of instructions and model their effects on the static information accordingly.

In the following chapter, we present the LLVM-based PhASAR framework that we implemented to account for the lack of static analyzers for C and C++ that implement the concepts we described in the above. PhASAR provides implementations for the theory and concepts described in this chapter. We use PhASAR's infrastructure to implement and evaluate our novel analysis approaches that we present in Chapter 4, Chapter 5, Chapter 6 and Chapter 7.

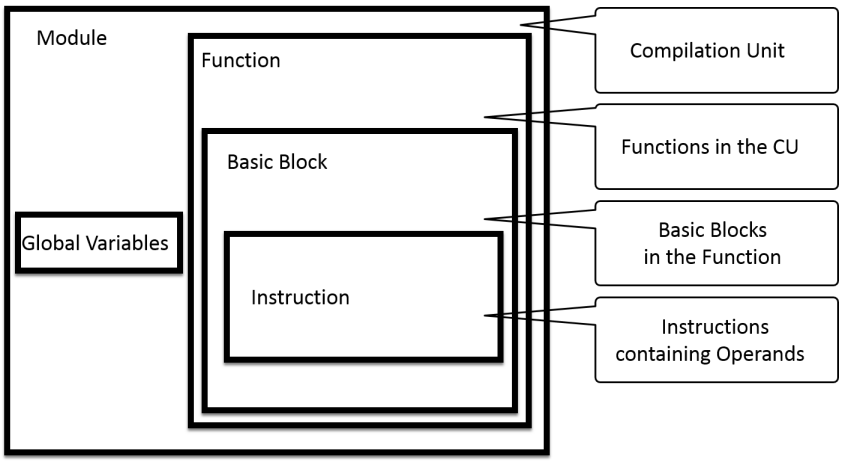


Figure 2.15: Scopes of the LLVM intermediate representation.

3 PhASAR

In this chapter, we describe the design and implementation of our LLVM-based static analysis framework PhASAR. PhASAR allows data-flow problems to be solved in a fully automated manner. It provides class hierarchy, control-flow (including callgraph), points-to, and data-flow information, hence requiring analysis developers only to specify a definition of the data-flow problem to be solved. PhASAR thus hides the complexity of static analysis behind high-level APIs, making static program analysis more accessible and easy to use. PhASAR is entirely written in C++ and available as an open-source project under the permissive MIT license at `https://phasar.org/` and `https://github.com/secure-software-engineering/phasar`.

We evaluate PhASAR's scalability during whole-program analysis. Analyzing 12 real-world programs using a taint analysis written in PhASAR, we found PhASAR's abstractions and their implementations to provide a whole-program analysis that scales well to small and midsize real-world target programs. For target programs that comprise several million lines of code and analyses that use complex data-flow domains, the frameworks starts running into difficulties when conducting such analyses on ordinary consumer hardware.

After presenting PhASAR as our basic infrastructure, we present further improvements of the framework and share our experience that we gained while developing static analyses for C and C++ over the last few years. In the chapters thereafter, we detail on how to scale static analysis to large programs, analysis problems with complex domains and modern software development workflows.

3.1 Introduction

Programming languages from the C family are chosen as the implementation language in a multitude of projects especially in cases where a direct interface with the operating system or hardware components is of importance. Large portions of any operating system and virtual machine (such as the Java VM) are written in C or C++. The reason for this is oftentimes the amount of control the programmer has over many aspects that allow for the creation of very efficient programs—but also comes with the obligation to use these features correctly to avoid introducing bugs or opening the program to security vulnerabilities.

To aid developers in creating correct and secure software, a multitude of checks have been included into compilers such as GCC [GCC18a] and Clang [Cla18a]. Various additional tools such as Cppcheck [Cpp18], clang-tidy [Cla18c], or the Clang Static Analyzer [Cla18b] provide additional means to check for unwanted behavior. Compiler-check passes and additional checkers both use static program analysis to provide warnings to their users. However, to create warnings in a timely fashion, these tools use comparatively simple

analyses that provide either only checks for simple properties, or suffer from a large number of false or missed warnings, due to the imprecision or unsoundness of the used analysis.

For programs written in the Java programming language, program-analysis frameworks like Soot [LBLH11], WALA [Wal19], and Doop [Doo18] are available which allow for a more precise data-flow analysis to determine more intricate program problems. Furthermore, algorithmic frameworks such as *Interprocedural Finite Subset (IFDS)* [RHS95], *Interprocedural Distributive Environments (IDE)* [SRH96], or *Weighted Pushdown Systems (WPDS)* [RSJ03] can be used to describe distributive data-flow problems and efficiently compute their possible solutions.

So far, such implementations have not been openly available for programs written in C or C++. This work thus presents the novel program-analysis framework PhASAR, an extension to the LLVM compiler infrastructure [LA04]. In its inception, we used our experience in developing previous such frameworks for JVM-based languages, namely Soot [LBLH11] and OPAL [EH14], to design a flexible framework that can be adapted to several different types of client analyses. Besides solving data-flow problems, PhASAR can be used to achieve other related goals as well, for instance, callgraph construction, or the computation of points-to information. Its features can be used independently and can be included into other software.

PhASAR is intended to be used as a static analyzer. Therefore, it does not substitute but complement features from the LLVM toolchain and provides also for analyses which during compilation would be prohibitively expensive. While we did not design or implement the analyses included in PhASAR to be a compiler pass, several parts might be used in such a way.

This chapter makes the following contributions:

- It provides a user-centric description of PhASAR's architecture, its infrastructure, and data-flow solvers,
- it presents a case-study that shows PhASAR's overall scalability as well as the precise runtimes of a concrete static analysis, and
- it discusses our experience in developing static analyses for C and C++.

3.2 Related Work

There are several established and well-maintained tools and frameworks for the Java ecosystems. Frameworks from academia include Soot [LBLH11], which is a static analysis framework that allows callgraph construction, computation of points-to information and solving of data-flow problems for Java and Android. Soot does not support inter-procedural data-flow analyses directly. However, a user can solve such problems using the Heros [Bod12] extension that implements an IFDS/IDE solver. The WALA [Wal19] framework provides similar functionalities for Java bytecode, JavaScript and Python. OPAL [EH14] allows for the implementation of abstract interpretations of Java bytecode. Also the manipulation of bytecode is supported. A declarative approach is implemented by the Doop framework [Doo18]. Doop uses a declarative rule set to encode an analysis and solves it using the

logic-based Datalog solver. The framework allows for pointer analysis of Java programs and implements a range of algorithms that can be used for context insensitive, call-site and object sensitive analyses.

Tooling for C and C++ includes Cppcheck [Cpp18] which aims for a result without false positives and allows to encode simple rules as well as the development of more powerful add-ons. The clang-tidy tool [Cla18c] provides built-in checks for style validation, detection of interface misuse as well as bug-finding using simple rules, but can be extended by a user. Checks can be written on preprocessor level using callbacks or on AST level using AST matchers that can be specified using an embedded domain specific language (EDSL). The Clang Static Analyzer [Cla18b] uses symbolic execution and allows custom checks to be written. The SVF [SX16b] framework computes points-to information for constructing sparse value flow and memory static single assignment (SSA). Hence, it can be used for analyses that rely on those information such as memory leak detection or null pointer analysis. Additionally, more precise pointer analysis can be build on top of SVF's results. However, as the computation of memory SSA does require a significant amount of computation, using SVF may not pay off for problems that can be encoded using distributive frameworks, which allow fast, summary-based solutions.

There are also commercial, closed-source tools for static analysis such as CodeSonar [Cod18] and Coverity [CS18], both of which support analyses for C, C++, Java and other languages. Whereas these products are attractive to industry as they provide polished user interfaces, they are not usable for evaluating novel algorithms and ideas in static-analysis research.

3.3 Architecture

As discussed in Chapter 2, precise data-flow analysis requires information from multiple supporting analyses which are typically run earlier, such as class-hierarchy, callgraph, and points-to analysis. Algorithmic frameworks like IFDS provide a generalized algorithm that is then parameterized for each individual data-flow problem. The infrastructure provided by these basic analyses and algorithmic frameworks is necessary to allow analysis designers to efficiently concentrate on the goal of a data-flow analysis. PhASAR is the first framework to provide such infrastructure for programs written in the C language family. Its infrastructure is designed modularly, such that analysis developers can choose the components necessary for their individual goals. Figure 3.1 presents the high-level architecture of the framework.

We allow PhASAR to be used in multiple ways. The first (and easiest) way is through its command-line interface `phasar-llvm`. Its implementation can be seen as a blueprint to create other tools which use PhASAR. The command-line interface provides the means to execute basic analyses such as callgraph construction or pointer analysis or run pre-defined IFDS/IDE-based analyses. The output of these analyses can then be processed using other tooling or presented to the user directly.

Since PhASAR is completely open source and the organization of its source code follows a modular structure, it is open to extension. Users of the framework are free to implement their own points-to, callgraph, data-flow, etc. analysis which they can directly develop as

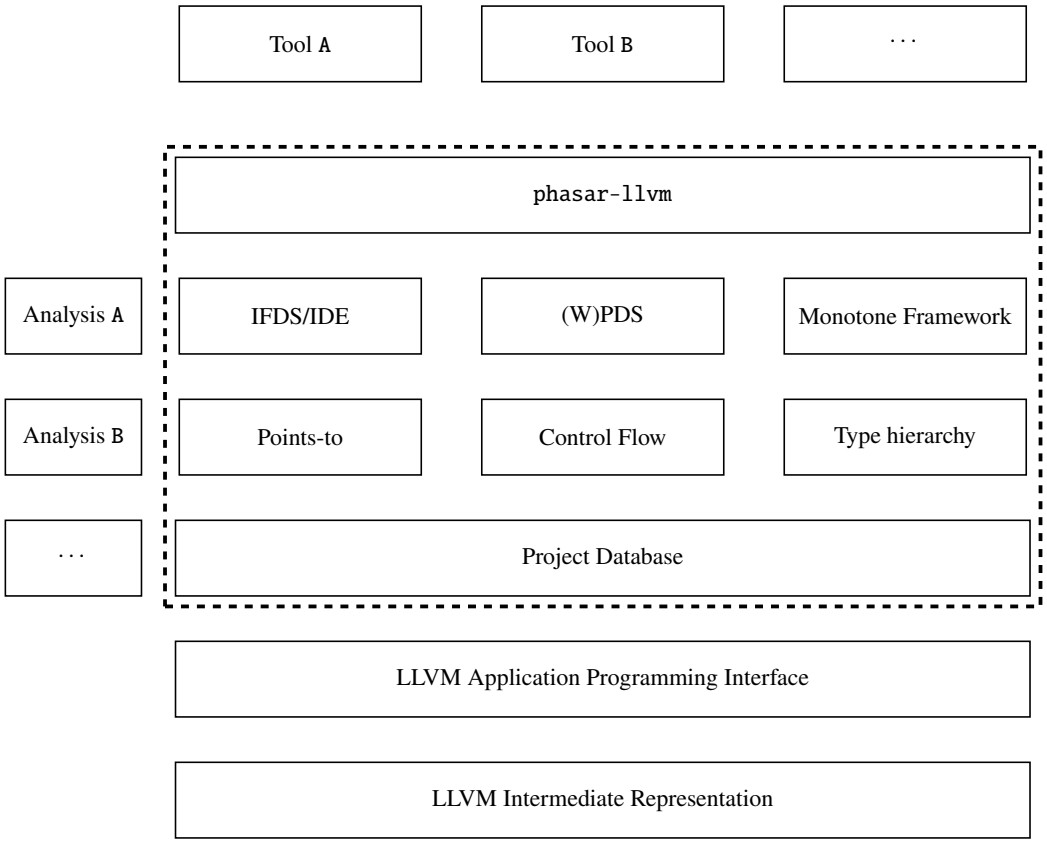


Figure 3.1: PhASAR's high-level architecture

part of the framework and eventually contribute back to the project. Providers of novel analyses need to create an implementation of a pre-defined C++ interface that then offers the respective functionalities in a unified way.

PhASAR can also be included into other tools by using it as a library. This way of using PhASAR provides the most flexibility as developers can freely select the components that should be part of an analysis and can reuse even parts of the components provided by the framework. Thus, the PhASAR framework can be used as a stand-alone application, as a host for custom analysis for various tasks and as a library giving its users full control over the analysis task to accomplish.

PhASAR allows analysis developers to specify arbitrary data-flow problems, which are then solved in a fully-automated manner on the specified LLVM IR target code. Solving a static analysis problem on the IR rather than the source language makes the analysis generally easier. This is because it removes the dependency on the concrete source language, as the IR is usually simpler since the IR involves no nesting and has fewer instructions. Various compiler front-ends for a wide range of languages targeting LLVM IR exist. Hence, PhASAR is able to analyze programs written in languages other than C or C++, too. The framework computes all required information to perform an analysis such as points-to, callgraph, type-hierarchy as well as additional parameterizable taint and typestate analyses.

PhASAR provides various capabilities and interfaces to compute data-flow problems or aid other types of analyses. First, the framework contains interfaces and implementations for the computation of an ICFG; we provide some parameterizable implementations for the LLVM IR.

Next, PhASAR currently supports the computation of function-wise points-to information using LLVM's implementations of the *Andersen*-style [And94] or *Steensgaard*-style [Ste96] algorithms. Points-to information and ICFG computation can be combined to obtain more precise results. We discuss the quality of points-to information and our current efforts to improve their quality in Section 3.9.

To resolve virtual function calls in C++, we provide means to construct a type hierarchy. We construct the type hierarchy for composite types and reconstruct the virtual function tables from the IR, which together with the hierarchy information allow PhASAR to resolve potential call targets at a given call-site.

PhASAR provides implementations of IDE and IFDS solvers as described by Reps et al. [RHS95] including the extensions of Naeem et al. [NLR10]. We implemented IFDS as a specialization of IDE using a binary lattice only using a top and a bottom element much alike the Heros implementation [Bod12]. Both solvers are accompanied by a corresponding interface for problem definition. To solve a data-flow problem using the IDE or IFDS solver, the data-flow problem must be encoded by implementing this interface. We present this in detail in Section 3.4.

For non-distributive data-flow problems PhASAR provides an implementation of the traditional monotone framework [KU77] which allows one to solve intra-procedural problems. The framework provides an inter-procedural version as well that uses a user-specified context in order to differentiate calling-contexts. PhASAR provides a context interface and implementations of this interface that realize the call-strings and value-based approach VASCO [PK13], in which context-sensitivity is achieved by reusing information that has

been computed for previous calls under the same context. The framework also implements a version of the context class to represent a *null context*. This context has the same effect as applying the monotone framework directly in an inter-procedural setting. Both solvers are accompanied by corresponding interfaces for problem descriptions which must be implemented to encode a concrete data-flow problem. The details are provided in Section 3.4.

All of PhASAR's data-flow solvers are implemented in a fully generic manner and heavily make use of templates and interfaces. For instance, a solver follows a target program's control-flow that is specified through an implementation of either the `CFG` or the `ICFG` interface. Analysis developers can parameterize a solver with an existing implementation or they can provide their own custom implementation. They can run a forward or backward analysis depending on the direction of the chosen control-flow graph. Moreover, all data-flow related functionality is hidden behind interfaces. A solver queries the required functionality such as flow functions or merge operations for the underlying lattice whenever necessary. We have specified problem interfaces on which the corresponding solver operates. Thus, analysis developers encode their data-flow problem by providing an implementation for the problem interface and provide this implementation to the accompanying solver. PhASAR is able to solve a problem on other IRs when suitable implementations for the IR specific parts such as the control-flow graphs and problem descriptions are provided by the analysis developer.

3.4 PhASAR's Implementation

Our goal with PhASAR is easing the formulation of a data-flow analysis such that an analysis developer only needs to focus on the implementation of the problem description rather than providing details how the problem is solved.

PhASAR achieves parts of its generalizability through template parameters. These template parameters include, among others, `N`, `D`, `F`. They are consistently used throughout the implementation of PhASAR. `N` denotes the type of a node in the ICFG, i.e., typically an IR statement, `D` denotes the domain of the data-flow facts, and `F` is a placeholder for the type of a method/function. When analyzing LLVM IR, `N` is always of type `const llvm::Instruction*` and `F` is of type `const llvm::Function*`, whereas `D` depends on the specific data-flow analysis that the developer wants to encode. For our example using linear constant propagation described in Chapter 2, `D = std::pair<const llvm::Value *, int>` could be used to capture the property of interest. LLVM's `Value` type is quite useful as it is a super-type that is located high in the type hierarchy. This allows an analysis developer to use values of all of `Value`'s subtypes in the value domain, which makes it highly flexible.

3.4.1 Encoding an IFDS Analysis

Listing 3.1 shows the interface for an IFDS problem. An analysis developer has to define a new type—the problem description—implementing the `FlowFunctions` interface.

```
1 template <typename N, typename D, typename F> struct FlowFunctions {
2   virtual ~FlowFunctions() = default;
```

```
3     virtual FlowFunction<D> *getNormalFlowFunction(N curr, N succ) =
          0;
4     virtual FlowFunction<D> *getCallFlowFunction(N callStmt,
5                                                  F destMthd) = 0;
6     virtual FlowFunction<D> *getRetFlowFunction(N callSite,
7                                                 F calleeMthd,
8                                                 N exitStmt,
9                                                 N retSite) = 0;
10    virtual FlowFunction<D> *
11    getCallToRetFlowFunction(N callSite, N retSite,
12                             std::set<F> callees) = 0;
13  };
```

Listing 3.1: Interface for specifying flow functions in IFDS/IDE.

The flow function factories shown in Listing 3.1 handle the different types of flows. The four factory functions each have an individual purpose:

- **getNormalFlowFunction** handles all intra-procedural flows.
- **getCallFlowFunction** handles inter-procedural flows at a call-site. Usually, the task of this flow function factory is to map the data-flow facts that hold at a given call-site into the callee method's scope.
- **getRetFlowFunction** handles inter-procedural flows at an exit statement (e.g. a return statement). This maps the callee's return value, as well as data-flow facts that may leave the function by reference or pointer parameters, back into the caller's context/scope.
- **getCallToRetFlowFunction** propagates all data-flow facts that are not involved in a call along-side the call-site, typically stack-local data not referenced by parameters.

These flow function factories are automatically queried by the solver, based on the inter-procedural control-flow graph.

The functions in Listing 3.1 are factories since they have to return small function objects of type **FlowFunction** which is shown in Listing 3.2. As a **FlowFunction** is itself an interface, an analysis developer has to provide a suitable implementation. The member function **computeTargets** takes a value of a dataflow fact of type **D** and computes a set of new dataflow facts of the same type. The solver automatically queries the respective flow function factory for each statement and then calls the **computeTargets** implementation of the flow function on each data-flow fact that holds before the instruction i under analysis to determine the flow facts that hold after instruction i. It thus specifies how the bipartite graph for the instruction that represents the flow function is constructed and can be thought of an answer to the question "What edges must be drawn?".

```
1  template <typename D> struct FlowFunction {
2    virtual ~FlowFunction() = default;
3    virtual std::set<D> computeTargets(D source) = 0;
4  };
```

Listing 3.2: Interface for a flow function in IFDS/IDE

As flow function implementations often follow certain patterns, we provide implementations for the most common patterns as template classes. Many useful flow functions like **Gen**, **GenIf**, **Kill**, **KillAll**, and **Identity** are already implemented and can be directly used. Any number of flow functions can be easily combined using our implementations of the **Compose** and **Union** flow functions. We also provide **MapFactsToCallee** and **MapFactsToCaller** flow functions that automatically map parameters into a callee and back to a caller, since this behavior is frequently desired. Flow functions which are stateless, e.g. **Identity** or **KillAll**, are implemented as a thread-safe singleton.

3.4.2 Encoding an IDE Analysis

If an analysis developer wishes to encode their problem within IDE, they have to additionally provide implementations for the edge functions. With help of the edge functions, an analysis developer is able to specify a computation which is performed along the edges of the exploded super-graph leading to the queried node (cf. Figure 2.7). The interface for the edge function factories and their responsibilities are analogous to the flow function factories in Listing 3.1.

Each edge function factory must return an edge function implementation: a small function object similar to a flow function which has a **computeTarget** function, a compose, a merge, and an equality-check operation. The **EdgeFunction** interface is shown in Listing 3.3.

```
1  template <typename V> class EdgeFunction {
2  public:
3    virtual ~EdgeFunction() = default;
4    virtual V computeTarget(V source) = 0;
5    virtual EdgeFunction<V> *
6    composeWith(EdgeFunction<V> *secondFunction) = 0;
7    virtual EdgeFunction<V> *
8    joinWith(EdgeFunction<V> *otherFunction) = 0;
9    virtual bool equal_to(EdgeFunction<V> *other) const = 0;
10 };
```

Listing 3.3: Interface for an edge function in IDE

As this interface is more complex than the flow function interface, we explain the purpose of each function. The **computeTarget** function describes a computation over the value domain V in terms of lambda calculus for an edge of the ESG.

The **composeWith** function encodes how to compose two edge functions. In most scenarios, this function can be implemented as $(f \circ g)(x) = f(g(x))$ by applying **computeTarget** of one edge function to **source** and the **computeTarget** function of the other on the result. To avoid additional boilerplate code, we provide an **EdgeFunctionComposer** class that performs this job and can be used as a super class.

joinWith encodes how to join two edge functions at instructions where two control-flow edges lead to the same successor instruction. Depending if a may or a must-analysis is performed, implementations of this function typically check which edge function computes a value that is higher up in the lattice, i.e., a more approximate value, and returns the corresponding edge function. For our linear constant propagation, for instance, this function

would return one of the edge functions if both describe the same value computation, the bottom edge function if both of them encode the $\perp$ value and the edge function encoding the top element otherwise. The intuition here is to always pick the element that is higher in the lattice as it represents more information.

The **equal_to** interface function has to be implemented to return true if both edge functions describe the same value computation, false otherwise.

A complete implementation of a linear constant propagation encoded within IDE can be found along with PhASAR's other examples at our website [Pha18].

3.4.3 Encoding an Analysis Within the Monotone Framework

If an analysis developer wishes to encode a problem that does not satisfy the distributivity property, they have to make use of the monotone-framework implementation or its inter-procedural variant. An excerpt of the interface for specifying an inter-procedural monotone problem is shown in Listing 3.4. Similar to an IFDS/IDE problem, an analysis developer has to specify flow functions for intra- and inter-procedural flows. But in contrast to IFDS/IDE, these flow functions do not operate on single, distributive data-flow facts, but on sets of data-flow facts instead. The data-flow facts in fact do not even need to be stored as sets but the concrete container implementation is parameterizable using a template parameter. A user thus can not only have the data-flow solver propagate *sets* of data-flow facts, but other kinds of containers such as maps, etc., too. The solver calls the flow functions and provides the container of data-flow facts which hold right before the current instruction. The return value to be computed in the flow function is a container of data-flow facts that hold after the effects of the current statement. The **merge** function specifies how information is merged when two branches lead to a common successor statement. This is typically implemented as set-union or set-intersection depending on whether a may or must-analysis has to be solved. Algorithms from C++'s STL may be used here. Finally, the **equal_to** function must be implemented to determine if two containers hold the same amount of information to check if a fixed-point is reached. The context that is used for the inter-procedural analysis can be specified by the analysis developer using the template parameter. An analysis developer can provide a pre-defined context class in order to parameterize the analysis to be a call-strings approach, a value-based approach, or they can define their own context to be used.

```
1  template <typename N, typename D, typename F>
2  struct InterMonotoneProblem {
3    virtual ~InterMonotoneProblem() = default;
4    virtual std::set<D> merge(const std::set<D> &Lhs,
5                              const std::set<D> &Rhs) = 0;
6    virtual bool equal_to(const std::set<D> &Lhs,
7                          const std::set<D> &Rhs) = 0;
8    virtual std::set<D> normalFlow(N Stmt, const std::set<D> &In) = 0;
9    virtual std::set<D> callFlow(N CallSite, F Callee,
10                                const std::set<D> &In) = 0;
11   virtual std::set<D> returnFlow(N CallSite, F Callee, N RetStmt,
12                                  N RetSite,
13                                  const std::set<D> &In) = 0;
14   virtual std::set<D> callToRetFlow(N CallSite, N RetSite,
```

```
15                                          const std::set<D> &In) = 0;
16 };
```

Listing 3.4: An excepert of the interface for describing an inter-procedural problem for the monotone framework.

3.4.4 Use PhASAR as a Library

Listing 3.5 demonstrates the use case of using PhASAR as a library. It shows that only a few lines of code are necessary to set up and run a custom data-flow analysis. The burden of providing the complete infrastructure such as class hierarchy, points-to, callgraph information as well as various data-flow solvers is done by PhASAR.

```
1  int main(int argc, const char **argv) {
2    if (argc < 2  !std::filesystem::exists(argv[1])
3        std::filesystem::is_directory(argv[1])) {
4      llvm::errs() << "A small PhASAR-based example program\n\n"
5                   "Usage: myphasartool <LLVM IR file>\n";
6      return 1;
7    }
8    psr::ProjectIRDB DB({argv[1]});
9    std::string EntryPoint = "main";
10   const auto *F = DB.getFunctionDefinition(EntryPoint);
11   if (!F) {
12     llvm::errs()
13         << "error: file does not contain a 'main' function!\n";
14     return 0;
15   }
16   psr::LLVMTypeHierarchy H(DB);
17   psr::LLVMPointsToSet P(DB);
18   psr::LLVMBasedICFG I(DB, CallGraphAnalysisType::OTF, {EntryPoint},
19                        &H, &P);
20   llvm::outs() << "Running an IDE-based linear-constant analysis:\n"
         ;
21   psr::IDELinearConstantAnalysis L(&DB, &H, &I, &P, {EntryPoint});
22   psr::IDESolver S(L);
23   S.solve();
24   S.dumpResults();
25   return 0;
26 }
```

Listing 3.5: Setting up an analysis run to solve a concrete data-flow analysis.

3.4.5 Handling of Intrinsic Functions and `libc`

LLVM currently has approximately 315 intrinsic functions. These functions are used to describe semantics in the analysis and optimization phase and do not have an actual implementation. Later-on in the compiler pipeline, the back-end is free to replace a call to an intrinsic function with a software or a hardware implementation—if one exists for the target architecture. Introducing new intrinsic functions is preferred over introducing

novel instructions to LLVM since, when introducing a new instruction, all optimizations, analyses, and tools built on top of LLVM have to be revisited to make them aware of the new instruction. A call to an intrinsic function can be handled as an ordinary function call.

The functions contained in the `libc` standard library represent special targets as well as these functions are used by virtually all practical C and C++[1] programs. Moreover, the functions contained in the standard library cannot be analyzed themselves as they are mostly very thin wrappers around system calls and are often not available for the analysis. In many cases, however, it is not necessary to analyze these functions when performing a data-flow analysis. PhASAR models all of them as the identity function. An analysis developer can change the default behavior and model different effects by using special summary functions. The **SpecialSummaries** class can be used to register flow and edge functions other than identity. This class is aware of all intrinsic and `libc` functions.

3.4.6 A Note on PhASAR's Soundness

Livshits et al. have introduced the notion *soundy* analyses [LSS+15] as presented in Section 2.4. Soundy analyses use sensible underapproximations to cope with certain language features that would otherwise make an analysis impractically imprecise. Analyses in PhASAR are currently *soundy*. For instance, PhASAR's ICFG misses one control-flow edge in the presence of **setjmp/longjmp**. Functions that are loaded dynamically from shared object libraries using **dlsym** can not be analyzed either for obvious reasons. PhASAR's data-flow solvers treat calls to dynamically loaded libraries and libraries for which function definitions are missing using the identity transformation, unless the analysis developer specifies otherwise. A sound handling would require setting all variables involved in such calls to the most coarse grain element of the underlying lattice, which again, may lead to so much imprecision that the analysis results become unusable.

3.5 Scalability

In this section, we present the runtime measurements for two concrete static analyses—**IFDS−SolverTest** we name $\mathbb{I}$ and **IFDSTaintAnalysis** we name $\mathbb{T}$—that are both implemented in PhASAR. $\mathbb{I}$ is a trivial IFDS analysis which passes the tautological data-flow fact Λ through the program. The analysis acts as a baseline as it is the most efficient IFDS/IDE analysis that can possibly be implemented. $\mathbb{T}$ implements a taint analysis. A taint analysis tracks values that have been tainted by one or more sources through the program and reports whenever one of the tainted values reaches a sink, which can be functions or instructions. Our taint analysis treats the command-line parameters **argc** and **argv** that are passed into the **main** function as tainted. Functions that read values from the outside (e.g. **fread**) are interpreted as sources. Functions that can leak tainted variables to the outside such as **printf** or **fwrite** are considered sinks. As a potentially large amount of tainted values have to be tracked through the program, analysis $\mathbb{T}$ will provide insights into the scalability of PhASAR's IFDS/IDE solver implementation.

[1] The compiler translates many of C++'s features into ordinary calls to `libc`.

Table 3.1 shows the programs that we analyzed. For each program, the IR's lines of code, number of statements, pointers, and allocation sites have been measured with PhASAR. The LLVM IR has been compiled with the Clang compiler using production flags. The figures give an intuition for the program's complexity. The programs that we analyzed comprise some C programs like some of the coreutils [Cor18] as well as two C++ programs like PhASAR itself and a PhASAR-based tool MPT. In addition, it shows the runtimes of the analyses $\mathbb{I}$ and $\mathbb{T}$ separated into different phases (in the format runtime $\mathbb{I}$/runtime $\mathbb{T}$). We measured the runtimes for the construction of points-to information (PT), class hierarchy (CH), callgraph (CG), data-flow information (DF), and the total runtime (Σ). We also measured the number of function summaries $\psi(f)$ that could be reused while solving the analysis. The latter one is a good indicator for the quality of the data-flow domain D, as higher reuse indicates a more efficient analysis. #G and #K denote the number of facts that have been generated or killed in the taint analysis, respectively.

We measured the runtimes by performing 15 runs for each analysis on a virtual machine running on an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz machine with 128GB memory. We removed the minimum and maximum values and computed the average of the remaining 13 values for each of the four analysis steps and the total runtime. We used an on-the-fly callgraph algorithm that uses points-to information for the coreutils. For PhASAR and MPT, we used a declared type-analysis (DTA) callgraph algorithm in order to reduce the amount of memory required to reproduce our results. In addition, we found that DTA performed well enough on our C++ target programs.

With one exception, PhASAR is able to analyze a program from coreutils within a few seconds. Analyzing cp using $\mathbb{T}$ takes around 13 minutes. This is because a large amount of facts is generated which must then be propagated by the solver. This result shows the cubic impact of the number of data-flow facts on IFDS/IDE's complexity. Analyzing the million-line programs PhASAR and MPT ranges from 7 to 18 minutes. As one can observe for PhASAR, an analysis may destroy data-flow facts more often than it generates them. This is caused by C++'s exceptional control-flow where the same fact is destroyed during normal and exceptional flow.

We observed that the DF part of $\mathbb{T}$ actually runs faster than $\mathbb{I}$ for our C++ target programs. This is because $\mathbb{T}$ should behave very similar to the solvertest for the C++ target programs, as only very few facts are actually generated. Furthermore, $\mathbb{T}$ will take shortcuts whenever it plugs in the desired effects at call-sites of source and sink functions. $\mathbb{I}$ in contrast, follows these calls making it slower than $\mathbb{T}$.

Analyzing all of the 97 coreutils, PhASAR, and MPT requires a total analysis time of of 30 minutes for $\mathbb{I}$ and 1 hour and 31 minutes for $\mathbb{T}$. These measurements show that PhASAR is capable of analyzing even a million-line program within minutes, even though PhASAR's algorithms and data structures have not yet undergone manual optimization.

3.6 The Need for Dedicated Debugging Capabilities

In this section, we elaborate on the difficulties of implementing static program analyses and the essential need for debugging capabilities within static analysis frameworks.

Table 3.1: Program's characteristics and performance figures for analyses $\mathbb{I}/\mathbb{T}$

Program	kLOC	Stmts	Ptrs	Allocs	CH [ms]	PT [s]	CG [s]	DF [s]	Σ [s]	$\#\psi(f)$	#G	#K
wc	132	63166	10644	396	24/24	1.0/1.0	0.1/0.1	0.2/11	2/13	119/125	10202	6830
ls	152	71712	13200	438	27/27	1.4/1.4	1.1/1.2	0.6/1.0	4/5	836/839	79	74
cat	130	62588	10584	391	24/24	1.0/1.0	0.0/0.0	0.1/1.3	2/3	21/22	2525	1262
cp	141	67097	11722	443	32/30	1.3/1.3	0.6/0.6	0.4/789	3/792	547/737	16999	12839
whoami	129	61860	10433	389	24/23	1.0/1.0	0.0/0.0	0.1/0.3	2/2	8/11	97	92
dd	137	65287	11150	408	25/25	1.1/1.0	0.2/0.2	0.2/37	2/40	164/176	14711	11058
fold	130	62201	10509	390	24/23	1.0/1.0	0.0/0.0	0.1/0.3	2/2	17/22	107	102
join	134	64196	11042	402	24/24	1.0/1.0	0.0/0.0	0.1/0.5	2/3	91/95	104	94
kill	130	62304	10527	394	24/24	1.0/1.0	0.0/0.0	0.1/0.1	2/2	24/24	22	4
uniq	131	62663	10650	396	24/24	1.0/1.0	0.0/0.0	0.1/0.4	2/2	50/53	96	90
MPT	3514	1351735	755567	176540	906/903	22/22	8.8/8.8	458/379	519/439	12531/12532	20	9
PhASAR	3554	1368297	763796	178486	962/946	23/23	24/24	987/917	1064/993	25778/25782	56	77

The development of a high-quality data-flow analysis—one that is precise and scalable—is a challenging task. A concrete client analysis not only requires data-flow but, as discussed in Chapter 1 and Chapter 2, type-hierarchy, points-to, and callgraph information, all of which need to be obtained by wisely chosen and correctly parameterized algorithms.

Therefore, many static analysis frameworks have been developed that provide analysis writers with generic data-flow solvers as well as those additional pieces of information. Such frameworks ease the development of an analysis by requiring only a description of the data-flow problem to be solved and a set of framework parameters.

Yet, analysis writers[2] often struggle when an analysis does not behave as expected on real-world code. It is usually not apparent what causes a failure due to the complex interplay of the various different algorithms and the client analysis code within such frameworks.

This section thus presents some of the insights we gained by instrumenting PhASAR and shows the broad area of applications at which flexible instrumentation supports analysis and framework developers.

We present five cases in which instrumentation gave us valuable insights to debug and improve both, the concrete analyses and the underlying PhASAR framework.

3.6.1 Instrumenting Static Analysis

There are several reasons why the development of a precise and scalable data-flow analysis is difficult. Concrete client analyses often need additional helper analyses to provide them with type-hierarchy, points-to, and callgraph information in order to provide precise results [Bod18].

However, writing a client analysis and all required helper analyses from scratch is impractical. For this reason, many different static analysis frameworks have been developed to ease that process. Frameworks from academia, among others, include Soot [LBLH11], Doop [BS09], Wala [Wal19], OPAL [EH14], Soufflé [JSS16], and PhASAR [SHB19]. Those frameworks provide implementations for the helper analyses and generic data-flow

[2]This includes the author of this thesis.

solvers that are able to solve a given user-specified data-flow problem in a fully automated manner. Thus, an analysis writer can focus on specifying the actual analysis problem.

Encoding an analysis is still challenging as an analysis developer has to perform a tremendous amount of complex tasks. Encoding an analysis in a general purpose language, for instance, as required by frameworks such as Soot, Wala, or PhASAR still requires an analysis developer to write several hundred to thousand lines of code that comprise the problem description [TG17]. Choosing the parameters for their analysis is also non-trivial as the parameters have to be chosen according to an analysis's requirements and the target program's characteristics to trade off precision and scalability.

The complexity further increases for frameworks that use analyses encoded in general purpose languages as they often use distributive frameworks like (IFDS) [RHS95], IDE [SRH96], or *Weighted Pushdown Systems* (WPDS) [RSJ03] to achieve a decent scalability [Bod18]. Those data-flow frameworks, in turn, solve a problem in a multi-step process. In general, an internal representation, e.g. exploded super-graph or pushdown system, is constructed first and then, the problem is solved on that representation in a second step. A buggy analysis might withstand the construction of the internal representation, but still fail the actual solving process.

Eventually, an analysis developer has encoded their analysis successfully with respect to a micro-benchmark that has been used to develop it (c.f. Section 3.6.2). Applying the analysis to real-world software, however, they will frequently observe their analysis to fail [TG17]. The reasons for such a failure can be manifold and oftentimes hide in the complex interplay of the involved algorithms and the complex nature of the analysis description.

Debugging analysis failures is non-trivial as it requires knowledge of algorithms, solvers, executing system, target programs, and intermediate representations. Detecting the cause of the failure with help of a standard debugger is usually a tedious to impossible task as the analysis developer needs to debug through large amounts of non-analysis code; it might be not helpful at all if the analysis is correct but the executing operating system causes an analysis run to fail, e.g. if the system runs out of memory. The work of Nguyen et al. [NKH+18] presents a special debugger for static analysis which shows the severity of the problem. The use of logging techniques produces log files that are too large to effectively debug failure on real-world code or slows down the analysis execution to a point that is not acceptable.

In this section, we show how a flexible instrumentation of a static analysis framework is able to aid the understanding of concrete analysis runs. Using an instrumentation in combination with a post-processing step of the recorded data allows to spot anomalies and root causes of analysis failures that would otherwise remain hard to detect when analyzing real-world software. In addition, an extensive instrumentation allows for detailed performance benchmarks which, in turn, allow for spotting bottle-necks and fine-tuning an analysis. Different algorithms can be assessed based on their performance on the target code and a framework user is able to precisely determine how much time of an analysis run is spend in which parts of a framework.

In summary, this section makes the following contributions:

- It presents our highly flexible instrumentation of the PhASAR framework [SHB19] called *PAMM*,
- and an experience report that presents five cases in which the instrumentation provided us with valuable insights into concrete analysis runs that we used for debugging and optimizations.

3.6.2 Analysis Development Process

In this section, we briefly explain a commonly used process to develop a client data-flow analysis.

A static analysis framework such as PhASAR provides the required infrastructure such as data structures, algorithms, and analysis pipelines with the goal to ease the process of developing a concrete client analysis. Figure 3.2 shows an overview of how an analysis framework is used to run an analysis. The gray boxes indicate the parts that require work from the analysis writer.

The most labor-intensive task involved in this process is crafting the description of the analysis problem. Depending on the static analysis framework that has been chosen, a developer needs to implement flow functions or specify rules in order to model the interaction of a program's statements with the data-flow facts that the developer is interested in.

The creation of an analysis description is an incremental process. In order to evaluate the correctness and the level of precision, a developer starts specifying their analysis to handle the basic language features and tests it on small example programs. The results reported by the analysis on the example programs are checked and compared to the expected results. Once the quality of the results suffices for the initial example programs, some more advanced example programs are written. These example programs form a micro benchmark that allows a developer to evaluate the quality and completeness of their analysis description. The example programs, that act as test cases, and the analysis code are alternately enhanced until the analysis is able to cope with all common language features and obtains the desired precision. Several micro benchmarks such as DroidBench [Dro19], SecuriBench [Sec19], DaCapo [BGH+06], or the Toyota ITC benchmark [SMM15] have been established to evaluate the quality of an analysis which shows that the development process described here is common practice.

When the complexity of the programs of the micro benchmark has risen to a certain level, another part of the development process becomes relevant: the framework's parametrization. Many frameworks allow for the construction of the type hierarchy, callgraph and points-to information which become necessary depending on the complexity of the test programs and the desired precision of the analysis when eventually run on real-world software. For instance, for the construction of callgraph and points-to information developers can choose from a variety of algorithms such as CHA, DTA, VTA, Spark for callgraphs, or Andersen [And94] or Steensgard [Ste96]-style algorithms for points-to information. Computations can be chosen to be performed in a full analysis mode or an on-demand manner. Finding *the best* or at least suitable parameters, however, is challenging. While heavyweight

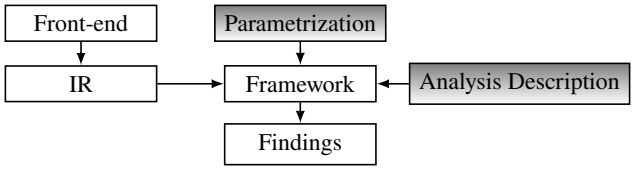


Figure 3.2: A typical pipeline of a static analysis framework

algorithms may produce precise results on the small test programs, they are oftentimes too slow to be used on larger real-world code. Finding the optimal point between scalability and precision is key and an ongoing challenge [Bod18, LTMS18, BBC+10].

3.6.3 Implementation

While designing PAMM we opt for a ready-to-use mechanism to collect different measures related to static analysis. Three basic types of measures turned out to be useful in practice: timer, counter, and histogram. We provide code to start, pause, stop and reset different timers, increase and decrease counters by a given value, and add data points to histograms. All measures used are identified by user-specified IDs and must be registered before use. This allows us to detect and minimize exceptional measurements caused by flawed code instrumentations, e.g. a misspelled ID. We implemented PAMM as a singleton to minimize boilerplate for the construction and destruction of PAMM. Each instrumentation instruction is wrapped into a corresponding preprocessor macro to hide implementation details. This also allows a user to disable PAMM without removing any code instrumentation manually, and thus, guaranteeing zero overhead during non-evaluation runs of PhASAR. However, recompilation is necessary to enable or disable PAMM.

Since code instrumentation is tedious and oftentimes requires a profound knowledge of PhASAR's internal structure, we provide a default instrumentation for all parts of PhASAR relevant to static analysis. Multiple measures can be grouped which allows a user to only collect the data of analysis runs that they are currently interested in. A user is able to instrument their own analysis code and register their instrumentation in a new group to record their client measures without using the default (*full*) framework instrumentation. Our instrumentation of the *core* group, for instance, comprises, among other measures, runtime information for each step of an analysis run and statistics of the analyzed program.

3.6.4 Experience Report

In this section, we discuss five cases in which PAMM provided valuable insights for debugging and optimizations.

Bug Finding and Detection of Anomalies

The GNU core utility programs (coreutils) [Cor18] are frequently used as a subject for evaluations on real-world C programs (e.g. in [LWWX16, YMX+10, BS16, MKK07, FHJ+06]).

To check the capabilities of the PhASAR framework to handle real-world code, we benchmarked it on the coreutils using several different analyses encoded in IFDS. We found that some of the analysis runs caused a segmentation fault. The backtracing capability of the GNU debugger GDB gave no useful clues what might have caused the segmentation fault. The Valgrind [NS07] tool for dynamic debugging memory issues was not usable while analyzing the coreutils as it slowed down the execution too much in the order of days. Unfortunately, it also did not report any errors using the micro-benchmarks that have been used to develop the analyses. As we used PAMM to record the analysis runs of the different coreutils and visualized the results, we found a correlation between lines of code, number of call-sites of the programs and the occurrences of segmentation faults. The plot is shown in figure Figure 3.3. The analysis of coreutils with more than 240k lines of code has led to segmentation faults and with more than 20k call sites have been likely to crash. Based on the recorded data, we assumed that the recursive nature of our IFDS/IDE solver implementation could be troublesome due to the operating system's default stack limit for processes. Increasing the stack limit indeed solved the problem and almost all programs of the coreutils could be successfully analyzed using a larger stack limit. The exceeded stack limit has been confirmed with help of the Linux kernel's ring buffer, too.[3] A small number of coreutils still caused segmentation faults regardless of the chosen analysis. That suggested that either the framework or all analyses did not cope with an infrequently used language feature. The backtracing capabilities of GDB revealed the segmentation fault to be caused by the flow function that handled function calls. A manual inspection uncovered that the failure was caused by C-style variadic functions which have not been handled by the analyses yet. At call sites that call variadic functions, the number of actual and formal parameters may not match. After adjusting the responsible flow functions to under-approximate that language feature in the analyses, all analysis runs could be executed successfully. Our handling of variadic functions is unsound. However, it retains an acceptable level of precision whereas a sound handling would lead to impractically imprecise results (cf. Section 2.4).

In a different scenario, we inspected the distribution of data-flow facts generated by an analysis. That is, we wanted to know the sizes of the sets of data-flow facts generated by the flow functions. For that reason, we instrumented PhASAR's IFDS/IDE solver to record the number of data-flow facts (ESG edges) generated at each statement. With the help of that information, we aimed at optimizing for the container type used to store the data-flow facts. Our initial implementation used STL's `std::set` which implements a red-black tree [Bay72]. In order to optimize for the container type, we measured the occurrences of different set sizes for an IFDS taint analysis which are shown in Figure 3.4. Figure 3.4 confirms that the vast majority of sets only contain very few elements. Therefore, we might wanted to switch to an implementation that is better suited for small sets such as Boost's `flatset` implementation which uses a sorted vector and a binary search to allow for logarithmic lookups. Interestingly, however, some sets contained an exceptionally large amount of facts caused by what is called "overtainting" [SB09]. We revisited the implementation of the taint analysis and found a place at which all context-insensitive aliases have been accidentally tainted when a tainted value has been stored to a memory

[3] The kernel stores a certain number of (error) log messages in a ring buffer.

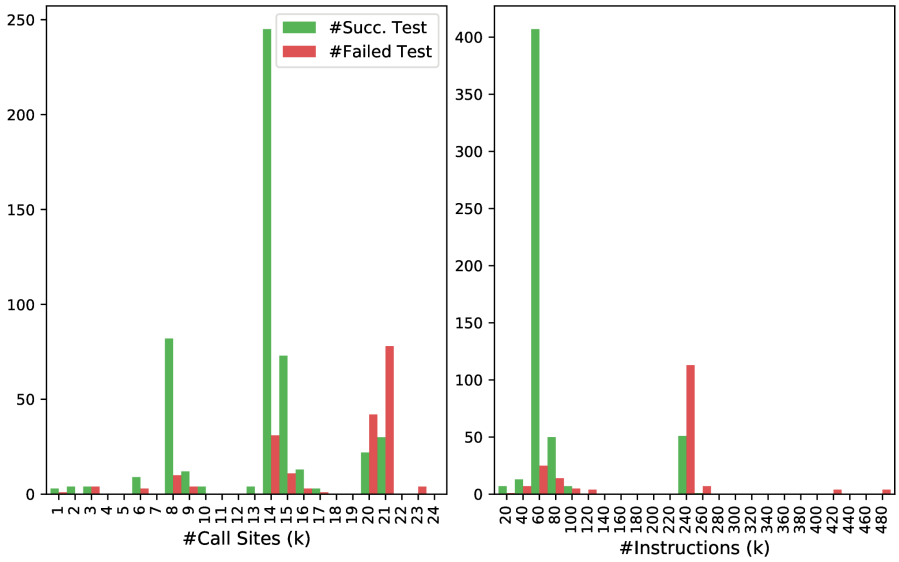


Figure 3.3: Number of analysis runs that executed (un)successfully and corresponding number of call sites and instructions of the program under analysis.

address. We could change the responsible flow function to only generate the relevant aliases. We will continue the discussion of the set implementations in terms of performance in Section 3.6.4.

Performance Benchmarking for Optimizations

Let us revise our assumption from Section 3.6.4 that the more compact `boost::flatset` implementation might be more efficient than the standard STL implementation in our case. In order to determine which implementation is better suited to hold the flow facts, we created a separate git branch in which we replaced the usages of `std::set` by `boost::flatset`. Since we initially already heavily instrumented PhASAR, we did not need to change any of the code other than specifying the container type to be used. We evaluated the performance by performing some analysis runs on the coreutils and some tools of the LevelDB project using a compile of PhASAR that uses `std::set` and compared the runtimes of various IFDS data-flow analyses with the figures obtained using a compile of the novel branch that uses the `flatset` implementation. Figure 3.5 shows a plot of the performance figures that we produced. In general, the difference in performance is negligible. The IFDS/IDE solver uses the sets to communicate with the analysis's description only. And in those cases, the compiler applies the *return-value optimization* (RVO) to directly construct the respective sets in the caller avoiding copying the set at the callee function's return instruction. Much more copying or accessing of those sets would be needed to cause a larger difference in performance. We thus sticked to C++'s STL `std::set` implementation for convenience.

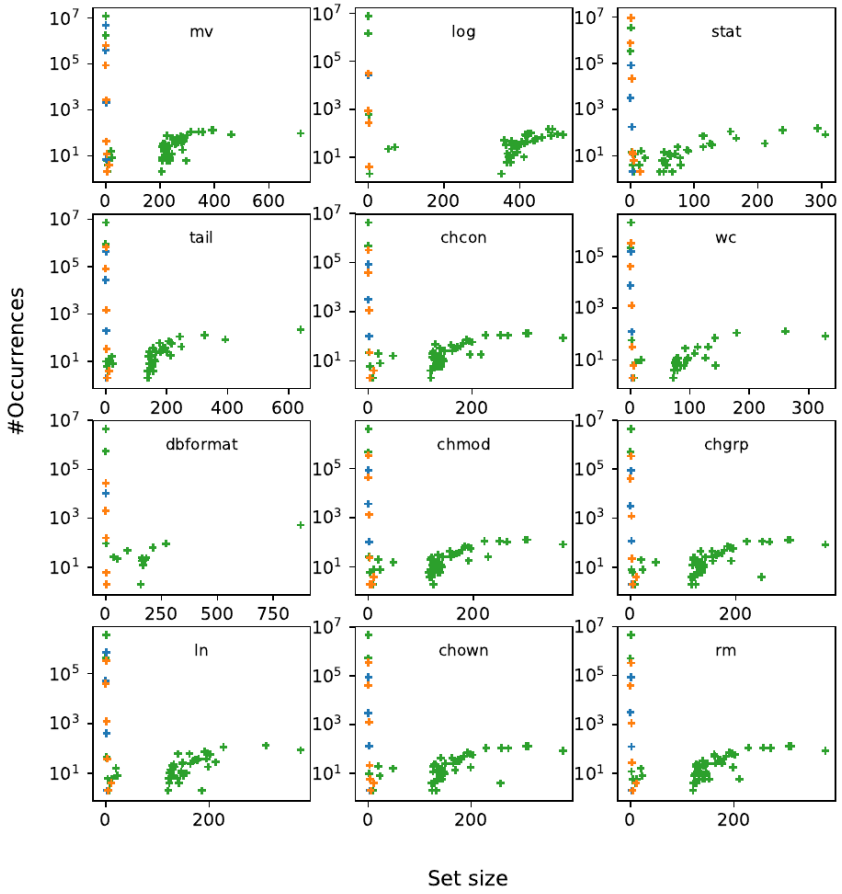


Figure 3.4: Occurrences of the different sizes of sets generated during ESG construction for several target programs.

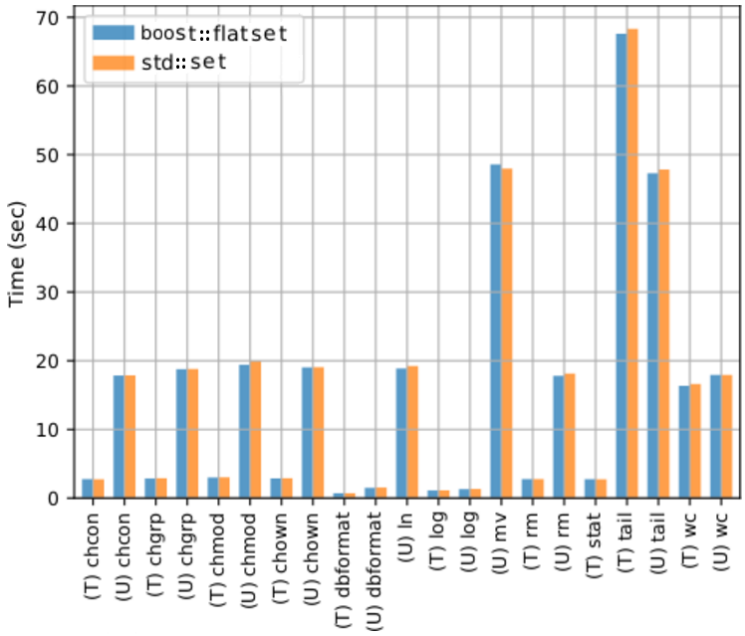


Figure 3.5: Runtimes using `std::set` versus `boost::flatset` in seconds for different programs and analysis runs.

The *C++11* standard introduced novel types for smart pointers that can be used to automatically deallocate heap memory that is no longer in use. `std::unique_ptr` can be used to handle memory that is limited to only one user; it is deallocated when the pointer goes out of scope unless ownership is explicitly transferred to another scope. Another type of smart pointer is `std::shared_ptr` that can be used if a piece of heap memory has more than one owner. It uses reference counting to determine at which point the memory can be deallocated. Our IFDS, IDE and WPDS solver implementations query the client analysis's code for flow and edge functions for each statement of the target program. The analysis code provides the respective solver with suitable implementations of these small function objects by returning a shared pointer. Shared pointers entail some amount of overhead due to the additional code that maintains the references and their larger size in memory. Using PAMM we were able to compare the initial implementation of PhASAR using smart pointers to an implementation that uses raw pointers. Figure 3.6 shows the comparison of smart and raw pointers in terms of runtimes. It can be observed that the use of shared pointers slows down each analysis run. We mitigated the noticeable slowdown due to the use of shared pointers as described in the following. Since the use of raw pointers in application code is considered bad style, we introduced a manager class that exclusively owns the shared pointers to flow and edge function objects following a recommended pattern: the manager class is able to hand out raw pointers to users that are "only looking" at the object; and eventually deallocates all objects it owns once its lifetime ends. Thus, the more expensive copying of shared pointers can be avoided which prevents the slowdown.

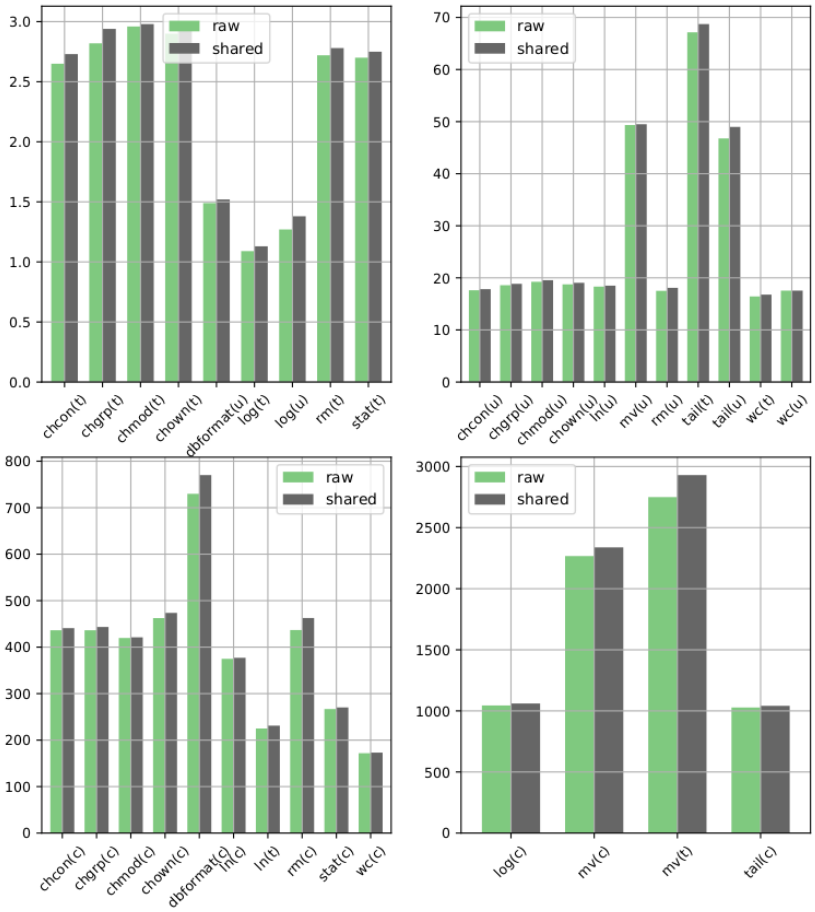


Figure 3.6: Runtimes using smart versus raw pointers in seconds for different programs and analysis runs.

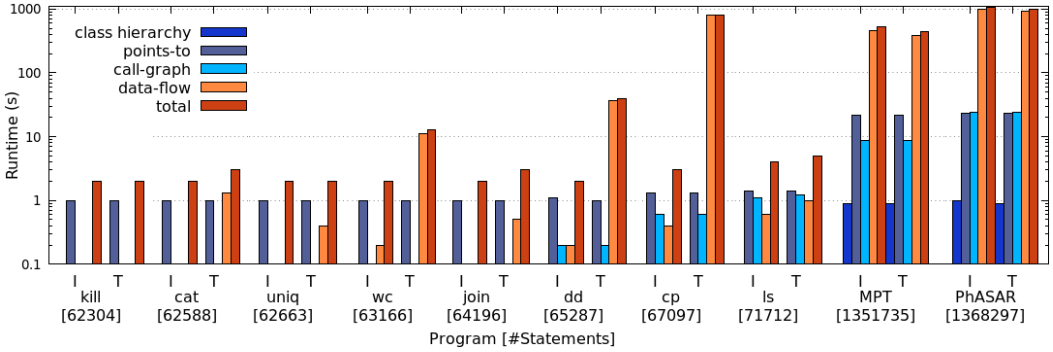


Figure 3.7: Runtime spent in different parts of an analysis.

In Section 3.6.2, we discussed that parameterizing an analysis framework is challenging. Always using the most precise algorithms wherever possible may lead to great precision but also to unsatisfactory performance for larger target programs. We do not want to rely on choosing an algorithms' parameters based on experience only. Depending on the target program under analysis the experience from analyzing one project might lead to false assumptions for another project. Therefore, we used PAMM to instrument all parts of PhASAR that are involved to perform a full analysis run. Thus, we are able to reveal the analysis runtime distribution of a concrete analysis run. Figure 3.7 shows such a distribution. Using that knowledge, one can then start adjusting specific parameters to speed up certain computations to cope with larger programs while comparing the precision based on the results obtained for the micro-benchmarks.

3.6.5 Related Work

The setup of a static analysis, encoding it in a framework, and finding a suitable parametrization, is a demanding task. Several works have been dedicated to reduce and ease the work that needs to be accomplished by analysis developers.

Lerch et al. developed an approach that follows the principle of separation of concerns [LH15]. They propose an approach that effectively separates different aspects and implementations to allow for better maintainability, testability, and reuse of individual components.

A special debugging environment for static analysis called VisuFlow [NKH+18] has been developed by Nguyen et al. for the Java ecosystem. It allows for a direct debugging of the analysis code in Soot without having to step through any of the framework code which makes the process of debugging an analysis feasible in practice.

3.6.6 Conclusions

In this section, we presented the design and implementation of a flexible mechanism for instrumenting PhASAR called PAMM. We presented five scenarios in which it provides us with valuable insights that help us to understand what a concrete static analysis run on

real-world code actually does. In general, we find that PAMM can be used in addition to or whenever standard debugging techniques are unable to track down the cause of an analysis failure. We advocate for integrating ready-to-use mechanisms that aid analysis understanding and debugging into the analysis frameworks to support developers, rather than burdening them with yet additional work. The data collected by the fine-grain instrumentation in combination with a suitable visualization allows for a gaze into concrete analysis runs. Thus, it enables us to spot anomalies and implausible figures. With these insights, we are able to determine how an analysis performs and where it goes wrong helping us to solve issues in a user's analysis code and the PhASAR analysis framework.

3.7 The Burden of Correctly Handling Global Variables

In this section, we present how one can provide framework support for handling global variables as required for precise data-flow analysis. Previously, global variables have been oftentimes either ignored or had to be tediously and explicitly modeled by an analysis writer.

Global variables make software systems hard to maintain and debug, and break local reasoning.

They also impose a non-trivial challenge to static analysis which needs to model its effects to obtain sound analysis results.

However, global variable initialization, codes of corresponding constructors and destructors as well as dynamic library code executed during load and unload not only affect control flows but data flows, too.

The PhASAR static data-flow analysis framework previously did not handle these special cases and also did not provide any functionalities to model the effects of globals. Analysis writers were forced to model the desired effects in an ad-hoc manner increasing an analysis' complexity and imposing an additional repetitive task.

This section presents the challenges of modeling globals, elaborate on the impact they have on analysis information, and present a suitable model to capture their effects, allowing for an easier development of global-aware static data-flow analyses.

We present an implementation of our model within the PhASAR framework and show its usefulness for an IDE-based linear-constant propagation that crucially requires correct modeling of globals for correctness and precision.

3.7.1 Framework Support for Global Variables

Global variables are best avoided. Not only do they increase the complexity of debugging and maintaining software systems but they also break local reasoning [WS73]. Global variables are used nonetheless to communicate information when using shared memory parallelism, to implement singletons, and to pass state across multiple functions without parameter passing.

Global variables are not only memory locations that can be accessed at all points in a program but they also come with code for initialization and de-initialization that is executed "before" *and* "after" **main**—the *actual* program. The situation gets more complex as there

are a multitude of different (de-)initializations depending on various conditions. Built-in types such as **int**, **long**, **double** are (de-)initialized differently than user-defined types, for instance. In addition, there is code that is executed whenever a shared library is loaded or unloaded which must be modeled, too.

A client data-flow analysis that verifies some property on a given target program does not only require data-flow information, but, in addition, information from various helper analyses such as callgraph and points-to. Depending on the complexity of the client analysis, it even requires information of additional data-flow analyses. Global (de-)initializers may affect all of these different analysis representations.

Static analyses are typically parameterized with a set of entry points that specify where in the program the analysis must start. Interestingly, the global (de-)initialization code is not explicitly connected with the program's actual entry point(s) such as **main**. If a user specifies the **main** function as an entry point to their analysis in a whole program analysis setup, the analysis still misses all of the global code that is executed "before" and "after" **main**.

PhASAR [SHB19] did not provide framework support for modeling the effects of global variables and associated code prior to this work. Other currently existing analysis implementations model globals in an ad-hoc manner or not at all. The effects of global code is modeled by repurposing flow-function implementations making the analysis code unnecessary complex and degrading analysis' performance. It is also unlikely that an ad-hoc handling of globals covers all possible scenarios and leads to sound analysis implementations.

While an unsound handling of globals may be reasonable for analyses such as uninitialized variables, which can safely ignore global variables as those are automatically zero-initialized in C and C++ if a programmer does not provide an initial value, many others crucially depend on a sound and precise handling of globals.

In this section, we thus present a structured overview on how global variables and associated (de-)initialization code are used in C and C++. We explain how these usages are represented in LLVM's [LA04] intermediate representation (LLVM IR) that is the target of PhASAR [SHB19] and many other analysis tools for C and C++. We elaborate on how to precisely model global effects for sound data-flow analysis and present an extension that we implemented for PhASAR [SHB19] to provide framework support. We show the usefulness of our model and its implementation by presenting a linear-constant propagation that crucially depends on correctly handling globals.

In summary, this section makes the following contributions:

- A comprehensive overview on the possible usages of global variables and global code in C and C++.
- A model and its open-source implementation in PhASAR [SHB19] that allows static-analysis writers to easily and soundly encode global effects into their analysis.
- A case study and an empirical evaluation that assesses the importance of correctly handling globals [Art21].

3.7.2 Background and Problem Description

In the following, we first present the various possible usages of global variables in C and C++ and describe their varying semantics depending on the situation they are being used. Then, we explain how the different semantics are represented in LLVM IR. We use these insights to design suitable abstractions that allow for precisely modeling the effects of globals in static data-flow analysis in Section 3.7.3.

Globals in C and C++

We present the different usages of global variables and their associated (de-)initialization code that is executed "before" and "after" the actual main program, respectively, by going through the code of Figure 3.8 line by line. We annotated the code to improve readability.

Built-in Typed Global Variables Line 2 declares a global variable that can be used across one or more compilation units as long as they contain a declaration of `i`. The variable `i` needs to be defined in exactly one compilation unit in which it is then automatically zero initialized as no explicit initial value is specified (cf. Line 15). The linker refers all users of `i` to this definition. In C and C++, all global variables are initialized with zero at compile time if no value is provided by the user as this does not entail any runtime costs.

Since *C++17*, the C++ standard allows for static inline definitions of global objects, i.e., *functions and variables* in header files (cf. Line 3). Due to the `inline` keyword this does not constitute a violation of the *one definition rule* (ORD). The one definition rule prescribes that non-inline objects (since C++17) and non-inline functions cannot have more than one definition in the entire program. Violations of that rule that span translation units are not required to be diagnosed and result in undefined behavior. Defining objects in header files using the `inline` keyword may produce multiple but equal definitions of the global object and therefore, it does not matter which definition the linker eventually arbitrarily picks and puts into the globals section of the final binary.

Line 6 and Line 7 depict analogous situations for class or struct types. Line 6 declares a global variable `c` that is part of the `Point` type. Similarly to the aforementioned situation, it must be defined in exactly one compilation unit (cf. Line 19). Consequently, the `inline` keyword allows for a definition in a header file without breaking ODR.

Line 16 defines the variable `k` that can be accessed globally but only within the compilation unit it is defined in. Line 18 shows an analogous situation using C++'s anonymous namespaces. The variable `l` is available across multiple compilation units within the namespace `ns`.

Class/Struct Typed Global Variables Line 20 defines the global variable `p`. Its constructor runs "at startup" before the C runtime starts the program's execution at `main`. Its destructor is called before exiting the program at the end of `main`. An analogous situation is depicted in Line 28. The function `getSingletonPoint` implements a thread-safe singleton, sometimes referred to as Scott-Meyers-Singleton, of type `Point`. The variable is initialized

```
1  // Header
2  Global var                 extern int i;
3  Global var                 static inline int j = 1024;
4                             struct Point {
5  Class member                 int a, b;
6  Static class member          static int c;
7  Static class member          static inline int d = 73;
8  Class default ctor           Point();
9  Class ctor                   Point(int a, int b);
10 Class dtor                   ~Point();
11                            };
12                            Point &getSingletonPoint();
13 // Implementation
14                            #include "overview-globals.h"
15 Global var                 int i;
16 Static init                static int k = 42;
17 Global in namespace        namespace ns { int l = 13; }
18 Anonymous namespace        namespace { int m = 9000; }
19 Static class member        int Point::c = 2;
20 Global class var           Point p(42, 13);
21 Class default ctor         Point::Point() : a(0), b(0) {
22                              printf("%d-%d", a, b);
23                            }
24 Class ctor                 Point::Point(int a, int b)
25                              :a(a),b(b){printf("%d-%d", a, b);}
26 Class dtor                 Point::~Point() { printf("%d", d); }
27                            Point &getSingletonPoint() {
28 Local static                 static Point s(11, 22);
29                              return s;
30                            }
31                            __attribute__((constructor))
32 Global ctor                void onLoad() { i = 9001; }
33                            __attribute__((destructor))
34 Global dtor                void onUnload() { i = 0; }
35                            int main() {
36                              Point &q = getSingletonPoint();
37                              return 0;
38                            }
```

Figure 3.8: An exemplary header and implementation file. The column on the left-hand side provides comments to ease readability of the code.

exactly once when **getSingletonPoint** is called for the first time. Its destructor is called before the program exits.

Global Con-/Destructors The definition of **onLoad** in Line 32 presents a global constructor. Function definitions that are annotated with the **constructor** attribute are executed while the compilation unit that defines these functions is loaded by the loader or the dynamic linker. Functions that are annotated with the **destructor** attribute present global destructors and are executed before the program exists. Line 34 shows an example for such a function. Even though the global constructor/destructor mechanism is currently not part of the C++ standard, it is often used in the context of shared libraries and therefore, should be supported by an analysis. Shared libraries may define several global con- and/or destructors that are executed when a shared library is explicitly loaded by another program using **dlopen** or **dlsym** and **dlclose**, respectively. In combination, this mechanism is used to implement plugins that (de-)register themselves within some other application that uses them.

Representation in LLVM IR

All types of global variables presented in the previous section can be found in the LLVM IR as well. Global variables whose access is restricted to the compilation unit or the function they are defined in are marked as **internal global**. Global built-in data types such as **int**, **char** or **double** are automatically initialized with zero.

Global variables of user-defined types are statically initialized with zero, too. Semantically, all data members of the given type are initialized with zero. Constructors and destructors come into play later.

LLVM provides two special global array variables **llvm.global_ctors** and **llvm.global_dtors** that carry information on the con- and destructors of global variables of user-defined types as well as global con- and destructors. The functions referenced by these arrays will be called in ascending order of priority, i.e., lowest first when the module is (un-)loaded by the loader or the dynamic linker. The order of functions with the same priority is not defined. Programs that introduce dependencies between global variables whose (de-)initialization code has the same priority are invalid.

When our module in Figure 3.8 is loaded, **onLoad** is executed. After **onLoad** has been executed, or before—in our case the priorities are equal, a special function responsible for executing the initialization code of all global, user-defined type variables is executed. Such a function is emitted for each compilation unit, if necessary. The linker handles merging these functions whenever modules are linked. The function itself calls other automatically generated functions each of which is responsible for initializing an individual global variable of a user-defined type. In our example, the function calls **p**'s constructor to correctly initialize it at the program's startup. It also registers **p**'s destructor to be called using the C runtime's **__cxa_atexit** function. The global variables' constructors are called in order of definition. Their destructors are called in reverse order once the program exits or the module is unloaded. Global destructors such as our **onUnload** function are registered in the **llvm.global_dtors** variable in an analogous way.

The **Point** singleton, like the other global variables, is zero initialized. Its initialization takes place at the very first call to **getSingletonPoint**. Depending if a compiler generated guard variable has been set atomically, its constructor is called and its destructor is registered in the C runtime. In case the guard variable is already set, this step is skipped and a reference to the initialized instance is returned directly.

3.7.3 Modeling the Effects of Globals

In this section, we present how global variables are currently handled by analysis writers and how one can model the behavior of global variables in a more stringent manner.

Status Quo

Current analyses that come with Soot [VRCG+99] or PhASAR [SHB19] (prior to this work) either ignore global variables completely or they repurpose an analysis' flow-function implementations to model their effects.

```
1  FlowFunctionPtrType getNormalFlowFunction(N Curr, N Succ) {
2    static bool InitGlobals = false;
3    if (!InitGlobals && InitialSeeds.count(Curr)) {
4      InitGlobals = true;
5      std::set<D> ToGenerate;
6      for (auto &Global : getGlobals()) {
7        if (Global.isConstant()) {
8          ToGenerate.insert(&Global);
9        }
10     }
11     auto GlobalsFF = std::make_shared<GenAll<D>>(ToGenerate,
           ZeroValue);
12     // Compute the flow function for the actual statement whose
           analysis
13     // we intercepted in the previous lines.
14     auto CurrFF = getNormalFlowFunction(Curr, Succ);
15     return std::make_shared<Union<D>>({GlobalsFF, CurrFF});
16   }
17   // ... code ...
18 }
```

Listing 3.6: An excerpt of global-variable-handling code using an IFDS [RHS95]/IDE [SRH96] *normal* flow function implementation.

The current scheme for modeling global variables that is often found in practice is shown in Listing 3.6. The scheme uses the flow function implementation by adding additional code that is executed once at the very beginning of an analysis. Because this scheme uses a call to the flow function that would normally be used to model the intra-procedural effects of the **Curr** statement, the query for **Curr** must be performed within the global-handling code and its result must be combined with the flow function that describes the effects of the global variables (cf. Line 15). The scheme as is, besides increasing the analysis' complexity, ignores code for (de-)initializing global variables. This handling is also not

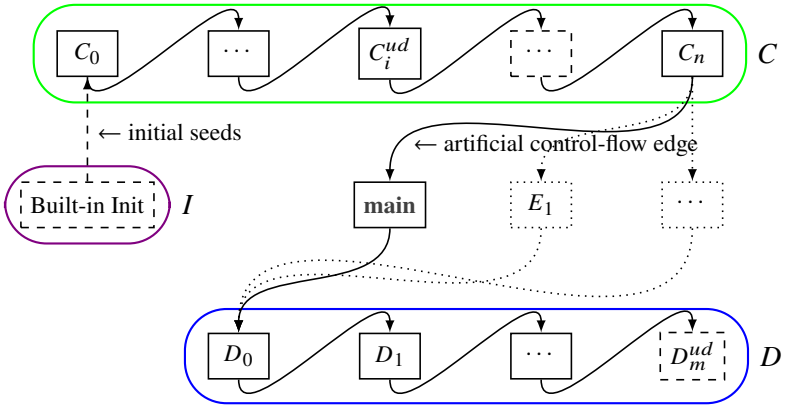


Figure 3.9: Schematic overview of a global-aware ICFG.

quite correct as it would not work if non-intra-procedural, i.e., non-*normal*, statements are chosen as entry points. One would therefore need to replicate the global-handling code in the flow functions that handle *call* and *return* flows, too.

We next describe how the most laborsome parts of modeling globals can be shifted to an analysis framework.

Control Flows

To conduct an inter-procedural, i.e., whole program analysis, an inter-procedural control-flow graph (ICFG) is required. An ICFG must be parameterized with one or more entry points $E_0, \ldots, E_n$. In case one wishes to conduct a whole program analysis, **main** is usually chosen as an entry point. However, by choosing **main** as an entry point, the ICFG misses lots of control flows that may be crucially important to the client analysis since a lot of functionalities involved in (de-)initialization are executed "before" *and* "after" **main**.

To produce a sound ICFG that supports whole program analysis, an ICFG algorithm must respect and analyze the global constructors. While analyzing the transitively reachable functions it must also register all functions that are registered to the C runtime using **__cxa_atexit** and retain their order. Only then an ICFG algorithm may analyze the control flows starting at **main**. Once the control flows of **main**—the main program, and transitively reachable functions have been computed, the algorithm must continue analysis in the global destructors and also the destructors that have been previously registered using **__cxa_atexit** in reverse order until *all* control flows have been analyzed and a complete model of the program under analysis has been constructed.

A schematic overview of an ICFG that respects global variables and global code for a given target program is shown in Figure 3.9. First, a global-aware ICFG must respect the primary initialization of global variables in order of appearance in the code indicated by the box labeled I. Note that these initializations are not bundled in a function and do

not represent instructions in LLVM IR. After considering the global variables, an artificial control-flow edge to the first global constructor C_0 must be introduced. The ICFG must determine the registered global constructors using the special **@llvm.global_ctors** variable, sort the functions according to their priority, analyze them, and introduce artificial control-flow edges between them. The return instructions of the first $n - 1$ global constructors conceptually transfer control to the next global constructor. As the behavior is not defined whenever two global constructors have the same priority, it does not matter in which order the ICFG organizes them; user code is not allowed to depend on initialization order in that case. We denote global constructors as C_i in Figure 3.9. Global constructors and destructors can call arbitrary functions of the program, however, we do not represent that fact in Figure 3.9 to avoid cluttering. C^{UD} denotes the special function, introduced by the compiler, that calls the constructors and registers the respective destructors of global variables of *user-defined* types. Once all global constructors have been analyzed, the control flow is artificially transferred to the actual user-defined entry point(s), i.e., **main** (and potentially a set of other entry points E_i). After constructing the control flows for the main program, an artificial control flow edge to the first global destructor D_0 are introduced. Similar to the constructors, the destructors are also chained according to their priority. The last global destructor transfers control flow to box that we denote as D^{ud} in Figure 3.9. D^{ud} is the sequence of calls to the destructors of global variables of user-defined types that have been registered in the C runtime. The D^{ud} destructors are called in reverse order of registration.

Data Flows

Similar to an ICFG algorithm that needs to be parameterized with a set of entry points, a data-flow solver needs to start at some program location(s). In the Soot [VRCG+99] and PhASAR [SHB19] frameworks, these program location(s) are referred to as *initial seeds*. Both frameworks allow analysis writers to specify the initial seeds by implementing a function of the appropriate problem interface that represents the analysis problem to be solved. The initial seeds mechanism allows analysis writers to not only specify the program locations but also data-flow facts that initially hold at these locations. The initial seeds implementation returns a mapping from start locations to a set of data-flow facts that hold initially.

Rather than using the flow-function implementations as described in Section 3.7.3, the initial seeds can be used to model the effects of the primary initialization denoted by the I box in Figure 3.9. In addition to the ordinary initial seeds that an analysis writer specifies for their analysis, they can iterate the global variables and their primary initializations in order of occurrence and model the effects by creating a set of data-flow facts that represents the behavior according to their concrete analysis problem. The propagation of this set of data-flow facts that represents the global variables is then started at the beginning, i.e., the first statement, of C_0. Due to the artificial control-flow edges introduced in the global-aware ICFG, the flow facts are made available to the global constructors and the behavior of those constructors can be modeled soundly. After the solver propagated the flow facts through the code in the box labeled C, they now capture the effects of any initializing code and can

then safely propagated into **main** (and potential other user-defined initial seeds E_i). At the end of **main**, the global variables are propagated through the chain of global destructors and destructors of global variables of user-defined types D^{ud} as indicated by the box labeled D.

To make the global variables available for analysis as data-flow facts at all statements, they are propagated into any potential call target at a given call site and propagated back to the caller at the callee's respective exit site(s). Data-flow facts that represent global variables must be killed at *call-to-return* flows to make the effects of callees visible to the subsequent program. In case only function declarations are available as call targets at a given call site, global variables are automatically killed by the *call* flow and instead must be propagated along the *call-to-return* flow. Otherwise, the globals would, again, not be available to the subsequent program. We discovered this special case while using the scheme presented here in a complex data-flow analysis we recently implemented.

3.7.4 Implementation

We implemented the scheme presented in Section 3.7.3 within PhASAR [SHB19]. We extended the existing LLVM-based ICFG implementation with functionalities that allow analysis writers to easily retrieve the global constructors and destructors. We also added an additional option to the ICFG's constructor to allow for ICFG construction that respects the functions that are called "before" *and* "after" the user-specified entry points, e.g. **main**, and correctly reflects the actual semantics of global variables and their (de-)initializers. If enabled, the global initializers C_i and C^{ud} are analyzed first and registered destructors D^{ud} are recorded. The ICFG then adds artificial control-flow edges to the actual user-defined entry points. From the exit sites of the user-defined entry point functions artificial control-flow edges to the global destructors D_i and registered destructors D^{ud} are added as shown in Figure 3.9.

We also generalized the **initialSeeds** implementation which, until now, has been shared across the IFDS [RHS95] and IDE [SRH96] problem interfaces. This, however, prevented analysis writers from specifying data-flow facts with initial values other than $\bot$ in IDE problems making it impossible to encode the effects of primary global initializations within the **initialSeeds** implementation. Our generalization now also allows for arbitrary initial edge functions in IDE [SRH96].

PhASAR's pre-defined flow function implementations for automated parameter mapping for *call* and *return* flows have also been extended. We added additional parameters to the respective flow functions that allow for automatically handling the data flows of global variables as described in Section 3.7.3.

3.7.5 Case Study: Constant Propagation

We demonstrate the usefulness of our PhASAR extension $\mathcal{G}^+$ by presenting how the new functionalities can be used to add global variable support to PhASAR's existing linear-constant propagation encoded within the IDE [SRH96] framework. We then present a quantitative evaluation that assesses the importance of correctly handling global variables (and code).

An Analysis Writer's Perspective

When constructing the target program's ICFG to conduct a global-aware whole program analysis, we specify **main** as an entry point and, in addition, turn on the option for global (de-)initializer awareness. The ICFG implementation then automatically analyzes the global code and introduces artificial control flows.

To capture the primary initialization (cf. *I*-labeled box Figure 3.9) we make use of the **initialSeeds** . We iterate the global variables using LLVM's standard API and create G a set of pairs of variables and associated edge functions describing their initialization. The set, among others, includes $k \mapsto \lambda x.42$, $j \mapsto \lambda x.1024$ for the program shown in Figure 3.8. We query the ICFG for C_0 and return as initial seeds a mapping from C_0's first statement to G. We use the extended flow functions for automated handling of inter-procedural flows, i.e., *call*, *return*, *call-to-return*, and enable the option allowing for automated handling of global variables. The correct propagation is then automatically handled by PhASAR's solver implementation which propagates the data-flow facts according to the global-aware ICFG.

Global Variables in Real-World Programs

Our empirical evaluation addresses the research questions:

RQ_1	*To what extend are global variables used in real-world programs?*
RQ_2	*How much precision does an analysis gain by making it global-aware?*
RQ_3	*What is the runtime cost of making an analysis global-aware?*

To address RQ_1, we counted the number of global variables for each benchmark program, recorded their respective types and determined their users by following their def-use chains. To address RQ_2, we ran a global-oblivious as well as a global-aware IDE [SRH96]-based linear-constant analysis that has been independently implemented in PhASAR on each benchmark target and compared the data-flow facts that have been generated and propagated by the analyses. We measured the analysis' running times to be able to comment on the expense that propagating the additional (global) variables incurs (RQ_3).

Experimental Setup

We have evaluated our framework extension $\mathcal{G}^+$ using as benchmark subjects 23 C/C++ programs that we obtained from Github. We compiled the programs to LLVM IR using WLLVM and subjected the resulting bitcode files to a linear-constant propagation, once using a global-oblivious $\mathcal{G}^-$ and once using a global-aware $\mathcal{G}^+$ version of the analysis. The target programs' corresponding LLVM IR ranges from 2,357 to 684,202 lines of code. We measured the running times for the experiments on an dual socket system with 2x Intel(R) Xeon(R) CPU E5-2630v4@2.20GHz CPUs and 256GB main memory, running Debian

Table 3.2: Results for the IDE-based linear-constant analysis.

program	#g	#u	#gen	#ntvas		#ntvae		runtime [s]	
				$\mathcal{G}^+$	$\mathcal{G}^-$	$\mathcal{G}^+$	$\mathcal{G}^-$	$\mathcal{G}^+$	$\mathcal{G}^-$
bison	1,806	7,130	113	113	0	113	78	2582	2295
brotli	163	272	0	0	0	0	0	143	142
curl	1,880	2,119	17	17	0	17	8	730	698
file	168	267	5	5	0	4	0	1	1
gravity	1,194	3,333	17	17	0	16	10	60586	60482
grep	415	978	60	60	0	60	46	290	256
gzip	351	2,007	97	97	0	96	15	63	47
htop	1,521	2,355	44	44	0	41	20	632	596
libjpeg	184	346	0					19780	19989
libpng	454	560	0					97	114
libssh	1,853	1,997	7					1232	1301
libtiff	1,309	1,422	1					560	645
libvpx[d]	1,372	2,778	19	19	0	19	0	10645	10160
libvpx[e]	1,682	3,191	21	21	0	21	1	12558	11974
libxml2	4,969	8,475	92					28555	29689
libzmq	1,191	3,154	0					1866	2481
lrzip	782	1,415	4	4	0	4	4	250	252
lz4	396	1,189	13	13	0	13	5	115	108
openssl	1,835	1,899	14					1642	2005
openvpn	4,343	4,893	41	41	0	0	0	21979	21994
opus	467	606	2					415	516
tmux	5,193	5,916	40	40	0	0	0	22246	22333
xz	455	932	48	48	0	46	35	33	26

10. We ran each experiment ten times and computed the mean time it took to execute the analysis. The mean relative standard deviation for all projects is 1.1%. Table 3.2 shows our results. The columns of the table present (from left to right) for each target subject the number of global variables, the number of their users, the number of global integer-typed variables that the analysis can potentially track, the number of constant variables that hold at the start of **main** and the number of constant variables that hold at the end of **main**—once using a globals-aware ($\mathcal{G}^+$) and once using the plain, unmodified constant analysis ($\mathcal{G}^-$)—as well as the running times in seconds. Our benchmark programs, the raw as well as the processed data produced in our evaluation are available in our artifact [Art21].

RQ_1: Usages of Global Variables

Table 3.2 shows that all of our real-world target programs make use of global variables. The amount of global variables used ranges from 163 to 5,193 with an average of 1,478.

These global variables, on average, have 2,580 users. Global con- and destructors using the __annotate__ keyword are used by two projects (`libssh` and `libzmq`) and thus seem to be used less frequently.

Global variables are frequently used throughout all of our target subjects. Hence, it is important for an analysis to model them correctly if a sound analysis is desired.

RQ_2: Precision

Our results for $\mathcal{G}^+$ show that most of the integer-typed global variables that are constant at the beginning of **main** remain constant or linearly depending on constants throughout the whole **main** function, i.e., the program. The `openvpn` and `tmux` programs present two exceptions where none of these variables remains constant. However, the results for $\mathcal{G}^-$ shows the necessity of handling global variables. Since global variable initialization is not taken into account by $\mathcal{G}^-$, it cannot find constant global variables at the beginning of **main**. As the amount of constant globals at the end of **main** indicates, there seem to be a few stores of constant values (or literals) to some of these globals. Still, the number of constant global variables at the end of the program lacks far behind $\mathcal{G}^+$.

While ignoring global variables might be acceptable for analyses that are used for bug finding, especially analyses that are concerned with software security or are used as a basis for program optimizations cannot afford to ignore these variables (and respective code of global (de)initializers).

RQ_3: Performance

As our results in Table 3.2 show, analyzing global variables impedes performances. This is because global variables need to be propagated through the complete program under analysis to represent the fact that they can be accessed (and modified) at any point in the program. Surprisingly, libraries benefit from our model. This is because PhASAR's points-to-based ICFG implementation and the global-oblivious analysis cause expensive repropagations when no dedicated **main** function (or C and D control flows cf. Figure 3.9) is present and global variables are discovered. Besides the implementation effort, this behavior can be mitigated in which case we expect $\mathcal{G}^-$ to be slightly less expensive than $\mathcal{G}^+$ similar to the non-library target subjects.

Supporting global variables impedes an analysis' performance. Making the IDE-based linear-constant analysis global-aware causes a performance hit of 7.5 % for ordinary programs and a performance gain of 12.6 % for libraries.

3.7.6 Related Work

Modeling the effects of global variables in static analysis is a demanding task. Doing so in a sound manner is virtually impossible for many realistic target programs. However, current

analysis frameworks such as Soot [VRCG+99] and PhASAR [SHB19] do not provide any framework support for modeling global variables.

Unfortunately, the compiler community does not provide solutions for comprehensive data-flow analysis of globals either. Optimizing compilers have to be rather conservative when it comes to performing code transformations, of course. While LLVM provides some optimizations w.r.t. global variables such as `globalsmodref`, `constmerge`, `globalopt`, and `internalize`, these are all rather simple analyses that back off as soon as a global variable's address is taken or its initialization is more complex. LLVM's implementations for (inter-procedural) constant propagation and constant folding does not optimize code that involves non-immutable globals with non-trivial (primary) initializers, and does not to aim to proof any properties for such variables or their users.

3.7.7 Conclusions

In the above, we have presented an overview on the complex semantics of global variables in C and C++ and how they map to LLVM's intermediate representation. Based on our observations, we presented a scheme that can be used to soundly model the effects of global variables in data-flow analysis. We extended the PhASAR [SHB19] framework and implemented new functionalities that allow analysis writers to model globals in an easier and more structured manner. We presented a possible usage of the proposed scheme and showed its usefulness by extending PhASAR's current IDE-based linear-constant propagation adding support for global variables. Using the proposed scheme allows one to trivially add sound, full-global support to any data-flow analysis. We already have successfully integrated this scheme with a non-trivial analysis that crucially depends on a sound handling of globals, which we discuss in Section 7.1. Handling global variables correctly and soundly is an important task that must be dealt with in order to build solid analysis implementations, which, in turn, serve as a basis for further advanced analyses.

3.8 A Few Years Later: Designing Static Analysis Implementations

In this section, we share our general experience in analyzing real-world C and C++ programs that we gained over the last few years. In particular, we report on the challenges of implementing static analysis toolboxes and what can make this task less cumbersome.

While traditional static analysis—albeit its complexity—is a topic that is well understood, we especially struggle when it comes to implementing its concepts. Designing and modeling software that implements static analysis is a challenging task.

However, developing usable static analysis implementations and providing toolboxes to researchers and practitioners is key to advance the overall progress in this field.

This section reports on the lessons learned from developing the PhASAR static data-flow analysis framework. We present some of the key mistakes of our first implementations of PhASAR and their corrections. From those corrections we distill guidelines that will be helpful to static analysis developers and their users.

In our work, we identified modularity as the key guiding principle supported—directly or indirectly—by virtually all other guidelines.

3.8.1 Experiences From Building a Static Analysis Framework

We implemented the PhASAR framework [SHB19] and open-sourced it in 2018 due to the lack of open-source data-flow analysis implementations for C and C++ that suited our needs. The analysis of C and C++ programs is notoriously hard, as this family of languages presents some unique properties that are seldom found in other languages. These properties include its low-levelness, arbitrary pointers to *memory* (including **void***), its deliberately unsafe type system, multiple inheritance, possession of a preprocessor, and language features such as **const_cast** and **setjmp/longjmp**—to list only a few. Yet, these languages are heavily used in practice making them relevant analysis targets.

While there are relatively lightweight analysis approaches that conduct *syntactic* checks on a given target program, and which are able to analyze even million lines of code in minutes, analysis approaches that compute *semantic* properties of a program are more heavyweight. Many interesting static analysis problems, such as data-flow-, shape, or type-state analysis require detailed, inter-procedural semantic program information. To solve these kinds of analysis problems, detailed abstractions are required that involve complex data-flow solvers, complex analysis domains, and oftentimes multiple different, potentially interleaving, helper analyses, forming an "analysis blend" that eventually provides useful results. This paper focuses on such semantic analyses.

For many real-world sized target programs, detailed abstractions that are necessary to solve those more heavyweight analysis problems lead to high runtime and memory requirements. This makes it almost impossible to integrate such analyses into software development processes, let alone compilers [Bab18]. Actual solutions to analysis problems are often undecidable, forcing analysis developers to resort to approximations. In addition, the complex concepts and algorithms that are required to solve analyses that reason about semantic properties of a program are one of the (many) reasons that lead to a restricted supply of static-analysis implementations that are able to solve those kinds of analysis problems.

Because there is no reference implementation, guide, or any form of advice on how to build a static data-flow analysis framework for C and C++, we initially borrowed several design decisions from the Soot framework [VRCG+99, LBLH11], and LLVM [LA04]. We built PhASAR on top of LLVM as it provides a usable, industrial-strength compiler infrastructure that offers an intermediate representation (IR) and, in addition, provides compilation of target programs into IR and all basic functionalities for inspection and transformation of the IR. Although we were able to use existing, static-analysis toolboxes and compiler infrastructures that allowed us to avoid repeating engineering mistakes others made before, such as (accidentally) introducing tight coupling, we still encountered various difficulties and had to learn many lessons the hard way.

In this section, we thus report on our findings of what makes the development of such frameworks easier. We present the basic concepts of static data-flow analysis in a nutshell and report on the key mistakes and design flaws of early implementations of PhASAR for

which we drew several design ideas from Soot [VRCG+99] and LLVM [LA04]. From their corrections and PhASAR's partial redesign we elaborate guidelines on how to model and implement static analysis that is usable in practice.

We found that the dominating overall design principle that static analysis implementations must follow is *modularity*. A modular design greatly counters complexity and allows one to build further abstractions on top of basic building blocks. Modularity is involved—directly or indirectly—in six of our eight major guidelines that we distilled from our experience.

In summary, this section makes the following contributions:

- It presents a report on the key mistakes and their corrections in designing and implementing the concepts of static data-flow analysis within the PhASAR [SHB19] framework,
- and shows guidelines derived from the corrections that will be useful to static analysis developers and their users.

3.8.2 Background

In this section, we briefly present the basic parametrization options of static analysis that need to be taken into account when modeling and implementing a static data-flow analysis framework.

Parametrization and Configurations

Virtually every algorithm that computes a piece of static analysis information can be heavily parameterized; oftentimes to trade off precision and scalability [HP00]. Depending on the concrete client analysis and given target program, some parametrizations may be more preferable than others.

In addition to the parametrization of individual analysis algorithms that even affect each others' properties, several configuration options may be applied that can be considered *global*. Those global configuration options apply to entire analysis runs, i.e., they apply to every entity involved in the dependency model presented in Section 2.3. Some of the configuration options are implementation independent. For instance, one could model *soundness* as a global option that may carry one of the values *sound*, *soundy* [LSS+15], or *unsound*. That option uniformly applies to all analyses required by a client and is independent of any concrete implementation. Other configuration options such as logging, export of results, etc. are global but implementation-dependent.

Analysis Styles

On top of the analysis setup presented above, various *analysis styles* or *strategies* may be used to conduct an analysis. Those styles include, among others, whole program, incremental, demand-driven, and compositional analysis. All these analysis styles require the same static-analysis information but each style requires them in a slightly different form. Demand-driven analysis, for instance, requires information on forward and backward

control flows [JKC20]. Incremental update analysis even requires additional communication between those different pieces of information.

3.8.3 Lessons Learned

In this section, we elaborate on implementation mistakes and design flaws we made and had to fix in PhASAR's initial implementations, respectively. We consider it a mistake whenever changes in the code or design have been necessary that required a disproportionate amount of time when building novel (analysis) abstractions on top of existing ones.

Modularity and Encapsulation To allow for efficient inter-procedural analysis, we built our initial implementation of PhASAR starting from a generic and parameterizable IFDS/IDE solver implementation similar to the Heros [Bod12] data-flow solver frequently used with Soot.

From our experience on Soot we knew that modularity is a key element when it comes to designing an analysis framework. Many of Soot's important data types are implemented as singletons that make it easy to globally access information wherever needed, but also break modularity and local reasoning. When requiring callgraph information, for example, a user sets up an instance of a callgraph type using its constructor. Especially novice users, however, cannot possibly know that there are additional setup possibilities using a singleton configuration object, as there is no direct connection between the type's interface and its setup. Those singletons also prevent several important use cases as it prevents loading multiple target programs into a single analysis process, for instance.

We borrowed several design decisions regarding the modeling of the solver interfaces from Soot. Thus, the solver operates on a "problem" interface whose implementations correspond to concrete analysis problems. The problem interface's constructor takes an implementation of the **ICFG** interface that guides the solver through the program. The client problem is also free to use the information provided by the ICFG. PhASAR manages the underlying target code using the **ProjectIRDB** type. One may thus conclude that providing an ICFG implementation that computes control flows on the code managed by the **ProjectIRDB** to the analysis "problem" would be sufficient.

While this design allowed us to implement and debug the solver, and generally allows to specify and solve basic inter-procedural client analysis problems, it oversimplifies the concepts of static analysis. For instance, the existence of points-to or type-hierarchy information is not mentioned at all, making it unusable for more complex client analyses.

Because of our modular design and missing capabilities to globally access information, it led to several severe code smells such as passing points-to information via the concrete ICFG implementation to the client problem and eventually prevented us from building further abstractions on top without completely breaking modularity and encapsulation, and thus losing control over the complexity.

Hence, we improved on our model and implemented it according to Section 2.3. We modeled each entity as an individual interface that can be separately used from all others.

We employ the type system to match our model shown in Figure 2.8 and express intent: an analysis problem always needs an ICFG implementation that guides the solver through

the program and is passed as a reference, and may uses additional information which we pass as pointers. These pointers can be `nullptr`s to indicate *information not available/used*. As the client analysis is potentially provided with all information on the helper analyses, it can, in turn, spawn additional (helper) data-flow analyses itself, if required.

Modularity and Encapsulation: Modularity and encapsulation are key to keep complexity manageable which has been impressively shown by the LLVM project [LA04], too. Design a model that is expressive enough to capture all interactions of the different analysis algorithms that are interesting to you, explicitly. Implement each entity of your model such that it can be used (and tested) individually.

Accessing Information Depending on what needs to be computed, the various involved algorithms will need to share lots of information.

Due to our prior experience with stateful singletons in Soot that not only decrease its maintainability but are also particularly bad for thread-parallelism, we could avoid large collections of information that are shared globally. Whenever possible we make information available using uniform parametrization across all entities of a certain type. For instance, we offer uniform constructors for all types of data-flow problems: call strings, IFDS, IDE, etc.—they all accept equal parameter lists.

This allows us to build further abstractions on top of our model shown in Figure 2.8. We recently started implementing an analysis strategy concept as presented in Section 3.8.2. It enables users to set up entire analysis runs that use one of the presented strategies in only a few lines of code.

To allow for an exchange of information for strategies that not only need to share but also update depending information like incremental update analysis, we use a special `ReviseInfo` type. We modeled the type to carry information on which kind of information needs updates and what pieces of code are affected. The corresponding strategy implementation has been built on top of the `ReviseInfo` type and controls the actual exchange of information using a mediator pattern. By using this model, we can keep a strict modular design and avoid making every piece of information globally available. We detail on how we use this model to make static analysis incremental in Chapter 6. Similar to the above, Helm et al. present a novel approach that allows for modular, collaborative program analysis by using so-called *blackboard* systems in [HKR+20].

Even the information that is general to all respective algorithms, such as the level of soundness that we implemented as suggested in Section 3.8.2, is managed separately for each analysis run. Modeling this information as global variables would forbid us from running individual analyses concurrently: a functionality that is often needed by more complex analyses that need to spawn additional helper analyses.

In our experience, the only information that can be shared globally safely is implementation-specific information that does not affect the semantics of an analysis. Thus, we implemented the constant global implementation-specific information about the system as a special thread-safe singleton configuration type that allows one to access this information.

Accessing Information: Avoid weakening interface boundaries that counteract modularity and encapsulation. If needed, rather than giving individual unrelated components access to each other, exchange information with help of proxy exchange types which are handled by a mediator. Provide unified interfaces to access information to ease building novel abstractions on top of existing ones.

Bugs and Debugging Once we solved analysis problems with a first version of the basic analysis infrastructure, we frequently observed crashes, strange program behavior, and incorrect analysis results.

Finding the root causes of bugs in static analysis is a challenging and time-consuming task, as many different analyses are involved while performing a concrete analysis run [SLHB19]. Standard debugging techniques such as debuggers are hard to use, as one needs to step through a tremendous amount of non-related solver code when debugging a client analysis. Complex analysis domains make it hard to even display interesting pieces of information in a meaningful way as we described in detail in Section 3.6.

For that reason in a subsequent revision we instrumented the entire framework. Each piece of code involved in solving a concrete analysis run has been instrumented using logging techniques and functionalities to record data that is relevant to static analysis like number of callgraph edges per call-site, data-flow facts generated per statement, etc. After post-processing the recorded data, we are able to gain insights about the undesired program behavior that eventually let us track down bugs.

Nguyen et al. built a specialized debugger called VisuFlow [NKH+18] to ease the debugging process. Unfortunately, this approach is currently only applicable to the Java ecosystem.

Extending on Lerch and Hermann's insights [LH15], we additionally follow a test-driven approach in PhASAR. We frequently observed that when implementing novel features, other seemingly unrelated parts of the framework broke; bringing us back to the problem described in this section. Those bugs would, without corresponding unit tests, either provoke further undesired behavior causing time-consuming debugging sessions, or—even worse—produce bugs that remain undetected and may corrupt critical analysis users. As a consequence, we now use test-driven development to implement all major parts of the framework.

Bugs and Debugging: Integrate means to allow for debugging especially complex parts, e.g. using instrumentation. Implement individual components using test-driven development to ensure their correctness and retain the ability to check correctness continuously in an automated manner.

Parametrization, Configuration and Usability The large amount of parametrization and setup options decrease overall usability, especially for novices.

In a first implementation of the IR-managing **ProjectIRDB** type, we offered a broad variety of functionalities, many of which could be accessed through public member functions, which in some instances required a distinct order of function calls, e.g. certain IR annotation

passes needed to be run before being used by other functionalities. This design turned out to be error prone and difficult to use as it is too easy to introduce mistakes by confusing the order of calls.

To make it more difficult to use the **ProjectIRDB** class in an incorrect manner, we thus revised large parts and moved lots of tasks directly into constructors. Based on the experience gained from Soot we avoided separating a type's interface and its setup. To reduce the amount of configuration needed, for non-essential parameters and configuration options we chose sensible default parameters. Whenever possible we reduced the number of parameters even further. For instance, a callgraph based on points-to information can be constructed by specifying the enumerator option **CallGraphAnalysisType::OTF** in the **LLVMBasedICFG** constructor's parameter list. This specific callgraph option requires additional points-to information. However, if no additional points-to information are provided by a user of that type, the required information is constructed on-the-fly rather than reporting an error to the user.

Parametrization, Configuration and Usability: Model entities from static analysis as types and couple a type's setup *directly* to its interface. If possible, avoid complex setup mechanisms, use simple constructors instead. For novice users, make it sufficiently hard to misuse a type. Reduce the amount of essential parameters to a minimum by providing suitable default parameters. Compute missing information on-the-fly rather than aborting with an error message.

Flexible Usage Modes Initially, we implemented PhASAR as a command-line tool. However, we received many requests to also allow for further use cases. We extended the framework to allow for the usage of a plugin mechanism. Users can thus ignore most of the framework's infrastructure and focus on specific details they are interested in without the need for modification and costly recompilation of PhASAR's code base. C++ compilation times are typically relatively long compared to C or Java, even for incremental builds. Reasons for that include the hundreds or even thousands of header files that need to be (re)processed for every compilation unit, the monolithic linking process, complex parsing of the complex syntax, code generated by templates, and optimizations. We counteract the compilation times with potent build machines.

Due to the framework's modularity, we could also offer individual functionalities as libraries. Thus, users are free to only choose whatever functionalities they are interested in and can integrate these parts in their own tools. We added full CMake support to PhASAR which eases using it as a library and building tools on top of it.

Since the removal of the aforementioned restrictions of the usages, we noticed that the number of people interested in the framework increased. We could observe a growing number of users and recently received several valuable performance optimizations for our callgraph algorithms from a company that uses our callgraph construction functionalities in their software product.

Flexible Usage Modes: Provide flexible use cases unless you have good reason to apply restrictions. Do not make any assumptions on the users's workflows because people will come up with usages that you did not think of.

Analyzing C, C++, and LLVM IR Although the LLVM IR is expressive enough to capture arbitrary source languages, we found that the characteristics and complexity of the source language propagate into the IR. Observe the following call site in LLVM IR: `%retval = call i32 %fptr(%class.S* dereferenceable(4) %ptr, i32 5)`, assuming C to be the source language, a plain function pointer is called. If C++ is the source language, we can not be sure whether a function pointer or a virtual member function of class `S` is called. Depending on the source language and the translation of its features into LLVM IR there could be even more possibilities which have to be checked to find a solution. This is the reason why we observed that the analysis runtime for C++ target programs is usually much higher than for C programs.

For more complex languages like C++ we also have to keep track of special member functions. These functions are mapped into ordinary LLVM IR functions that Clang places in a well-defined order in the generated IR. For some analyses like the declared-type analysis (DTA) callgraph algorithm, we need to be aware of these special member functions in order to preserve high precision.

We also found that even a well-debugged analysis that has been hardened on a large variety of test programs may still fail on production code as some corner cases have not been thought of. The large amount of information available to an analysis run makes debugging errors hard. A standard debugger does not suffice because an analysis writer has to step through a lot of code that is not relevant for them. For Java, a special dedicated debugger for static analysis has been developed [NKH+18] which shows the relevance of the problem.

Depending on the optimization passes that have been applied to code in LLVM IR before it is handed over to the analysis, it may have very different characteristics. Although optimization passes are required to have no impact on the semantics, the structure of the IR code changes. In our experience, it is helpful to start developing an analysis on small test programs that are translated into IR without optimization passes, and cover as many cases as the analysis should find. Once an analysis handles these test cases correctly or with the desired precision, optimization passes should be applied to the test cases. After rerunning the analysis the results should be checked against their unoptimized version. When applying an analysis to production code, the code should be compiled using production flags in order to analyze code that is as close as possible to what actually runs on the machine.

We found that the usage of debug symbols is helpful. The Clang compiler's -g flag can be added to propagate the debug symbols into the IR. Those can then be queried using LLVM's corresponding API. However, the debug symbols may not always present, which is why an analysis should not rely on them.

Analyzing C, C++, and LLVM IR: Take the characteristics of the input source language into account when analyzing the compiler's IR. If possible, use debug information to

provide extra information to the analysis that would be otherwise difficult or impossible to retrieve.

Build Systems The earliest versions of PhASAR used Makefile as a build system. This worked as long as PhASAR comprised only a few source files, but after a few months we realized that this harmed the project's maintainability. The monolithic Makefile made it difficult to organize the project in suitable subcomponents, to integrate other libraries, and to allow for cross-platform support.

At the point at which only the initial creator of the Makefile could maintain it, we stopped and replaced the build system with CMake. CMake is an open-source, cross-platform, *modular* tool chain that is designed to build, test and package software. It is now also the de-facto standard for many modern C and C++ open-source projects. Due to its modularity it allows for an easy integration with other software projects—a property that makes it suitable especially for research projects that often need to combine multiple projects to create a prototypical implementation quickly.

Build Systems: Choose a build system that suites the project's needs and integrates well with others in advance. Think ahead and assume that the project will grow not only in terms of its code base but also its number of users.

LLVM IR Generation Following Section 3.8.3, we develop small micro benchmarks comprising several single-file programs that are used to test certain aspects of an analysis implementation.

While LLVM IR can be obtained for individual compilation units by running the clang compiler with the `-emit-llvm` flag, it is difficult to obtain LLVM IR for larger, more complex projects. However, that is exactly what analysis writers wish to do in order to evaluate an analysis' scalability and ability to deal with real-world code. C and C++ neither have a real module system nor a standardized build mechanism. Instead, individual compilation units are compiled into object files that are eventually linked to (hopefully) produce the desired binary. Preprocessor macros and other important flags passed to the compiler can change the semantics and correctness of the final binary. To produce LLVM bitcode for a given real-world project, one needs to extract the exact compile *and* link commands encoded in the build system used by the project. Doing so manually is an infeasible task if one recalls the multitude of different build systems such as Makefile, CMake, Bazel, etc.—if the project uses a build system at all.

Luckily, compiler wrappers such as WLLVM [WLL21] and (a faster implementation in Go) GLLVM [GLL21] have been developed. These tool chains interrupt the compiler and extract the compile command to produce LLVM bitcode for the compilation unit under processing. The path to the LLVM bitcode is stored in an artificial section in the resulting object code. Linker commands are interrupted as well, and, in addition to the ordinary linking job, the bitcode paths of the object codes that are linked are collected and placed in an artificial section of the resulting binary. To produce whole program LLVM IR, the paths

to the bitcode files that constitute the binary can be automatically extracted and linked, and finally subjected to a whole program analysis.

LLVM IR Generation: Use WLLVM [WLL21] and GLLVM [GLL21] to build whole program LLVM bitcode files from unmodified C and C++ projects.

Contributing Guidelines We are still affected by having failed to provide contributing guidelines in the early days. Initially, we did not provide suitable contributing guidelines and coding standards, and after we did, we did not enforce them at first. Due to the various contributions from students and practitioners that the project received over time, it has picked up different coding styles and code of varying quality.

A unification of coding styles and overall improvement of code quality using automated analysis and transformation tools such as `clang-tidy` was not directly possible due to various corner cases that those automated approaches cannot handle. Manual unification was very expensive and underwent an incremental process. We updated pieces of code that are adjacent to new features to ensure software evolution over time. It finally allowed us to remove those corner cases and to employ automated tooling.

Contributing Guidelines: Provide contributing guidelines and documentation at the beginning of the project. Assume that the framework has multiple users and developers that provide contributions, which is what eventually will happen. Use tools for automated analysis and transformation to retain a uniform and high-quality code base. Take measures to support community building and communication.

3.8.4 Related Work

There are several mature program-analysis frameworks from academia like Soot [VRCG+99], Opal [EH14], WALA [Wal19], or Doop [Doo18], there is very little advice on how to actually design and implement the underlying theory.

Some insights on good design of static analysis frameworks, provided by Soot's maintainers [LBLH11], refer to avoiding redundant re-computations by using incremental or reactive computation, and quasiquoting for easily generating code from templates. Allowing to independently release framework extensions without having them included in the main distribution also greatly benefits the tool and its community.

Experience reports on applying static analysis tools in commercial context emphasise the importance of low false positive rates and clear error messages to overcome warning blindness of tool users [SAE+18]. Additionally, tools must handle real-world code: resilience and robustness is vital when coping with large code bases and peculiar code constructs [BBC+10].

An extensive experience report on how to employ distributive, summary-based static analysis to benefit analysis precision and performance is given by Bodden [Bod18]. The report presents practical design tricks for data-flow analysis.

We present an approach that modularly computes and summarizes all pieces of static analysis information required to answer queries of a concrete client analysis as shown in

Figure 2.8 in Chapter 5. This approach shows that modularity not only eases implementation but also improves flexibility and counters the complexity of static analysis itself.

3.8.5 Conclusions

In the above, we reported on major design flaws and implementation mistakes that we detected in our first implementations of the PhASAR framework. From those incidents, for which we had to provide corrections in order to keep the complexity manageable, we distilled guidelines that we think are useful to static analysis writers and its users.

As shown in this section, even when using knowledge of the past and falling back to design ideas of existing frameworks one may still suffer from design decisions that turn out to be not advisable. Applying those guidelines helped us to improve PhASAR's overall quality. It reduced complexity, made its usage less error prone and eased building novel abstractions, eventually advancing the progress in this field.

3.9 Future Work

In this section, we briefly summarize our plans for future improvements.

It would be interesting to evaluate the use of PhASAR for analyzing different intermediate representations. One type of IR might come with advantages over others, depending on the analysis problem to be solved. We plan to additionally support the GENERIC, GIMPLE and RTL [Mer03, gcc18b] as well as the Multi-Level IR [MLI23] intermediate representations from the GCC and MLIR projects.

Other interesting frameworks for solving distributive data-flow analysis problems are *Pushdown Systems* [EHRS00] (PDS) and *Weighted Pushdown Systems* [RSJ03, SAB19a] (WPDS). Both, PDS and WPDS are able to solve data-flow analysis problems using stack automata. PDS and WPDS allow for more compact data structures, the generation of witnesses, as well as precise queries specifying paths of interest using regular expressions. Späth et al. opens up a new line of research by synchronizing two separate pushdown systems to allow for solving distributive data-flow problems in a flow-, ∞-field- and ∞-context-sensitive manner [SAB19b] to the extend that is theoretically possible [Rep00]. Späth et al. implemented their *Synchronized Pushdown Systems* [SAB19b] (SPDS) approach on top of Soot [LBLH11] for the Java ecosystem. We plan to support PDS, WPDS and SPDS in future versions of PhASAR. However, we have already identified several conceptual challenges when trying to apply SPDS directly to languages from the C family, which would make this an interesting and challenging piece of research.

Checking the correctness of an IFDS/IDE analysis is complex, since checking the correctness of the underlying exploded super-graph is tedious and time consuming. A high-quality visualization is likely to help reducing the amount of time spent debugging an analysis. In addition, a graphical user interface will reduce the amount of knowledge that is required to use the framework.

Since the flow (and, in case of IDE, edge) functions have to be implemented in a general purpose programming language, they require some amount of boilerplate code. The Clang

compiler's ASTMatcher [AST23] library contains a small, yet powerful embedded domain-specific language (EDSL) that allows one to specify sophisticated AST-based analyses and transformations using expressions that can be read like English sentences. It remains an open question, if one could design a non-Turing-complete EDSL with a library like `boost::proto` [Boo18], for instance, which simplifies the task of encoding data-flow problems in PhASAR.

PhASAR currently uses LLVM's points-to information, which is rather imprecise, unfortunately. We plan to integrate a more precise pointer analysis with PhASAR to support more precise callgraph construction and client data-flow analyses that require precise points-to information by adapting the demand-driven Boomerang approach presented in [SNAB16] to PhASAR.

3.10 Conclusions

In this chapter, we presented our implementation of a static analysis framework for programs written in C and C++ named PhASAR. We presented its architecture and implementation from a user's perspective to make practical static analysis more accessible. We presented experiments which have shown PhASAR's scalability and discussed the runtimes of the key parts of two concrete client analyses. We further presented valuable extensions that make developing static program anlaysis easier and shared the experience that we gained over the last few years.

With PhASAR we strive toward the goals of providing a framework for static analysis targeting (but not limited to) C and C++, a base for quickly evaluating novel ideas and applications, and a suitable way of handling the complexity. PhASAR is open-source and available under the permissive MIT licence, and therefore, open for contributions, feedback and use. PhASAR has already received tremendous support in the research community and from practitioners as indicated by the respective statistics on Github.

Since with PhASAR we now have a toolbox and the basic building blocks for statically analyzing C and C++ code by automatically inspecting the LLVM compiler's intermediate representation, we next discuss how to make C and C++ target programs, which in each non-trivial case actually represent software product lines, analyzable in practice and show how precise whole-program analysis can be scaled to large, real-world programs and modern software development workflows.

4 Variability

Following our goal of making the analysis of large, unmodified, real-world C and C++ projects feasible and scalable in practice, we next detail on how to make C and C++ code bases analyzable using distributive data-flow solvers such as IFDS [RHS95] and IDE [SRH96].

Many critical codebases are written in C, and most of them use preprocessor directives to encode variability, effectively encoding software product lines. These preprocessor directives, however, challenge any static code analysis.

SPLlift, a previously presented approach for analyzing software product lines, is limited to Java programs that use a rather simple feature encoding and to analysis problems with a finite and ideally small domain. Other approaches that allow the analysis of real-world C software product lines use special-purpose analyses, preventing the reuse of existing analysis infrastructures and ignoring the progress made by the static analysis community.

This chapter presents VarAlyzer, a novel static analysis approach for software product lines. VarAlyzer first transforms preprocessor constructs to plain C while preserving their variability and semantics. It then solves any given distributive analysis problem on transformed product lines in a variability-aware manner. VarAlyzer's analysis results are annotated with feature constraints that encode in which configurations each result holds.

Our experiments with 95 compilation units of OpenSSL show that applying VarAlyzer enables one to conduct inter-procedural, flow-, field- and context-sensitive data-flow analyses on entire product lines for the first time, outperforming the product-based approach for highly-configurable systems.

4.1 Introduction

Software product lines (SPLs) enable software developers to encode a set of software products in a common code base. The different variations, so-called configurations, are typically described with the help of static conditionals, so-called *features*, that enable conditional compilation. In the programming languages C and C++, developers typically use the preprocessor's functionalities, particularly the well-known `#ifdef` directives, to establish SPLs. The preprocessor's static conditionals allow developers to check the presence of a symbol or its value—an integer or a string literal. At compile time, the preprocessor transforms every compilation unit according to the given set of symbols (and their respective values), before the preprocessed compilation unit is handed over to the actual compiler. The compiler thus only compiles the code that has been included by the preprocessor, which allows it to produce efficient object code. This also means that in the worst case an SPL induces a number of software products that is exponential in the number of static conditionals.

Static data-flow analysis is not only used as a basis for compiler optimizations [Onl18, ICC18], but also for bug finding [CS18, Cod18] and software hardening [ARF+14, KNR+17, LL05, HREM15, HHL+17]. However, previous software vulnerabilities such as Apple's FileVault vulnerability [Pro12] show that program analysis of configurable systems is crucial. The FileVault vulnerability was caused by accidentally shipping a Mac OS X version with logging code enabled that stored the user login passwords in clear text. Such a vulnerability might have been detected early, had Apple had the capability to analyze FileVault's codebase with respect to all possible configurations.

The problem with traditional static analysis techniques, however, is that they cannot be applied to software product lines directly. Instead, one must first generate a concrete software product by preprocessing the common code base and then analyze the resulting plain C or C++ program. Due to the possibly exponential number of software products in practice, this process becomes prohibitively expensive even when analyzing only a few variants, let alone all possible software products.

SPLlift [BTR+13] was proposed to analyze an entire SPL as a whole, a so-called family-based approach [TA12], which avoids generating all potential software products. While doing so, it avoids an exponential blowup through a time and memory efficient encoding of feature constraints in distributive flow functions. However, SPLlift is restricted to *Interprocedural Finite Distributive Subset (IFDS)* [RHS95] problems, which include simple problems such as taint analysis, but exclude problems with large or potentially infinite domains such as constant propagation [SRH96] or typestate analysis [Str83, SY86]. More importantly, it is a prototype for a seldom-used product-line dialect of Java [KTS+09] and thus cannot be applied to real-world SPLs, particularly not those that use the C preprocessor.

Existing techniques that are able to analyze real-world SPLs written in C operate on un-preprocessed C code and include new or modified algorithms for parsing [KGR+11, GG12, GJ05], data-flow analysis [LvRK+13, RLJ+18], type checking [KOE12], and rewriting [IMD+17]. The only available data-flow analysis [LvRK+13, RLJ+18], however, is intra-procedural only. In addition, all those techniques are special-purpose analyses, making it infeasible to reuse existing state-of-the-art static analysis infrastructures. The situation becomes even more complicated when looking at the long term. While the research on "variability-oblivious" program analysis marches on, those variability-aware toolchains must be maintained in parallel, doubling the engineering effort, which explains why none of the above approaches have been maintained in the long term. Other works proposed new preprocessors [MB05, Käs10]. Language adoption, however, is a notoriously slow development. And even *if* those new preprocessors get adopted over time, one cannot expect that millions of lines of existing legacy code will be rewritten. Despite C's known issues, it is the most popular programming language according to the TIOBE programming index.[1]

In this work, we present the design and implementation of VarAlyzer, a novel static data-flow analysis approach built on top of SuperC [GG12] and PhASAR [SHB19]. The idea is to revoke the preprocessor's special role by first transforming preprocessor directives into ordinary C code. Preprocessor conditionals are replaced with C conditionals, preprocessor

[1] As of March, 2021, TIOBE programming index `https://www.tiobe.com/tiobe-index/`

macros are replaced with C variables, and the existence of declarations is controlled via C expressions that use these declarations. The transformation uses a configuration-aware type checker which supports static behaviors at runtime that could not be implemented before, e.g. type errors caused by infeasible configurations are expressed as runtime calls to an error function. VarAlyzer allows one to automatically make any existing (or new) distributive data-flow analysis on real-world C software product lines *variability-aware* which it then solves in a single analysis run on the transformed software product line.

On top, and in contrast to SPLlift, VarAlyzer does not just support analyses encoded in IFDS [RHS95] but also in *Interprocedural Distributive Environments* (IDE) [SRH96], which includes problems with infinite domains. As a result, VarAlyzer outputs the fully context- and flow-sensitive data-flow facts along with a feature constraint describing the product configurations for which they hold. This allows developers to find bugs and vulnerabilities much earlier in the development process, requiring no product to be generated. Whereas previously developers of highly-configurable software had to identify vulnerabilities separately for each concretely preprocessed variant, using VarAlyzer they can exclude such vulnerabilities in all relevant configurations ahead of time.

We evaluate VarAlyzer's effectiveness by conducting a typestate analysis [Str83, SY86] that checks for the correct usages of OpenSSL's Envelope (EVP) APIs on 95 compilation units. Typestate analysis belongs to an important class of analyses whose efficient encoding, due to the internal state, requires IDE [SRH96] or equally expressive frameworks such as weighted pushdown systems [RSJ03]. The IFDS-based SPLlift approach thus could not solve such an analysis on realistic programs. The (hand-)written compilation units in C comprise realistic uses of EVP's APIs for message digest (MD), encryption/decryption (CIPHER), and message authentication codes (MAC). The compilation units, ranging from 8 to 219 lines of code, comprise preprocessor conditionals and valid as well as invalid API usages. For this work, we have to restrict ourselves to evaluating our approach on individual compilation units because several fundamental challenges that are beyond the scope of this paper currently prevent us from evaluating VarAlyzer on full SPLs. Large-scale projects not only encode variability in the preprocessor but also in other parts of the system software toolchain. To support full SPLs, an approach would additionally need to solve the difficult problem of supporting variability-aware linking and build automation. VarAlyzer provides full support for application configurations. However, system configuration macros provide yet another challenge. Not only would an approach need to support platform-dependent header file differences, but would also require one to construct a superset of all C variations. We detail on these challenges in Section 4.3.1.

The implementation of VarAlyzer is available as open source. VarAlyzer's transformation parts are part of the SuperC project and can be found at `https://github.com/appleseedlab/superc`. Its variational-aware analysis is available as part of PhASAR and can be found at `https://github.com/secure-software-engineering/phasar`. VarAlyzer's implementation is available under the permissive MIT license. All accompanying artifacts of this paper, including processed analysis targets and result data, are available as supplemental material [Art21].

In summary, this chapter makes the following contributions:

- A novel end-to-end variability-aware static analysis approach that enables variational analysis of C software product lines. The approach transforms software product lines to ordinary C code while preserving the complete preprocessor semantics and performs an automated lifting that allows one to solve *arbitrary* distributive data-flow problems in a *variability-aware* manner.
- An open-source implementation based on SuperC [GG12] and PhASAR [SHB19].
- An experimental evaluation of VarAlyzer, which assesses its effectiveness in solving general IDE [SRH96] problems on 95 compilation units that use OpenSSL.
- An assessment of the further challenges that need to be overcome to make static analysis of arbitrary C applications a reality.

4.2 Motivating Example

To motivate the need for variability-aware analyses, we show an example using typestate analysis on a software product line. Most APIs are required to be called in a particular order or pattern. The valid sequences of operations can be encoded using state machines. A typestate analysis [Str83, SY86] or protocol analysis is a static analysis that tracks variables of a certain type and their associated states through the program. Typestates define sequences of operations that may be performed upon a variable. The state information associated with each variable is used to determine—at compile-time—the validity of operations invoked upon variables. Existing analysis techniques for SPLs that rely on special-purpose analyses formulated for variability-preserving ASTs cannot solve this problem class.

The state machine shown in Figure 4.1 describes the valid usages of OpenSSL's EVP message digest (MD) API. An SPL that performs a message digest using OpenSSL's EVP message digest (MD) API is shown Listing 4.1. The SPL comprises a debugging feature encoded with the symbol **DEBUG**. When this symbol is enabled in the preprocessor, and therefore debugging is enabled at runtime, MD's API protocol is violated as the call to **EVP_DigestFinal_ex** at Line 21 is omitted—a potential security threat. Even variability-aware intra-procedural data-flow analysis cannot properly solve this analysis problem in our example program because the variable **MDCTX** that carries the state information is processed across multiple different functions.

Traditional techniques would first generate a particular variant (and all variants we are interested in, possibly all of them) of the SPL, and then uncover this problem in a static analysis of that particular variant. A brief inspection of our example SPL using the GCC compiler shows that it comprises 6,946 preprocessor macros and (transitively) includes 221 different header files. 261 of those 6,946 macros are used in preprocessor conditionals. Therefore, traditional analysis techniques can not scale. Instead, it is desirable to analyze all potential configurations, i.e., feature combinations, at the same time. By transforming the preprocessor directives into ordinary C code, our approach allows to employ any existing C analysis tools to analyze the entire SPL as a whole. PhASAR's traditional typestate analysis, for instance, would be able to detect the protocol breach caused by the missing

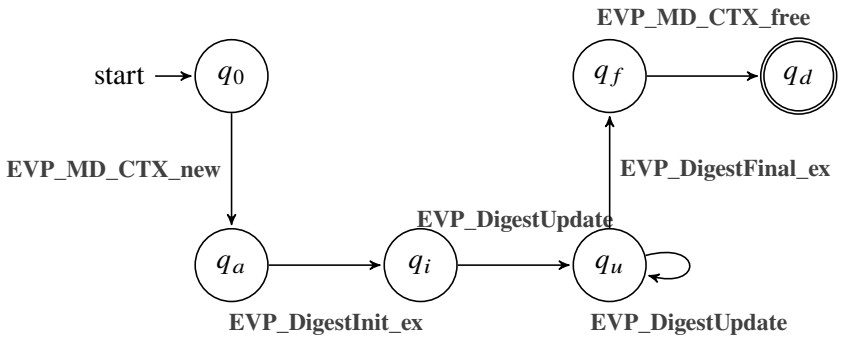


Figure 4.1: State machine that describes the correct usages of the OpenSSL EVP message digest (MD) API.

call to **EVP_DigestFinal_ex**. In more complex scenarios, however, it would also report a large number of false positives because the results are valid across all configurations, making any findings virtually impossible to debug. Traditional analysis would need to merge information at control-flow merge points even for branches that originate from static preprocessor conditionals, which is impossible in practice. Therefore, it is desirable to have an analysis that can handle preprocessor variability to produce results that are actually useful to the analysis users.

4.3 Analysis

In this section, we detail our approach to statically analyzing C software product lines. An overview on the various possible workflows that allow for analyzing software product lines is shown in Figure 4.2. VarAlyzer consists of two phases. First, it transforms software product lines into an intermediate representation (IR). Second, it applies a novel data-flow solver that enables *variational* analysis of arbitrary distributive analysis problems and produces precise results for all variants of a software product line in a single analysis run.

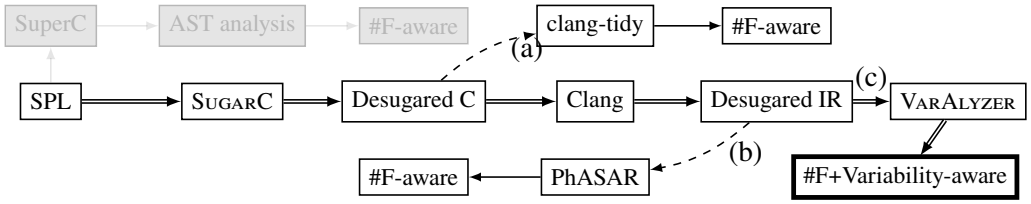


Figure 4.2: Overview on the various SPL analysis workflows. Our approach allows for the workflow denoted by the double arrows (⇒). The dashed arrows (⇢) denote alternative workflows that our approach enables as a by-product. Analysis results produced by workflow (a) and (b) are valid across all possible configuration. Workflow (c) produces results that are variability-aware.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #include <openssl/crypto.h>
5  #include <openssl/evp.h>
6
7  void digestMessage(EVP_MD_CTX *MDCTX,
8      const unsigned char *Msg,
9      size_t MsgLen,
10     unsigned char **Dgst,
11     unsigned int *DgstLen) {
12   EVP_DigestInit_ex(MDCTX, EVP_sha256(), NULL);
13   EVP_DigestUpdate(MDCTX, Msg, MsgLen);
14 #ifdef DEBUG
15   const char DebugHash[] = "Hello, Hash!";
16   *DgstLen = sizeof(DebugHash);
17   *Dgst = OPENSSL_malloc(sizeof(DgstLen));
18   strncpy((char *)*Dgst, DebugHash, *DgstLen);
19 #else
20   *Dgst = OPENSSL_malloc(EVP_MD_size(EVP_sha256()));
21   EVP_DigestFinal_ex(MDCTX, *Dgst, DgstLen);
22 #endif
23 }
24
25 int main() {
26   const char *Data = "My secret data.";
27   unsigned char *Dgst;
28   unsigned int DgstLen;
29   EVP_MD_CTX *MDCTX = EVP_MD_CTX_new();
30   digestMessage(MDCTX,
31                 (const unsigned char*) Data,
32                 strlen(Data),
33                 &Dgst,
34                 &DgstLen);
35 #ifdef DEBUG
36   printf("hashed data: %s\n", Dgst);
37 #endif
38   EVP_MD_CTX_free(MDCTX);
39   OPENSSL_free(Dgst);
40   return 0;
41 }
```

Listing 4.1: An SPL using OpenSSL's EVP message digest (MD) functionalities. Error handling code has been omitted for brevity.

```
1  #ifdef M1
2  extern void
3   f(int x);
4  #endif
5  #ifdef M2
6  static void
7   f(int x) { }
8  #endif
9  void g() {
10   f(10);
11 }
```

(a) Before

```
1  const bool M1, M2;
2  extern void
3    __f_1(int x);
4  static void
5    __f_2(int x) { }
6  void g() {
7   if(M1 && !M2) __f_1(10);
8   if(M2 && !M1) __f_2(10);
9   if(M1 && M2||!M1 && !M2)
10     __type_error();
11 }
```

(b) After

Figure 4.3: Desugaring a variational function definition, adapted from Linux v4.18 kernel/sched/sched.h.

```
1  #ifdef MACRO
2    if (cond)
3      i = 31;
4    else
5  #endif
6      i = -32;
7
```

(a) Before

```
1  const bool MACRO;
2  if (MACRO) {
3    if (cond)
4      i = 31;
5    else
6      i = -32;
7  } else
8    i = -32;
```

(b) After

Figure 4.4: Desugaring a variational if statement, adapted from Linux v2.6.33.3 drivers/input/mousdev.c.

4.3.1 Transforming Preprocessor Directives

The main idea of VarAlyzer's transformation is that the static preprocessor conditionals are automatically replaced with runtime C conditionals. The key challenge is that preprocessor conditionals may appear around any arbitrary set of C tokens, irrespective of C's syntax [EBN02, LKA11], while C conditionals may only appear around complete statements. For instance, in Figure 4.3a, preprocessor conditionals appear around a declaration (Lines 2–3) and a function definition (Lines 6–7) of the same name. While the preprocessor technically has a language distinct from pure C, we take the view that unpreprocessed C files are effectively written in a single, mixed language. To preserve the encoding of variability in unpreprocessed C, VarAlyzer *desugars* source files into ordinary C, which is a subset of the mixed language.

The preprocessor performs macro evaluation, header inclusion, and conditional compilation to generate C code at compile-time. With conditional compilation, the preprocessor selects which parts of the source code to send to the compiler by evaluating the values of

configuration macros passed into the preprocessor at compile-time. Developers use these preprocessor conditionals to encode variability.

Developers may wrap these conditionals around any fragment of the C code. Common patterns in real-world code include putting conditionals around entire functions, declarations, and even individual C tokens. Since preprocessing happens before parsing in the compiler, these conditionals do not need to respect C's syntax. Developers may even wrap them around incomplete C constructs, so-called "undisciplined" uses [LKA11]. Figure 4.4a is an example of this usage, where a preprocessor conditional surrounds all but the else-branch body of an if-then-else statement (Lines 2–4).

Since our goal is to preserve the behavior of these preprocessor conditionals, we need to consider their meaning when they interact with C constructs. While a preprocessor conditional has simple semantics (i.e., it conditionally includes or excludes the contained C fragment), its effect on C program behavior depends on what C constructs it surrounds. For instance, it is illegal in C's semantics to write multiple declarations of the same variable to vary its type. By surrounding these declarations with mutually-exclusive preprocessor conditionals, it is "legalized": the preprocessor only chooses one declaration to send to the C compiler. The only way to allow such multiple declarations in C is to use unique variable names. In contrast, a preprocessor conditional around a C statement behaves much like a C conditional, except that the preprocessor does not respect C's scoping rules and it takes configuration macros instead of C variables.

Phases of the Desugarer

VarAlyzer takes unpreprocessed C code, such as that in Figure 4.4a, and produces an equivalent C program using run-time conditionals to preserve variability (Figure 4.4b). There are three phases in VarAlyzer's desugarer: (1) parsing, (2) type checking, and (3) rewriting. Parsing takes the unpreprocessed C code and produces an AST that preserves all preprocessor behavior. Type checking collects symbols and their types across all variations of the SPL. Rewriting emits ordinary C code that corresponds to the unpreprocessed C constructs.

Parsing. For parsing, we reuse an existing parser, SuperC [GG12]. Unlike the standard C preprocessor and parser, SuperC solves the problem of parsing all variations of a C file. It provides a complete solution to parsing C syntax even when mixed with any combinations of preprocessor usage. Eschewing incomplete heuristics, SuperC's parsing formalism enables comprehensive parsing of unpreprocessed C, supporting complicated and even pathological cases, such as conditionally-defined macros and headers, macros with incomplete C syntax, stringification and token-pasting combined with ifdefs, and more. Listing 4.2 and Listing 4.3 present two more complex examples that use a combination of some of these features. The specifics of this parser can be found in [GG12]. An overview of the possible interactions between the C preprocessor and C's language features is shown in Table 4.1. SuperC's output is a C AST that has special "static conditional" nodes that capture every possible variation of the syntax of the input source file. The parsing algorithm ensures that conditional nodes are guaranteed to appear around complete C syntactic units,

Table 4.1: Interactions between C preprocessor and language features. Reproduced from [GG12].

Language Construct	**Implementation**	**Surrounded by Conditionals**	**Contain Conditionals**	**Contain Multiply-Defined Macros**	**Other**
Lexer					
Layout	Annotate tokens				
Preprocessor					
Macro (Un)Definition	Use conditional macro table	Add multiple entries to macro table		Do not expand until invocation	Trim infeasible entries on redefinition
Object-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions		Expand nested macros	Get ground truth for built-ins from compiler
Function-Like Macro Invocations	Expand all definitions	Ignore infeasible definitions	Host conditionals around invocations	Expand nested macros	Support differing argument numbers and variadics
Token Pasting & Stringification	Apply pasting & stringification		Host conditionals around token pasting & stringification		
File Includes	Include and preprocess files	Preprocess under presence conditions		Hoist conditionals around includes	Reinclude when guard macro is not false
Static Conditionals	Preprocess all branches	Conjoin presence conditions		Ignore infeasible definitions	
Conditional Expressions	Evaluate presence conditions			Hoist conditionals around expressions	Preserve order for non-boolean expressions
Error Directives	Ignore erroneous branches				
Line, Warning, Pragma Directives	Treat as layout				
Parser					
C Constructs	Use FMLR Parser	Fork and merge subparsers			
Typedef Names	Use conditional symbol table	Add multiple entries to symbol table			Fork subparsers on ambiguous names

```
1  #include <assert.h>
2  #include <stdio.h>
3
4  #if DEBUG && WARN_LEVEL == 0
5  #define WARN_IF(EXP) \
6    do {               \
7      assert(!(EXP));  \
8    } while (0)
9  #elif DEBUG && WARN_LEVEL > 0
10 #define WARN_IF(EXP)                              \
11   do {                                            \
12     if (EXP)                                      \
13       fprintf(stderr, "Warning: " #EXP "\n"); \
14   } while (0)
15 #else
16 #define WARN_IF(EXP) while (0)
17 #endif
18
19 int main() {
20   int x = 2;
21   WARN_IF(x == 2);
22   return x;
23 }
```

Listing 4.2: A combination of stringification (or stringizing) of expressions, (function-like) macros and ifdefs. Modified example reproduced from [Onl21].

even when the unpreprocessed input file does not, by duplicating any tokens needed to comphrehensively represent all variations of the nearest ancestor construct. For instance, Figure 4.4a's AST will have a static conditional node with two branches, one for MACRO and the other for !MACRO. The former branch will contain a complete if-then-else statement with no other static conditionals inside and the latter will have a single assignment statement.

Type checking. Traditionally, type checking ensures the absence of type errors at runtime. VarAlyzer, however, relies on the type checking phase to enable desugaring. To emit C code equivalent to the unpreprocessed C, the desugarer needs to know what variables have been declared (or left undeclared) in all the variations of the source code. As with typical C type checking, we maintain *a symbol table* and apply *C type checking rules with semantic actions* during parsing. A symbol's entry in the table, however, depends on what variation we are analyzing. For instance, in Figure 4.3a, declarations of `f` (Lines 3 and 7) have incompatible type qualifiers (`extern` and `static`). However, these two declarations can never appear in the same variation. VarAlyzer's type checker needs to track both types throughout the source file.

The symbol table binds a symbol to all of its possible types across all variations of the source code. The binding is akin to a "variational set" [WKE+14], where each type element is tagged with configuration information. The set also includes special entries to record the conditions under which the symbol is undeclared or has a type error in its declaration.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  // contents of state.def
5  // #ifndef STATE_SELECT
6  // #define STATE_SELECT(NAME, VALUE)
7  // #endif
8  // STATE_SELECT("First", First)
9  // STATE_SELECT("Second", Second)
10 // STATE_SELECT("Third", Third)
11 // STATE_SELECT("Fourth", Fourth)
12 // #undef STATE_SELECT
13
14 enum state {
15 #define STATE_SELECT(NAME, VALUE) VALUE,
16 #include "state.def"
17   Invalid
18 };
19
20 enum state strToState(const char *str) {
21 #define STATE_SELECT(NAME, VALUE)              \
22   if (strncmp(str, NAME, strlen(NAME)) == 0) { \
23     return VALUE;                              \
24   }
25 #include "state.def"
26   return Invalid;
27 }
28
29 char *stateToStr(enum state s) {
30   switch (s) {
31 #define STATE_SELECT(NAME, VALUE) \
32   case VALUE:                     \
33     return NAME;                  \
34     break;
35 #include "state.def"
36   case Invalid:
37     return "invalid state!";
38     break;
39   }
40 }
41
42 int main() {
43   enum state s = strToState("First");
44   printf("%s\n", stateToStr(s));
45   printf("%s\n", stateToStr(Second));
46   return 0;
47 }
```

Listing 4.3: Code generation using the preprocessor as often used in C++ to preserve type safety when dealing with enumerations.

This is necessary because a typical type checker will use the absence of a binding to mean undeclared and will simply halt on a type error. When desugaring, only some of the variations of the source file may have an undeclared symbol or other type errors. We continue to desugar any valid configurations instead of halting. Our type checker, in effect, tracks types in all variations of the source code simultaneously.

For instance, the symbol table entry for `f` in Figure 4.3a contains a set with four elements, one for each possible variation of this source code. `f` is undeclared if *both* M1 and M2 are undefined. `f` is a redeclaration type error if both M1 and M2 are *defined*. There are two more entries for the valid type declarations, which happen when only one of M1 and M2 is defined, but not both. The resulting symbol table entry for `f` is as follows:

$$\texttt{f} \mapsto \begin{cases} \texttt{extern void} & \textbf{if } \texttt{M1} \wedge \neg\texttt{M2} \\ \texttt{static void} & \textbf{if } \neg\texttt{M1} \wedge \texttt{M2} \\ \text{<ERROR>} & \textbf{if } \texttt{M1} \wedge \texttt{M2} \\ \text{<UNDECLARED>} & \textbf{if } \neg\texttt{M1} \wedge \neg\texttt{M2} \end{cases}$$

Rewriting. The rewriting phase produces ordinary C code that preserves the behavior of the unpreprocessed source file. The underlying parser of VarAlyzer ensures static conditionals are lifted around only complete C syntax, i.e., syntactic lifting, but our rewriter still needs to consider the behavior of static conditionals on those C constructs. When a static condition surrounds a construct, VarAlyzer lifts the construct's semantic value to the nearest ancestor that is a statement, declaration, or function definition, if not already one of these. This step ensures that VarAlyzer will output valid C code by only inserting C conditional around complete statements.

The rewriting rules depend on what C construct a preprocessor conditional surrounds: statements, declarations, etc. In general, statements are surrounded by a C conditional and configuration macros are transformed to C constant variables. Figure 4.4b shows the result of desugaring Figure 4.4a. Recall that the parser ensures that the static conditionals appear around a complete if-then-else statement and a complete assignment statement. The desugarer declares a new C constant called MACRO on Line 1, and then emits a C conditional that uses this variable around the two complete C constructs. Notice that any tokens shared by the two complete constructs are duplicated under the C conditional, which provides guarantees of "disciplined" uses of conditionals.

Declarations and function definitions cannot be desugared by surrounding them with a C conditional, since they are not statements. VarAlyzer handles multiply-declared symbols by emitting all declarations unconditionally, resolving name clashes by renaming the symbols. VarAlyzer preserves variability at runtime by instead emitting C conditionals *where the symbols are used in statements*. In Figure 4.3b, VarAlyzer creates fresh identifiers for the two declarations (Lines 3 and 5). The *usage* of the symbol `f` is replaced with a C conditional (Lines 7–8) and the mutual exclusion of the two declarations in different configurations is preserved in Lines 9–10.

The type checking phase is instrumental in VarAlyzer. It records all variations of the original symbol, which enables VarAlyzer to assign a fresh name to each of the

```
1  struct {
2    int x;
3  #ifdef MACRO
4    int y;
5  #endif
6  } var;
7  var.y;
8
9
```

(a) Before

```
1  const bool MACRO;
2  struct {
3    int x;
4    int y;
5  } var;
6  if (MACRO) {
7    var.y;
8  } else {
9    __type_error();
10 }
```

(b) After

Figure 4.5: Desugaring variational if statement. Adapted from Linux v2.6.33.3 drivers/input/mousdev.c.

variational set's entries, e.g., `__f_1` and `__f_2` in Figure 4.3b. In addition, the type checker records which configurations have type errors. Type errors are normally emitted at compile-time. VarAlyzer, however, cannot halt with such errors when only some variations have them. Instead, it preserves type errors as runtime errors, by transforming them into calls to a specially-defined `__type_error` function that always halts. In Figure 4.3b, VarAlyzer preserves the type error with Line 10, reflecting the fact that there is no declaration of `f` when macros `M1` and `M2` are both undefined and a conflicting declaration if both macros are defined. The subsequent analysis can then rule out invalid configurations as unreachable code, avoiding the imprecision by analyzing these configurations.

Desugaring C Type Specifications

While duplicating multiply-declared symbols is sufficient for variables and functions, C also supports user-defined types via typedefs, structs, unions, and enums. The latter three can also appear within declarations. The declaration in Figure 4.5a declares `var` to be a new type `struct s`. Structs and unions contain field declarations which themselves may contain struct and union definitions. A naive desugaring could take all combinations of struct/union definitions and emit each one as a separate declaration in the output C program. Real-world SPLs, however, may have highly-configurable structs, where some fields only appear in certain variations. Struct fields may also be declared using highly-configurable structs, further exploding the possible combinations of declarations.

In addition, C allows forward references to type definitions under certain conditions, which originally made one-pass compilation possible. A struct, for instance, may be referenced in a declaration of a variable before the struct itself is defined, at least in the global scope. Handling forward references would require multiple passes of the AST, making a complete desugaring not possible in a single pass.

To solve these problems, VarAlyzer handles type definitions separately from variable and function declarations. In addition to storing type declarations in the symbol table, as with typical C type checking, we maintain a separate table for struct, union, and enum type

definitions. This table collects all possible field variations (or enumerators) for each type definition, regardless of where in the scope they are defined. As with the symbol table, we are tagging each field definition with a logic formula describing which variations contain the particular field. Then, before emitting the desugared contents of each static scope, we emit a single declaration of the struct, union, or enum containing all possible fields or enumerators. When a struct variable accesses its field, we emit runtime checks for type errors.

For instance, in Figure 4.5b, the resulting desugared struct definition contains both the `x` and `y` fields, because there is no language construct in pure C for defining conditionally-defined structs. But `y` is only meant to be defined under configurations that have `MACRO` enabled. Since the desugarer's struct symbol table tracks the configurations under which each field is defined, the desugarer accounts for the configuration where fields are *accessed*, rather than where they are defined. For example, in Figure 4.5b the desugarer has transformed the access of field `y` to a C conditional (Lines 6–10) that covers both possible variations of the struct. The first branch of this conditional covers configurations where `MACRO` is enabled and therefore the field `y` exists (Line 7). The else branch accounts for all other configurations, where accesses to `y` are type errors, since the field is not defined those configurations. The desugarer preserves this type error as a run-time error with a call to a specially-defined function on Line 9.

Forward references to structs, unions, and enums require further special handling in order to desugar in a single pass. Since VarAlyzer does not know yet what all fields or enumerators of the type will be, it instead emits a fresh type name for the forward reference. Once it has collected all fields for a given type at the end of the static scope, it emits a definition of the fresh forward reference type that contains a field for each definition of the type symbol.

Desugaring Function Definitions

C function definitions combine a type declaration of the function name with a compound statement for its body, so VarAlyzer needs to both preserve all variations of the function in its symbol table and emit all variations of the function's body. VarAlyzer uses its variation-preserving symbol table to hold function symbols, while the function body is transformed like any other compound statement using C conditionals to preserve variations in statements.

As with declarations, a function with multiple variations of its type is desugared into multiple function definitions to reflect each variation. Any calls to the original function name are replaced by all renamed variations of the function, as long as the function type matches the type at the call site. All top-level declaration and definitions in a C file are global and externally-linked by default, unless specified otherwise with the `static` keyword. Therefore, any renaming at the global scope affects the symbols exported for linking by the compiler. Since C does not provide language constructs for defining modules, it relies on the underlying system's object file format, linker, and build system to coordinate interfaces between C source files. In this work, we focus on desugaring variability encoded by the

preprocessor within a C file and leave the support for build system and linker variability as future work.

Instead, we assume a project only exports one type per global symbol, emitting a type warning when a global symbol has multiple, incompatible type declarations. Each C file that uses functions defined externally needs a copy of the external functions' declarations, typically provided in a shared header file that developers copy into the source file using a preprocessor **#include** directive. It is then up to the compiler to produce an object file with a linker table that maps global functions and variables to either their addresses in the object file or to a placeholder. The linker can then automatically match undefined symbols from one object file with its definition in another, as long as the developer has properly defined the build sequence with, for instance, a Makefile.

If a globally-defined symbol's declaration depends on what variation of the program is being compiled, i.e., it is affected by preprocessor conditionals, then preserving all variations of the SPL requires modeling the behavior of the linker across all variations. Such a variation-preserving linker would need to record all renamings of multiply-declared global symbols and resolve these across all C files that comprise the project. This resolution, in turn, depends on knowing what C files are to be linked during the build of the project, information that is only captured in Makefiles or whatever build automation, if any, a project uses. In this work, we focus on desugaring variability encoded by the preprocessor in C files and instead assume a project only exports one type per global symbol, emitting a type warning when a global symbol has multiple, incompatible type declarations.

Limitations of the Transformation

VarAlyzer's transformation part is generally complete and supports the full (mixed) C language. However, we discuss some fundamental challenges that we discovered while pursuing this research in what follows.

While VarAlyzer translates variability encoded in the preprocessor, large-scale projects also encode variability in other parts of the system software toolchain. All top-level declaration and definitions in a C file are global and externally-linked by default, unless specified otherwise with the **static** keyword. Object files act as modules that import and export these external symbols used in other object files. The definitions of these external symbols can vary based on configuration options, which introduces variability in the linking process. VarAlyzer leaves the difficult problem of supporting variability-aware linking and build automation to future work and focuses on the variability within C files.

Real-world software often includes dozens or hundreds of header files for the C standard and additional libraries. As shown in Section 4.2, such headers may add hundreds or thousands of function declarations and macro definitions to a C file. These function declarations and macro definitions have to be processed over and over again for each C file that includes the respective headers. Since these header files themselves also encode variability to support different operating systems, various compiler versions, and programming languages (C vs. C++), they currently still pose a scalability challenge to VarAlyzer. Tackling this scalability issue for the transformation requires numerous technical details and implementation tricks that are out of scope for this piece of research and require thorough

Table 4.2: Preliminary transformation times for transformations that use partially preprocessed system headers.

Program	Runtime in seconds	#Source files	#Configuration variables
axTLS	302	28	94
Toybox	586	230	316
BusyBox	484	554	998

descriptions on their own. One particular compelling idea is to partially preprocess system headers for specific system configurations to counteract unnecessary processing of these headers at each place they are included. In another branch of research, we have started to implement this idea and have since then be able to successfully transform larger programs such as BusyBox, Toybox and axTLS. Table 4.2 shows some preliminary results for these programs and should give a first impression in which order of magnitude realistic programs can be transformed.

We support variability across application configurations, but assume a single system configuration. System configuration macros provide several challenges for desugaring variability; they require supporting the header file differences between multiple operating systems, multiple (versions of the same) compiler(s), multiple versions of system libraries, etc. These differences not only cause the number of possible configurations to explode, even when the application code's behavior does not depend on them, but they also impose foundational challenges. VarAlyzer cannot leave these system configuration macros unresolved during transformation since the transformed code could then not be compiled to an intermediate representation for analysis. However, resolving these system configuration macros requires information on all possible operating systems, system libraries, etc. which can hardly be obtained, if at all. And even if it could be, a software product line could not be compiled to an intermediate representation since the environment and the compiler used to produce the intermediate representation of the machine on which the transformation takes place are fixed. In addition, SuperC's underlying parser is based on one particular version of C as implemented by GCC. Supporting multiple versions of compilers would require constructing a superset of all C variations, a daunting and potentially infeasible task.

4.3.2 Variational Data-flow Analysis

We next explain how VarAlyzer makes the analysis variability-aware. This allows one to compute, for all configurations at the same time, results that pinpoint under which configurations they are valid.

VarAlyzer accepts as an input *any* given distributive data-flow problem encoded within IFDS [RHS95] or IDE [SRH96], and transforms it into a *variational version* of the problem which can then be solved on an SPL that has been desugared according to Section 4.3.1. Because IFDS problems can be encoded within IDE by using edge functions that operate on the binary lattice $V = ^{\top}_{\bot}$ [SRH96], we continue by presenting how we model general IDE problems in a variability-aware manner.

VarAlyzer builds on SPLlift's idea to make use of IDE's edge functions to encode all variants of possible data flows a SPL might induce. SPLlift, however, only allowed "lifting" *IFDS*-based analyses. As mentioned earlier, this precludes an efficient encoding of any problem with a large or even infinite abstract domain, e.g., typestate analysis and constant propagation. To efficiently compute on such large (or infinite) domains, we must instead encode the computation within the edge functions of the IDE framework, but it means that the value computation already occupies the edge functions. Therefore, we then cannot use the edge functions (directly) to capture an SPL's variability information. To be able to solve general IDE problems that already use the edge functions for computing while still capturing an SPL's variability, we need to solve two different value computation problems using IDE's edge functions.

VarAlyzer thus lifts edge functions of the user-defined IDE problem by extending their value domain V_u to produce lifted edge functions that operate on the cartesian product domain $V_l = C \times V_u$, where C is the domain of feature constraints used to describe the variability induced by the preprocessor. This enables VarAlyzer to solve both value computation problems at once, relating analysis results to the exact feature constrains under which they hold. A lifted edge function $\hat{e} : C \mapsto V_u$ is thus a mapping from edge functions that describe the feature constraints to the respective user-defined edge functions that specify the value computation problem that is valid under the associated constraint. Whenever a reachability check is performed on the exploded super-graph that has been produced by the lifted analysis problem P_l, the analysis computes the values specified by the user edge functions and the corresponding constraints that are associated with those values. The result for each reachability check of an ESG node (s, d) for a given statement s and data-flow fact d, i.e., the evaluation of a lifted edge function, is a mapping from feature constraints to their corresponding value $\{c_i \mapsto v_i\}$. In the following, we describe this lifting in more detail. Note that our solution is fully transparent: VarAlyzer can automatically lift any IFDS/IDE analysis problem pre-defined for C programs to software product Lines without having to change a single line of code.

Automated Lifting of Edge Functions

The IDE algorithm is guided through the program using its inter-procedural control-flow graph (ICFG). VarAlyzer operates on a variability-aware version of the control-flow graph $ICFG_v$. The $ICFG_v$ respects the encoding of preprocessor directives as presented in Section 4.3.1. Preprocessor symbols are modeled as **extern** global variables that follow a special naming convention. An $ICFG_v$ can be queried for those global variables and their usages. Any statement that directly interacts with one of those global variables through a def-use chain has been artificially introduced by the code transformation. This allows us to distinguish between any ordinary statement s_u that originates from the user program and any statement s_p that is generated by VarAlyzer transformation parts, originating from preprocessor directives (PPDs).

Initially, i.e., at lifting-time, a lifted edge function $\hat{e}$ maps exactly one edge function that describes a feature constraint to an edge function that, in turn, represents a (user-defined) value computation for a given statement under analysis. The lifting process is depicted

Figure 4.6: Lifting of edge functions for an ordinary user statement s_u (left) and a branching statement s_p^b that originates from a preprocessor directive (right). For the statement s_u, the user edge function is queried and results in $\lambda x.x + 42$. Because the statement has no effects on the preprocessor constraints, the edge function for the constraint domain is modeled as identity. For s_p^b, the user edge function is modeled as identity because it has no effects on the user's value computation. However, it extends the domain with edge functions that add the preprocessor feature-constraints F and $!F$, respectively.

in Figure 4.6. *Ordinary* statements s_u have no effect on the presence of a certain feature. VarAlyzer thus lifts its user-defined edge function e_u to $\hat{e} := (\lambda c.c) \mapsto e_u$. Here, the identity edge function $\lambda c.c$ over constraints expresses that the feature constraint is not altered. The statement's original flow function (opposed to edge function) remains as is.

For statements s_p that are *generated from the preprocessor directives*, the analysis can safely ignore the non-branching statements since they have been artificially introduced by the transformation and must have no effect on the user-defined value computation. For these statements, VarAlyzer applies the identity flow and edge function. For each generated *branching* statement s_p^b that originates from a preprocessor directive, VarAlyzer produces the corresponding edge function $\hat{e}$ by conjoining the feature constraint F specified by the respective preprocessor conditional with the incoming constraint c, i.e., $\hat{e} := (\lambda c.\ c \wedge F) \mapsto (\lambda x.x)$. Here on the right-hand side, we use the identity edge function $\lambda x.x$ because the statement does not influence the user-defined value computation.

Operations on Lifted Edge Functions

To allow for the construction of the exploded super-graph, edge functions need to support the following four operations:

The *composition* operation ($\circ$) composes two edge functions e and f. This operation is used to extend an edge function e and is required to construct the so-called *jump functions* (summaries) that describe the effects of *sequences* of code. An example is shown in Figure 2.7. The edge functions e, f, and g can be composed to produce the jump function $i = g \circ (f \circ e) = \lambda x.x + 2 \circ (\lambda x.1 \circ \bot) = \lambda x.3$, which describes the value computation problem for variable a from Line 1 to after Line 3.

The *join* ($\sqcup$) operation is applied when two paths in the exploded super-graph lead to a common ESG node and the respective edge functions must be combined, for instance, as a result of branching. Consider the example in Figure 2.7: the two jump functions i and j are joined to produce the new function $k = i \sqcup j$ that describes the value computation problem

for variable a from Lines 1 to 5. An *equals* (=) operation, comparing two edge functions for equality, is required to update jump functions efficiently within the IDE algorithm, and to ensure termination.

Once an ESG, i.e., all jump functions, is constructed, the value computation problem that is specified by the jump functions can be solved for any given ESG node by simply applying these jump functions. Practical implementations usually do not construct and store the complete ESG but rather only maintain the essential jump functions. To determine the possible value that may be printed in Line 6 of Figure 2.7, one *evaluates* ($\hookrightarrow$) the respective jump function k. The analysis finds that *any* value may be printed as a result of $\hookrightarrow k = i \sqcup j = \top$.

We next show how to define these four operations for the *lifted* edge functions that operate on the extended user domain $V_l = C \times V_u$ such that a transformed problem P_l can be solved by the IDE algorithm.

Join. To join information that is obtained along two (or more) different paths in the ESG, a binary *join* operation is required, see Definition 1. An example of the *join* operation is shown in Figure 4.7. When joining, we wish to join also user-defined edge functions for such constraints that are equal along both branches, as these cases relate to identical feature configurations. Hence the edge functions c_1, c_2 that describe the constraints of the lifted edge functions to be joined are compared pair-wise. If $c_1 = c_2$, their corresponding user edge functions u_1 and u_2 are joined. This situation is depicted in the left-hand side graph of Figure 4.7. Else if $c_1 \neq c_2$, both results are simply joined by set union, retaining all information about the varying constraints. The latter situation is shown in the right-hand side graph of Figure 4.7.

Definition 1. $\sqcup$*: Let* $\hat{e} = \{c_e^i \mapsto u_e^i\}_{i=0}^n$ *and* $\hat{f} = \{c_f^j \mapsto u_f^j\}_{j=0}^m$ *be two lifted edge functions. We define the* join *operation as:*

$$\hat{e} \sqcup \hat{f} := \bigcup_{\substack{(c_e \mapsto u_e) \in \hat{e}, \\ (c_f \mapsto u_f) \in \hat{f}}} \begin{cases} \{c_e \mapsto u_e \sqcup u_f\} & \text{if } c_e = c_f \\ \{c_e \mapsto u_e, c_f \mapsto u_f\} & \text{otherwise} \end{cases}$$

Composition. Definition 2 defines the composition operator for lifted edge functions. The program's control can flow only along the two functions' respective program statements when the preprocessor directives that guard these statements are both enabled. Hence, the *compose* operator conjoins the respective feature constraints. The user-defined edge functions meanwhile are composed using their own original composition operator. Whenever the composition operator is applied, one of those edge functions comprises exactly one map entry and the other one may comprise one or more map entries due to potential prior applications of the *join* operation. Those two possible situations for the *composition* operation are shown in Figure 4.8. The left-hand side graph of Figure 4.8 shows the composition of lifted edge functions for non-branching code. In this case, the edge functions c_1, c_2 that describe the constraints and the user edge functions u_1, u_2 must be composed with each

Figure 4.7: Join of lifted edge functions that have been computed along different control-flow edges. Individual edge functions are denoted by straight arrows (⟶). Jump functions are denoted by dashed arrows (⇢). The graph on the left depicts the situation when two lifted edge functions must be merged whose constraints are equal. In this case, their user edge functions must be joined. In case the constraints are not equal, they must be left unmerged as two separate pairs of edge functions.

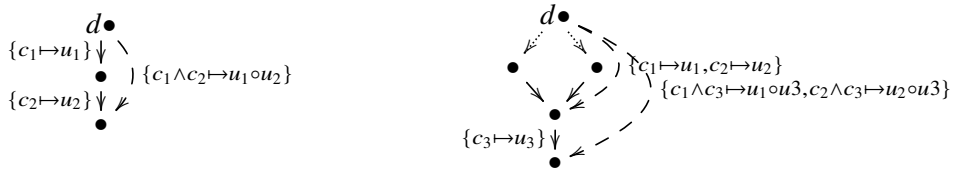


Figure 4.8: Composition of lifted edge functions. The left-hand side graph shows the composition of lifted edge functions for non-branching code. The join of two lifted edge functions at merge points may produce a new edge function that comprise multiple edge function pairs that need to be composed with the edge function of the next common successor statement. This situation is depicted in the right-hand side graph.

other. As the join of two lifted edge functions at merge points may produce a new edge function that comprises multiple map entries $\{c_1 \mapsto u1, c_2 \mapsto u_2\}$ that need to be composed with the lifted edge function $\{c_3 \mapsto u_3\}$ of the next common successor statement, a pairwise composition must be applied. This situation is depicted in the right-hand side graph of Figure 4.8.

Definition 2. ∘*: Let* $\hat{e} = \{c_e^i \mapsto u_e^i\}_{i=0}^n$ *and* $\hat{f} = \{c_f^j \mapsto u_f^j\}_{j=0}^m$ *be two lifted edge functions. We define the* compose *operator as:*

$$\hat{f} \circ \hat{e} := \bigcup_{(c_e \mapsto u_e) \in \hat{e}, (c_f \mapsto u_f) \in \hat{f}} \{c_e \wedge c_f \mapsto u_f \circ u_e\}$$

Equality. In addition, the IDE algorithm needs to be able to check for equality of two edge functions. Since we maintain the feature constraints in normalized form, we are able to define two edge functions to be equal if they are equal structurally.

Evaluation. Once an exploded super-graph has been constructed, the solver evaluates the value-computation problems described by the jump functions. The value for each ESG node (s, d) that is reachable from the tautological Λ fact is computed by evaluating its associated jump function. We define the unary *evaluate* operation in Definition 3. The evaluation operation of a jump function applies the constraint and user edge-function components of each map entry to the tautological constraint $true$ and the bottom element $\bot$ of the user-defined problem, respectively. The result is a map of values that the data-flow fact d can assume, each of which is associated with the feature constraint that encodes the set of configurations under which d holds.

Definition 3. $\hookrightarrow$*: Let* $\hat{e} = \{c_e \mapsto u_e\}_{i=0}^{n}$ *a lifted jump function. We define the unary* evaluate *operator* $\hookrightarrow$ *as:*

$$\hookrightarrow \hat{e} := \{\ c_e^i(true) \mapsto u_e^i(\bot)\ \}_{i=0}^{n}$$

Why IDE Is the Ideal Framework of Choice

While VarAlyzer supports IDE, and not only IFDS, IDE still has the restriction that flow functions and edge functions must distribute over the merge operator. The advantage of using such a *distributive* analysis framework to solve data-flow problems on SPLs is that this allows merging variability information directly at each control-flow merge point, *without loss of precision*. This is because for any flow function f and any two abstract domain values x and y of a distributive analysis problem, by definition it holds that $f(x) \sqcup f(y) = f(x \sqcup y)$. As a result, the meet-over-all-paths solution, which is undecidable in general, can be efficiently computed within such frameworks through the maximal-fixed-point solution [Bod18]. This solution is the most precise solution possible. The use of IDE thus is guaranteed to retain full precision w.r.t. a product-based analysis on pure C code, and it guarantees an efficient handling of feature constraints because they are merged and simplified at the earliest opportunity. In result, IDE is the most expressive framework that one can choose without jeopardizing efficiency.

Our idea of capturing variability by using a transparent extension of the user's analysis domain could theoretically also be applied to non-distributive problems. However, this would sacrifice precision and, due to missing summarization capabilities, would likely be prohibitively expensive for any real-world application.

4.4 Implementation

We implemented VarAlyzer on top of the SuperC [GG12] parser and the PhASAR [SHB19] static analysis framework. SuperC supports Bison-style grammars [Bis20] for implementing language processors, and automatically parses all variations of a SPL. C constructs with multiple variations due to `#ifdef`s are combined with a static choice tree node that captures each variation and its condition as represented with a logical formula.

VarAlyzer uses SuperC's existing C grammar and implements the desugarer using semantic actions. A semantic action defines a snippet of code to be executed after each

language construct and produces a semantic value for that construct. VarAlyzer records all variations of a construct's desugaring transformation, along with each static condition, as the semantic value of the grammar production. The semantic actions are executed bottom-up, and VarAlyzer gradually constructs the complete, desugared version of the input program by combining the desugared child constructs into larger constructs until reaching the top of the grammar.

VarAlyzer preserves semantic preprocessor information using calls to artificial function headers. Type errors, caused by invalid configurations, are transformed into runtime function calls. VarAlyzer makes the information on symbol renaming available by introducing a symbol table. For each compilation unit, it emits a definition of a static initializer function that specifies the renaming using a function call for each renamed symbol. The static initializer function can be thought of the compilation unit's initializer, because it has no other runtime behavior. The static conditional variables are declared as global boolean variables, since preprocessor macros have no scope and are project-wide. We model preprocessor conditionals using logic formulas and emit a mapping that associates the conditional variables with their respective textual Z3 [dMB08] solver representation using function calls within the initializer function.

VarAlyzer implements the variational analysis presented in Section 4.3.2 on top of the PhASAR [SHB19] framework. VarAlyzer provides a wrapper type that can be wrapped around any of PhASAR's IFDS *and* IDE analyses. The wrapper type wraps the regular user-defined edge functions in a special variability-aware edge function that supports the required operations as described in Section 4.3.2. Before VarAlyzer starts the actual analysis at the given entry points on the given target code, it analyzes the aforementioned static initializer function and retrieves the symbol table as well as the preprocessor conditionals. It then decodes the textual Z3 [dMB08] solver representations of the preprocessor conditionals into their corresponding in-memory `z3::expr` representations, which the analysis uses as part of its lifted edge-function domain. After construction, the variability-aware edge functions are passed to the data-flow solver. The solver follows the control flow of a variability-aware, LLVM-based ICFG implementation that is able to distinguish ordinary instructions from instructions that originate from the preprocessor and have been artificially generated by VarAlyzer's SPL-transformation part. Once the exploded super-graph is built, the IDE solver solves the value-computation problems, thereby also collecting and computing the feature constraints that are associated with each of the original user-defined edge functions and their respective evaluations.

4.5 Experiments

Our empirical evaluation addresses the following research questions:

RQ_4	*Does VarAlyzer produce results that are identical with these of a product-based analysis?*
RQ_5	*How efficient is VarAlyzer compared to a product-based analysis?*

RQ_6	*To what degree is variational analysis necessary to solve semantic analyses on VarAlyzer-transformed code?*

To address RQ_4 and RQ_5, we compiled each of our 95 benchmark subjects once using VarAlyzer's conditional compilation approach and once exhaustively using the standard compilation approach for all software products. We then subjected the resulting compiles to VarAlyzer's variability-aware analysis and a traditional product-based analysis that analyzes each individual software product, respectively. Our benchmark comprises 95 compilation units that make use of OpenSSL's EVP library. For each software product line, we compared the analysis results obtained by VarAlyzer to the results obtained by the product-based approach. We ran each compilation and analysis step five times to account for variance. To address RQ_6 and to answer the question whether variability awareness is necessary, we ran a traditional variability-oblivious inter-procedural typestate analysis encoded in IDE using PhASAR on VarAlyzer-transformed code. We parameterized the typestate analysis for three different APIs of OpenSSL's EVP library. We discuss the precision of the results produced by the traditional variability-oblivious analysis and comment on the reusability of existing static analysis infrastructure on the desugared code.

Unfortunately, comparisons of the VarAlyzer approach to existing tools such as TypeChef [KKHL10] or Hercules [Her20] are either not possible or not very meaningful as the implementations of previous works are not maintained or use different analysis techniques that do not allow one to solve more complex, inter-procedural data-flow analysis problems.

4.5.1 Experimental Setup

We have evaluated VarAlyzer using benchmark subjects consisting of 95 hand-written C compilation units ranging from 8 to 219 lines of source code that comprise correct as well as incorrect usages of OpenSSL's EVP library parts. These compilation units comprise between zero and eleven features and comprise intra- as well as inter-prodcedural usages of the EVP library. We also included compilations units with zero features to assess the potential overhead of VarAlyzer's conditional compilation and variability-aware data-flow analysis. To obtain correct API uses, we used the code examples presented in OpenSSL's wiki.[2] To ensure that our benchmark programs comprise realistic API usages, we mined 15 SPLs on GitHub using the advanced search and aimed for high-stared and popular projects that make use of OpenSSL's EVP library parts.[3] We then extracted the compilation units that comprise usages of the EVP library and used these to help modeling our benchmark. Surprisingly, several of the real-world API usages completely omit error handling. We thus also omitted error handling code in our benchmark subjects to allow for easier debugging of our transformation and analysis. We then introduced different kinds of protocol breaches, some of them unconditionally and some of them depending on certain (invalid) configurations.

To evaluate VarAlyzer, we used a client typestate analysis $\mathcal{T}$ that had been independently implemented using PhASAR's implementation of the IDE framework. To allow

[2] OpenSSL Wiki `https://wiki.openssl.org/`

[3] Github advanced search `https://github.com/search/advanced`

the analysis to validate useful typestate properties w.r.t. OpenSSL, we parameterized it for the OpenSSL EVP APIs message digests $\mathcal{T}_{MD}$, encryption/decryption (cipher) $\mathcal{T}_{CPR}$, and message authentication codes $\mathcal{T}_{MAC}$. OpenSSL's EVP functionalities provide a high-level interface to OpenSSL's cryptographic functions that are commonly used by projects that require such cryptographic functionalities.

We set up the parameterized typestate analyses to run both in a traditional, variability-oblivious manner using plain PhASAR, which we denote as $\mathcal{T}^{PSR}$, and in a variability-aware manner, which we denote as $\mathcal{T}^{VAR}$. For RQ_4 and RQ_5, we *exhaustively* sampled and compiled all concrete software products for each SPL of our benchmark to LLVM intermediate representation (LLVM IR) to run the traditional, variability-oblivious typestate analysis $\mathcal{T}^{PSR}$. To be able to run VarAlyzer's variability-aware analysis $\mathcal{T}^{VAR}$, we desugared each SPL using VarAlyzer's transformation and then compiled the transformed C code to LLVM IR. We used the standard Clang compiler to produce LLVM IR. For each matching analysis pair, e.g. $\mathcal{T}^{PSR}_{MD}$ (variability-oblivious typestate analysis parameterized for the message digest API) and $\mathcal{T}^{VAR}_{MD}$ (variability-aware typestate analysis parameterized for the message digest API), we automatically checked if the data-flow results produced by $\mathcal{T}^{VAR}$ coincide with all sampled results produced by $\mathcal{T}^{PSR}$, to evaluate the correctness of VarAlyzer's lifted analysis (RQ_4). The running times and memory usages of the two approaches are compared in RQ_5. For RQ_6, we ran the traditional feature-oblivious typestate analysis $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code and compared with its results with the variability-aware analysis $\mathcal{T}^{VAR}$ to assess $\mathcal{T}^{PSR}$'s precision.

We measured the running times and memory usages for the experiments on an Intel i7-5600U CPU@2.60GHz machine running Ubuntu 16.04 with 16GB main memory. We ran each experiment five times, removed the minimum and maximum measuring and computed the average of the remaining three values. We determined the runtimes and peak memory usages of the experiments using the UNIX `time` tool. We measured the lines of code of the compilation units using the UNIX `wc` tool. We formatted the code using the `clang-format` tool and its default settings to allow for a fair comparison of the lines of code measurement. Our benchmark programs, the raw as well as the processed data produced in our evaluation is available in our artifact [Art21].

4.5.2 RQ_4: Analysis Correctness

The PhASAR framework comprises a variety of unit tests for various different parametrizations of the typestate analysis to assess its correctness. It contains tests for parametrizations for C's file API(s) that are concerned with the type FILE, OpenSSL's secure heap and secure memory APIs as well as OpenSSL's EVP key derivation API. We developed the typestate parametrizations for OpenSSL's EVP message digest (MD), encryption/decryption (CIPHER), and message authentication codes (MAC) and manually checked their correctness on individually software products that we derived from our benchmark targets. Hence, we can ensure the correctness of the variability-oblivious typestate analysis for the parametrizations $\mathcal{T}^{PSR}_{MD}$, $\mathcal{T}^{PSR}_{CIPHER}$, $\mathcal{T}^{PSR}_{MAC}$ w.r.t. derived programs they have been tested with.

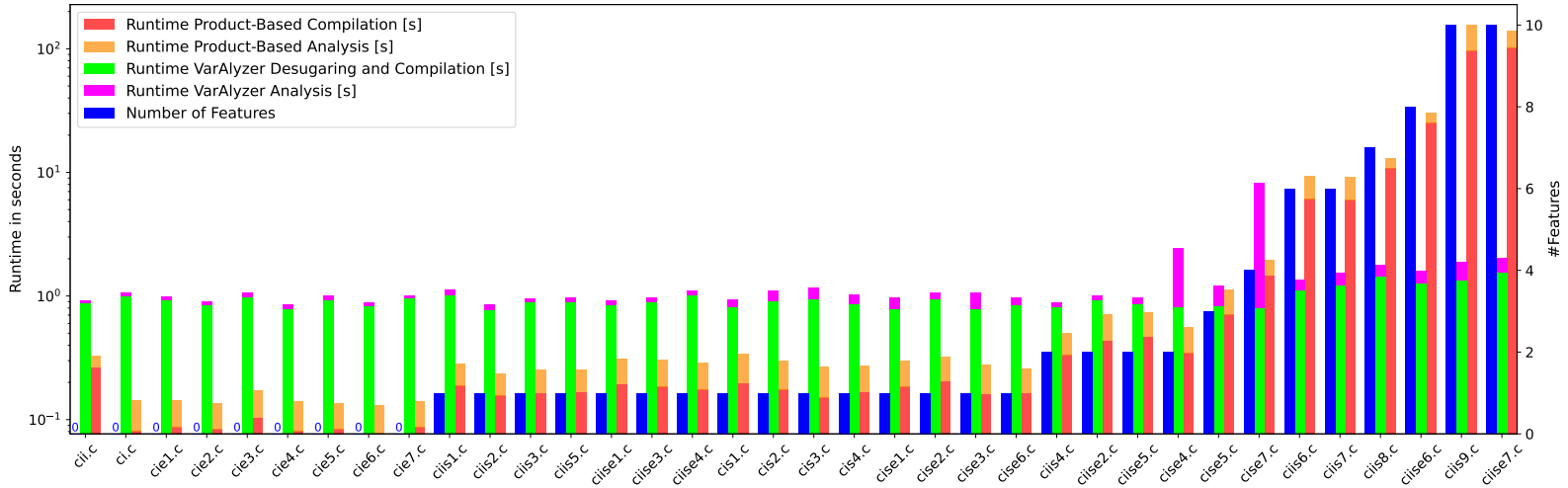


Figure 4.9: Analysis efficiency on the benchmark programs that use OpenSSL's EVP CIPHER API. Naming scheme of the benchmark targets: c{'i' - intra-, 'ii' inter-procedural}{'s' - software product line, '_' - no software product line}{'e' - erroneous API usage, '_' - correct API usage}.

VarAlyzer's process of lifting IFDS- and IDE-based analysis has been designed to be fully transparent, i.e., it does not modify the semantics of the analysis that is lifted but instead lifts its domain to make it variability-aware—allowing it to distinguish between data-flow facts that have been computed under different feature constraints.

To show that not only theoretically but also in practice VarAlyzer-lifted analyses retain precision compared to their un-lifted, product-based counterpart and also compute the results of all possible software products in a single analysis run, we wrote an automated comparison tool. The comparison tool ran our variational analysis $\mathcal{T}^{VAR}$ on each benchmark subject and then ran its variability-oblivious counterpart $\mathcal{T}^{PSR}$ on all of the exhaustively sampled concrete software products, performing an in-memory comparison of the results. The tool found that the results of $\mathcal{T}^{VAR}$ included the results produced by each analysis run of $\mathcal{T}^{PSR}$. All results, i.e., protocol breaches—on a data-flow fact-level—for each analysis run of $\mathcal{T}^{PSR}$ on a sampled software product can be found in the mapping from feature constraints to data-flow facts produced by $\mathcal{T}^{VAR}$ for the respective feature constraints that describes the software product. Besides that, $\mathcal{T}^{VAR}$ does not introduce spurious data-flow facts that cannot be found in the results of $\mathcal{T}^{PSR}$ run on any concrete software product and hence, VarAlyzer's results in fact coincide with the results produced by a product-based analysis.

Our variability-aware analysis approach produces results that coincide with the results computed using a corresponding variability-oblivious product-based analysis.

4.5.3 RQ_5: Analysis Efficiency

Section 4.5.1 presents the results concerning VarAlyzer's efficiency. Due to space restrictions we can only include the data for the analysis of the 36 benchmark programs that use the OpenSSL EVP encryption/decryption (CIPHER) API and report on the accumulated data for the remaining ones. We have made the complete results available [Art21].

Our experiments show that the standard Clang compiler requires between 0.07 and 188.1 seconds to exhaustively compile all concrete software products of a software product line in our benchmark set. VarAlyzer's two-step desugaring compilation (comprising desugaring and compilation of the desugared code) is, on average, 7.1 times faster, ranging between 0.6 and 1.6 seconds. Running the variability-aware analysis $\mathcal{T}^{VAR}$ on a complete SPL is, on average, 8.0 times faster than analyzing the target software product line using the product-based approach $\mathcal{T}^{PSR}$ that needs to analyze each software product in separate. In total, the complete variability-aware $\mathcal{T}^{VAR}$ pipeline that includes variational compilation and variability-aware analysis is, on average, 7.5 times faster than compiling and analyzing each concrete software product derived from a SPL, while the analysis's memory usage increases by a factor of 1.17.

Section 4.5.1 shows the number of features on a linear scale (the y axis at the bottom), and the accumulated compilation and analysis times of a product-based approach using plain PhASAR for analyzing all sampled concrete software products and VarAlyzer per target program concerning OpenSSL's CIPHER API on a logarithmic scale (the y axis at the top). For the benchmark targets that comprise no variability, the running times

of VarAlyzer are generally higher than those of the product-based approach. For most of the compilation units that do comprise variability, the variability-aware approach runs faster than the products-based one as soon as the target comprises more than four features with only one exception. This trend is particularly clear on programs with more features (e.g., *ciis9.c* and *ciise7.c* in Section 4.5.1). On two occasions (cise4, cise7), however, VarAlyzer's variability-aware data-flow analysis runs significantly slower than expected. We manually checked the SPLs' source code and found that they use preprocessor integer arithmetic which in our current implementation translates to a relatively long and complex Z3 constraint that slows down the analysis during constraint simplification. The running times of VarAlyzer, apart from the aforementioned two exceptions, generally remain in the same order of magnitude while those of the product-based approach clearly grow exponentially in the number of features reflecting the fact that a software product line may comprise up to $2^{\#features}$ individual software products.

In terms of code size, VarAlyzer's desugarer causes an increase in lines of code by a factor of 9.2, on average. This is mainly because the desugarer emits artificial function declarations and definitions to preserve the preprocessor's semantics. The definition of the static initializer function, which encodes the symbol table for the renamings applied by VarAlyzer and the preprocessor conditionals in Z3's textual representation as described in Section 4.4, is the main cause for the increase in code size. Each renamed symbol leads to an additional line of code, and each preprocessor conditional leads to at least one more line of code. Nevertheless, because the static initializer function is not called within the program but is only used to describe semantics, it does not affect *traditional* variability-oblivious static analyzers.

> VarAlyzer outperforms the product-based approach in all cases that comprise more than four features except for one which the current implementation cannot yet handle efficiently. The results would favor the variability-aware analysis even more with an increasing number of features. While the product-based approach requires one to compile and analyze each product, VarAlyzer only requires a single desugaring-compilation and analysis pass.

4.5.4 RQ_6: Analysis Precision

By comparing the results obtained from running the traditional variability-oblivious analysis $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code, we can assess the precision gained by making the analysis variability-aware. We manually checked the results produced by these analysis runs for the 95 benchmark subjects and observed that the over-approximation leads to great imprecision that renders the results practically unusable. This is because the analysis incorrectly introduces interaction between data flows computed within different features that—according to the preprocessor's semantics—cannot happen in any concrete product. We find that whenever an API's respective context variable is modified across multiple features that cannot actually be enabled together, the typestate analysis $\mathcal{T}^{PSR}$ directly associates those variables with an error state. We can observe that $\mathcal{T}^{PSR}$ returns the most coarse-grain analysis element for the context variables for all 68 compilation units that do

comprise variability and whose features differ in the modifications made to the context variable.

Existing analysis approaches presented in literature such as the one by Iosif-Lazar et al. [IMD+17] only employ code transformations to enable the (re)use of existing, unmodified feature-oblivious static analyzers for software product lines. However, information on variability is not preserved and even if it would be, cannot be understood by existing feature-oblivious analyzers. While this generally allows one to apply existing analyzers to entire software product lines, their results are unusable for semantic program analysis as our manual inspection of the results produced by $\mathcal{T}^{PSR}$ on VarAlyzer-transformed code shows. And while simpler, syntax-based analyses may report bugs, it is hard to impossible to account them to a specific feature combination in order to validate and action on them.

For more complex semantic analyses, variability awareness is essential to allow one to distinguish information that is obtained along different mutually exclusive features. To produce useful analysis results on software product lines, one not only requires variability awareness for the transformation, *but* also the analysis parts.

4.6 Related Work

Several previous approaches address, in part, the difficult problem of statically analyzing real-world software product lines [KGR+11, GG12, KATS12, CEW12, BRTB12, BTR+13, MDBW15, CCS+13, Dim16]. Prior work either created new analyses that had to account for the semantics of the static conditions [KGR+11, GG12, GJ05, KOE12, KKHL10] or performed limited syntactic transformations from the preprocessor into C and used off-the-shelf tools [IMD+17]. The works that lift the analysis [KGR+11, GG12, GJ05, KOE12, KKHL10] must work on the combined preprocessor/C languages which makes those harder to implement. This causes these approaches to resort to simpler analyses. The approach presented by Iosif-Lazar et al. [IMD+17] misses preprocessor semantics which passes the problem to downstream analyses. The only available data-flow analysis for software product lines written in C [LvRK+13, BRTB12] is intra-procedural only. To employ precise, inter-procedural static analysis the transformation of the preprocessor directives into ordinary C must be able to handle all of the preprocessor's constructs and, in addition, preserve full information on static preprocessor conditionals. The latter requirement is necessary to make this information available to downstream analysis to avoid a loss in precision.

SPLlift [BTR+13] avoids generating all potential software products by analyzing the entire SPL as a whole. This so-called family-based approach encodes feature constraints in distributive flow functions. SPLlift solves IFDS [RHS95] problems on SPLs using IFDS's generalization IDE [SRH96]. However, SPLlift can only solve data-flow problems with the small and finite domains, limited by IFDS. SPLlift is a prototype for a seldom-used product-line dialect of Java [KTS+09] and thus cannot be applied to real-world product lines, especially not those that use the C preprocessor.

SuperC [GG12] presents a configuration-preserving lexer, preprocessor, and parser. Its preprocessor resolves includes and macros while leaving static conditionals intact to pre-

serve its variability. A configuration-preserving parser then generates an abstract syntax tree (AST) that is additionally amended with static choice nodes to represent the static conditionals. SuperC uses a performant fork-merge parsing: it forks subparsers whenever a choice node is encountered and merges after the conditionals. The approach explores how to perform syntactic analysis of C code while preserving its variability. SuperC provides detailed insights on preprocessor usages and interactions of preprocessor usages of software product lines.

TypeChef [KKHL10] is another variability-aware parser and type-checker for product lines written in C and allows for detecting variability-induced bugs in configurable systems. It avoids combinatorial explosion by parsing the entire source code in a variability-aware fashion *without* preprocessing. Similar to SuperC, it produces an AST that captures the variability using static choice nodes. Based on TypeChef's AST, a variability-aware type system has been developed that type-checks C code with compile-time configurations. While it is possible to implement static program analyses that operate on variability-aware ASTs, those analyses would still only be syntax based and, in addition, would still need to encode the variability themselves (e.g., [LvRK+13, BRTB12]). Variability-aware control-flow and syntax-based data-flow analysis can also be implemented on top of TypeChef. However, this requires the development of syntactic AST-based analyses from scratch for the preprocessor/C language. Instead, our approach does not need to capture the static behavior. This allows us to build on existing, sophisticated program analyses; we use PhASAR and our variability-aware extension VarAlyzer.

Hercules [Her20], a rewriting and refactoring engine built on top of TypeChef, is a source-code transformation tool similar to the goal of SugarC. It transforms compile-time variability into runtime variability. It no longer relies solely on syntactic analysis only but also allows for more difficult semantic analyses as well. Hercules, however, relies on TypeChef's variability-aware parsing and analysis infrastructure which limits the application to code that is type-error-free, a requirement that real-world code does not hold. Our approach is able to pass *all* information of static preprocessor conditionals to our downstream analysis. This allows for more precise subsequent analyses. For instance, it expresses type errors as ordinary function calls, which allows its subsequent analysis to collect type errors while analyzing the program without the need to exit immediately.

Iosif-Lazar et al. [IMD+17] created C Reconfigurator that translates product lines into single programs by replacing compile-time variability with run-time variability. The resulting programs can be analyzed using traditional off-the-shelf analysis tools such as clang-tidy [Cla18c] or Frama-C [CKK+12]. However, C Reconfigurator does not preserve information on the origin of a static conditional, making the results produced by the off-the-shelf tools on the transformed code variability-unaware. Instead, our approach preserves full information on the preprocessor's semantics and can compute the analysis results and their respective variants on-the-fly in a single analysis run. C Reconfigurator also does not include feasible but invalid configurations in the transformed program, making the bugs caused by these configuration impossible to detect.

Le et al. [LP14] presented the Hydrogen framework that introduced multiversion inter-procedural control-flow graphs (MVICFGs). MVICFGs represent the control flows of multiple versions of a program in a single graph whose edges are annotated with the

version(s) under which a control flow is feasible. MVICFGs can be used for incremental update analysis and for determining the bug/patch impact for multiple program releases. The ICFGs of VarAlyzer-transformed programs can be viewed as MVICFGs with the difference that VarAlyzer's ICFGs represent all possible variants of a software product line instead of (potentially) all versions of an individual software product. While Hydrogen employs a demand-driven symbolic analysis whose queries must be parameterized with a specific version for which to compute results, VarAlyzer's lifted distributive data-flow analyzes compute the results for all possible software products in a single analysis run and accounts them to the constraints under which they are valid.

4.7 Conclusions

We have presented the design and implementation of VarAlyzer. VarAlyzer allows one to produce a configuration-preserving encoding off all variability in regular C code which it then subjects to a variability-aware, context- and flow-sensitive data-flow analysis. It enables computing precise results on entire software product lines, annotated with feature constraints that encode in which product configurations each result is valid. Our empirical study using 95 compilation units that make use of OpenSSL shows that this approach outperforms a traditional product-by-product analysis as soon as more than four products need to be analyzed. As a result, for the first time VarAlyzer allows one to conduct an effective static data-flow analysis of software product lines on real-world C code. This has the great potential to allow developers to find bugs and vulnerabilities much earlier in the development process. For instance, whereas previously developers using OpenSSL had to identify vulnerabilities separately for each concretely preprocessed variant of OpenSSL, using VarAlyzer now has the potential to allow the OpenSSL maintainers to detect such vulnerabilities for all relevant configurations ahead of time.

While now being able to inter-procedurally analyze and deal with software product lines written in C that would otherwise present an exponentially hard barrier to data-flow analysis, we next explain how we reduce analysis times to a minimum for C and C++ projects that use libraries and are organized with help of version control systems.

5 Modularity

In this chapter, we elaborate on how we scale complex static analysis to large projects that make use of libraries and are maintained under version control. In particular, we show how one can employ summarization to avoid expensive and unnecessary reanalysis of software components that do not change from one analysis run to another.

Whole-program analysis (WPA) can yield high precision, however causes long analysis times and thus, does not match common software-development workflows, making it often impractical to use for large, real-world applications.

This chapter hence presents the design and implementation of ModAlyzer, a novel static-analysis approach that aims at accelerating whole-program analysis by making the analysis modular and compositional. It shows how to compute *lossless*, persisted summaries for callgraph, points-to and data-flow information, and it reports under which circumstances this function-level compositional analysis outperforms WPA.

We implemented ModAlyzer as an extension to LLVM and PhASAR, and applied it to 12 real-world C and C++ applications. At analysis time, ModAlyzer modularly and losslessly summarizes the analysis effect of the library code those applications share, hence avoiding its repeated re-analysis. The experimental results show that the reuse of these summaries can save, on average, 72% of analysis time over WPA. Moreover, because it is lossless, the module-wise analysis fully retains precision and recall. Surprisingly, as our results show, it sometimes even yields precision superior to WPA. The initial summary generation, on average, takes about 3.67 times as long as WPA.

5.1 Introduction

Static analysis plays an important role in modern software development. While intra-procedural static data-flow analysis might only be useful in a limited number of use-cases, inter-procedural analysis is a powerful building block for bug finding [Cod18, CS18, EHMG15], compiler optimization [Onl18, ICC18] and software hardening [ARF+14, KNR+17, LL05, HREM15, HHL+17].

Static analysis is known to be an undecidable problem [Ric53], which challenges static-analysis designers to define analyses that are both precise (yielding little to no approximate information) and efficient. To obtain good precision, static program analyses need to be inter-procedural, i.e., cross procedure boundaries, and also must be context sensitive [SBL11]. Moreover, they must be based on precise points-to analyses [Bod18].

Such inter-procedural analysis, however, especially if implemented as a whole-program analysis (WPA), is notorious for causing problems with scalability in both runtime and memory consumption. The memory consumption required for larger programs to keep the

complete program representation as well as all of the data structures required to perform the analyses and optimizations in memory can easily grow to a large two-digit GB figure [Thi18, TG17]. Analysis times can amount to several hours, impeding development processes even in cases where the analysis is deployed as nightly build [LTMS18, BBC+10, SKB14].

There are application scenarios for which one can yield useful results with *intra-procedural* analyses that are simple enough to scale. The clang-tidy tool [Cla18c] and Cppcheck [Cpp18] use *syntactic* analyses that are able to analyze software comprising a million lines of code within minutes. Many *semantic* program analyses, however, such as data-flow [Kil73], typestate [Str83, SY86] or shape analyses [WSR00], for instance, require detailed program representations that incorporate the effects of procedure calls, yet are virtually impossible to scale if computed for the whole program at once. This precludes important application scenarios, for instance, IDE integration or the automated scanning of frequently changing software. As mentioned in Chapter 1, Facebook, for instance, reports that its code base changes so frequently that it has become a real challenge to design analysis tools such that they can report errors quickly enough so that they are still relevant and actionable when reported [HO18].

In this work, we aim to scale static context-, flow-, and field-sensitive inter-procedural program analysis using a compositional computation of analysis information. The effectivity of this compositional program analysis depends on the number of reusable parts of an application, e.g., program parts that constitute frameworks or libraries, or for parts that simply do not change from one analysis run to the next. A recent study by Black Duck (Synopsis) has shown that more than 96% percent of the applications they scan contain open-source components and that those components now make up, on average, 57% of the code [Sof18]. As those dependencies are updated much less frequently than application code, compositional analysis can potentially accelerate the analysis of applications by reusing analysis results from previous runs.

Previous work on compositional program analysis has been restricted to certain types of data-flow analysis only. Reviser [AB14], for instance, allows for the ahead-of-time computation of reusable taint-analysis summaries for Java libraries. Reviser builds on concepts by Rountev et al. [RSX08], who showed how to obtain reusable libraries for general distributive data-flow problems. Both those previous approaches, however, have two significant limitations: First, they only apply to Java, making it unclear which concepts carry over to other languages, particularly C and C++, which allow more liberal pointer accesses to the stack and heap. Second, they only apply to data-flow analysis and leave out the composition of points-to and callgraph information. Especially the latter is a serious practical limitation: when composing a library summary with application code, these approaches again perform an expensive whole-program points-to and callgraph analysis, which in itself can take several minutes if not hours to complete. In result, these approaches incrementalize only the tip of the proverbial iceberg. Addressing this limitation is complex as callgraph, points-to, and data-flow information are inter-dependent. A core conceptual contribution of this chapter is therefore also *a mechanism for analysis dependency management for a fully compositional analysis*. This mechanism automatically triggers updates whenever novel information becomes available that affects existing information.

An important practical factor impacting the scalability of compositional analysis is the mechanism to persist summaries. While the approach by Rountev et al. [RSX08] *computes* summaries, they are *not persisted at all* [Rou14] but rather discarded at analysis shutdown, which completely defeats their purpose. Reviser [AB14] does persist summaries, but its summary format is only applicable to taint analysis that uses a binary lattice $\begin{smallmatrix}\top\\\bot\end{smallmatrix}$. Finding an efficient summary format that is able to persist general data-flow information is challenging due to arbitrarily complex lattices used by more advanced analyses. However, efficient and generalized persistence of summaries is key to effective compositional analysis.

This chapter presents ModAlyzer, a novel approach to compositional analysis that in contrast to earlier approaches performs an *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information in a module-wise fashion. ModAlyzer allows the compositional pre-computation of all three pieces of information for individual C or C++ modules, such as libraries and frameworks. Information precomputed this way is then efficiently persisted, and later-on *merged* into larger analysis scopes. Merging analysis information efficiently is an integral part of any compositional analysis approach as combining analysis information computed on individual pieces of code is required to produce overall analysis results.

As our experiments show, this frequently helps to achieve a more efficient analysis of entire applications (compared to WPA) while retaining the same level of precision and recall of a matching WPA.

Interestingly, as this paper shows, merge operations on different types of analysis information can be modelled in a common way by defining merge operations on their respective graph representations. ModAlyzer thus conducts its compositional computation of callgraph, points-to, and data-flow information using those graph operations. While ModAlyzer compositionally computes all these kinds of information, it also manages the dependencies among them, and updates dependent information as required. ModAlyzer creates summaries for callgraph and points-to analysis, and for data-flow analyses expressed in the IFDS [RHS95] and IDE [SRH96] frameworks. Those frameworks support data-flow analyses whose flow functions distribute over the meet operator, which in turn allows for an efficient and—as we also show empirically—*lossless* summarization. ModAlyzer does not lose any information and also does not have to overapproximate missing information. Instead, it leaves gaps that will be eventually filled-in during summary application resulting in the same information that would have been obtained by a matching whole program analysis. Many useful data-flow analyses, among others taint analysis as well as all Gen/Kill problems, can be encoded in those *distributive frameworks*. ModAlyzer also allows for the computation of more expressive analyses in the monotone framework [KU77]. While one generally cannot create *data-flow* summaries for such analyses (an undecidable problem), these analyses nonetheless can benefit from summaries for points-to and callgraph information. This still allows to greatly accelerate analysis computations even for *non-distributive* analysis problems.

We have implemented ModAlyzer on top of PhASAR [SHB19] and LLVM [LA04]. We show the improvements of ModAlyzer's compositional analysis over traditional whole-program analysis by analyzing 12 real-world C and C++ applications of various sizes, reaching from 129,000 to 1,400,000 lines of code. For each application, we perform two

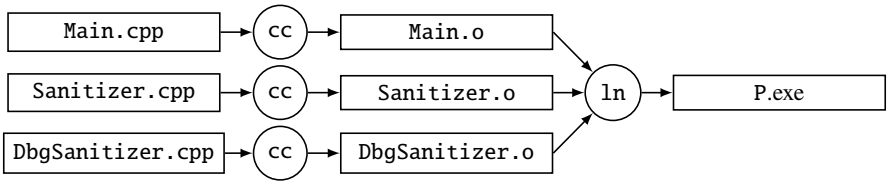


Figure 5.1: C and C++'s compilation model. `cc` is the C or C++ compiler. `ln` is the linker.

client analyses (uninitialized-variables analysis and taint analysis), once in whole-program mode and once using library summaries pre-computed by ModAlyzer. We compare the resulting running times and client reports to validate the equivalence in precision and recall, and to assess analysis time. Our experiments show that ModAlyzer can decrease the analyses' runtimes between 28% and 91% while keeping the initial one-time runtime overhead for summarization of library parts at 3.67 times as long as the cost of a whole-program analysis.

The implementation of ModAlyzer is available as open source under the permissive MIT license as part of the PhASAR framework. All accompanying artifacts of this paper, including the processed target applications, their modularizations, and result data are available online under the MIT license [Art21].

In summary, this chapter makes the following contributions: it presents

- the first *integrated compositional analysis* for callgraph, points-to and context-sensitive data-flow information with appropriate summarization techniques and summary formats,
- ModAlyzer, an open-source C++ implementation within the PhASAR [SHB19] framework, allowing the full module-wise computation of arbitrary distributive static analysis problems (and module-wise computation of points-to and callgraph information for non-distributive analysis problems),
- and an experimental evaluation of ModAlyzer, which shows that not just in theory but also in practice precision and recall are retained, and which assesses under which circumstances the reuse of summaries can decrease the overall analysis time.

5.2 Motivating Example

C and C++ programs are usually organized in several files that provide some limited form of modularity. An implementation and its corresponding header file are often referred to as a *compilation unit* or *module*. The compiler translates each module separately and thus, has only knowledge about the information contained within the module that is currently compiled. The resulting object file contains executable program code, which may, however, contain unresolved references. The linker resolves these references across two or more object files and may adds links to external libraries. The result after the linkage step is an executable program. Figure 5.1 depicts the corresponding mechanism.

```
1  int main(int argc, char **argv) {
2    auto *con = driver->connect(/* connection properties */);
3    auto *stmt = con->createStatement();
4    std::string q = "SELECT name FROM students where id=";
5    std::string input = argv[1];
6    std::string sanin = applySanitizer(input);
7    auto *res = stmt->executeQuery(q + sanin);
8    res->beforeFirst();
9    if (!res->rowsCount()) {
10     std::cout << "no record found\n";
11   }
12   while (res->next()) {
13     std::cout << res->getString("name") << '\n';
14   }
15   delete stmt; delete res; delete con;
16   return 0;
17 }
```

Listing (5.1) `Main` — Contains the main application code.

```
1  struct Sanitizer {
2    virtual ~Sanitizer() = default;
3    virtual std::string sanitize(std::string &in) {
4      if (isMalicious(in)) {  in = /*actual sanitization*/; }
5      return in;
6    }
7    bool isMalicious(std::string &in) { return /*check if malicious*/;
         }
8  };
9  std::string applySanitizer(std::string &in) {
10   Sanitizer *s = getGlobalSan();
11   std::string out = s->sanitize(in);
12   return out;
13 }
```

Listing (5.2) `Sanitizer` — A module of the sanitization library.

```
1  struct DbgSanitizer : Sanitizer {
2    bool disable = true;
3    ~DbgSanitizer() override = default;
4    std::string sanitize(std::string &in) override {
5      if (!disable && isMalicious(in)) { throw malicious_input(":'("); }
6      return in;
7    }
8  };
9  Sanitizer *getGlobalSan() {
10   static Sanitizer *s = new DbgSanitizer;
11   return s;
12 }
```

Listing (5.3) `DbgSanitizer` — A module of the sanitization library.

Figure 5.2: Modular example program

The vast majority of modern software is not written from scratch, but rather uses libraries, which enable code reuse, faster development and is less error prone [Syn18, TG17]. Thus, only a small amount of a program is actual application code and large parts are library code. Once a library has been introduced as a dependency it is rarely changed compared to the application code that uses it.

Our example program is comprised of three compilation units (CUs)—often called *modules* in the C and C++ context—`Main`, `Sanitizer`, and `DbgSanitizer` shown in Listing 5.1, 5.2, and 5.3. We omit the header files for brevity of presentation. The example program is built according to the compilation model presented in Figure 5.1.

Let us assume that `Sanitizer` and `DbgSanitizer` form a library for sanitization tasks called `libsan`. In C and C++, a library is a collection of one or more object files that have been compiled in form of an archive or shared object file. We further assume that `Main` represents the user application that makes use of the `libsan` library. We use the example program shown in Figure 5.2 as a running example to detail on our module-wise analysis (MWA) approach.

As a client analysis we use a *taint* analysis which is able to detect potential SQL injections in programs. A taint analysis tracks values that have been tainted by one or more *sources* through the program and reports a leak, if a tainted value reaches a *sink*. The analysis considers all user inputs of a program which potentially contain malicious data as tainted, e.g. the parameters **argc** and **argv** that are passed into the **main** function in our example program presented in Listing 5.1. The function **Statement::executeQuery** serves as a sink in this scenario. Without sanitization, a malicious user of the program could carefully craft the string **"1 OR TRUE;"** and pass it as the program's second command-line argument. As the input string is just concatenated the database server will return the names of all students not just the one where the id matches. By crafting such malicious inputs, a user can leak or alter the data stored in the database. A tainted value may be *sanitized* in our scenario by using the **Sanitizer :: sanitize** function (Listing 5.2) that clears malicious contents, and therefore *un-taints* a value. The client analysis $\mathbb{T}$ aims to find flows of (unsanitized) tainted values to sinks and reports a potential SQL injection vulnerability whenever it finds such an illegal flow.

5.3 Strategy

In this section, we elaborate on our compositional, *module-wise analysis*. We first present the idea of the algorithm in a nutshell and continue with our concept of summary generation. We then explain the steps we take for result merging and optimizations. As summaries are always depending on assumptions made, we discuss them at the end of this section.

5.3.1 Idea of the Algorithm

We have built our module-wise analysis approach following C and C++'s compilation model. To determine a program property of interest, a concrete data-flow analysis, the *client*, may require information from other analyses as shown in the dependency graph

in Figure 2.8 as elaborated in Section 2.3. The numbered edges in Figure 2.8 determine computation order. Since many useful static analyses can be encoded using this dependency model, we will assume such a scenario in this chapter and in Chapter 6.

To achieve fully compositional analysis information for all levels of information as shown in Figure 2.8, we must be able to (i) compute all information required for a client analysis on a function level (except the type hierarchy, which is always computed on a module level) and *summarize* them, (ii) *merge* the information and (iii) perform an *update* if a merge reveals new information that affect the current results. The merge operation combines static analysis summaries computed on two individual modules into a novel summary such that it reflects the information that would have been obtained by linking those modules first and then computing the static analysis information afterwards. In such an MWA-style analysis library modules would be analyzed separately. Their computed summaries would be merged whenever necessary while analyzing a program which uses those library modules.

As mentioned in Section 5.1, the compositional approaches to static data-flow analysis presented by Rountev et al. [RSX08] and Reviser [AB14] only apply to Java. In that regard, ModAlyzer can take advantage of C's and C++'s language characteristics, which are quite different from Java. The ModAlyzer approach merges summaries for each function per compilation unit. The intuition is that related source code often resides within the same compilation unit. Because C and C++ are often used to implement performance-critical applications [Str18, Pro18], developers have a great interest in making as much information available to the compiler as possible within an individual compilation unit. Otherwise, the compiler would not be able to perform inlining and other important optimizations in an ordinary (i.e., non-WPA) compilation setup [Mey05, Mey14]. Additionally, whereas all function members (or methods) in Java are virtual, function members are non-virtual by default in C++. It generally seems that C++ programs make use of dynamic dispatch less frequently to avoid performance penalties [DH96, CG94, AH96], a property that ModAlyzer, again, uses to our advantage. Summaries computed for C and C++ code are thus more expressive and less likely to contain *gaps* due to missing information. While ModAlyzer, in general, is applicable to other languages as well it might work better for C and C++ programs than for programs written in Java or C#, for instance, which use virtual calls all over the place. For those languages, the portion of partial summaries will increase and the overall performance of ModAlyzer will degrade as more gaps need to be closed while analyzing the "main application". Previous works by Rountev show that summarization techniques nonetheless can greatly improve running times for large Java applications, even when restricted to data-flow analysis only. We elaborate on that in detail in Section 5.5.

In the following, we show that merge operations on analysis information can be modelled in a common way through merge operations performed on their respective graph representations. However, special care must be taken to update the dependent information accordingly if new information becomes available due to merging two module summaries. This makes it crucial to keep particularly the callgraph up to date, as all other information except the type hierarchy depend on it.

5.3.2 Summary Generation

In the following, we will explain the steps of our analysis based on the example presented in Section 5.2. The assumption is that `Main` changes frequently, and `libsan` only once in a while. For presentation here we start our library pre-analysis by analyzing the `Sanitizer` module, although the analysis algorithm does not make any assumptions about module order.

Type Hierarchies

Our approach first computes the type hierarchy as it is the most robust structure in the sense that the amount of information can only grow monotonically. We use $\tau_{\mathbf{t}}$ to denote the type of a **class** or **struct** **t** and we use $T_{\mathbf{C}}$ to denote the type hierarchy for a module **C**. In addition, the type hierarchy maintains information on the virtual function tables (call targets) for C++'s *struct* or *class* types that declare virtual functions.

Example 5.3.1. *The analysis will find that the type hierarchy for the* `Sanitizer` *module consists of a graph containing a single node representing the type* $\tau_{Sanitizer}$*. The call target for* $\tau_{Sanitizer}$ *contains two entries,* { ***Sanitizer ::~ Sanitizer*** , ***Sanitizer :: sanitize*** }.[1] *The (partial) type hierarchy for the* `Sanitizer` *module is shown in Figure 5.3*

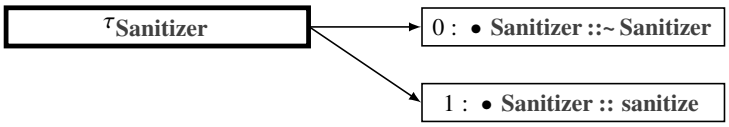


Figure 5.3: Type hierarchy and respective virtual function table(s) of the `Sanitizer` module.

Intra-Procedural Points-To Information

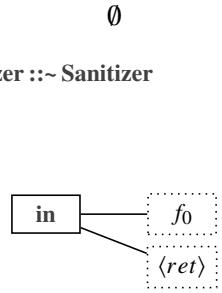
In the next step, the analysis computes function-wise, intra-procedural, never-invalidating[2] points-to information using an *Andersen* [And94] or *Steensgaard* [Ste96]-style algorithm. The points-to information computed is flow-insensitive, and we store it as graphs. These function-wise pointer-assignment graphs (PAGs) are used to resolve potential call targets at dynamic call sites. We merge those intra-procedural PAGs later to obtain inter-procedural pointer information while constructing the callgraph. We use $\pi_{\mathbf{C::f}}$ to denote a pointer-assignment graph for function **f** in module **C**. We use $\Pi_{\mathbf{C}}$ to denote a pointer-assignment graph containing all pointer-assignment graphs for module **C**.

Example 5.3.2. *For each function definition contained in the* `Sanitizer` *module a PAG is computed and added to the graph* $\Pi_{Sanitizer}$*. The* $\Pi_{Sanitizer}$ *graph containing*

[1] If a C++ type is meant to be used polymorphically, its destructor has to be declared **virtual**. Otherwise, if the static type of an object to be deleted differs from its dynamic type, the behavior is undefined.

[2] Intra-procedural points-to information is, by definition, never invalidated by additional program information from other procedures.

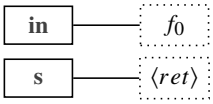
$\pi_{\textit{Sanitizer ::\sim Sanitizer}}$, $\pi_{\textit{Sanitizer :: sanitize}}$, $\pi_{\textit{Sanitizer :: isMalicious}}$, $\pi_{\textit{Sanitizer :: applySanitizer}}$ *is shown in Figure 5.4. Inter-procedural points-to relations are not followed and thus, formal pointer-typed parameters and calls to functions that return a pointer value remain unresolved and represent boundaries to the respective PAG. For instance, the pointer* `s` *in the* `applySanitizer` *function points to the return value of* `getGlobalSan` *which is indicated by a special node in the respective PAG (cf. Figure 5.4d).*



(a) $\pi_{\textbf{Sanitizer ::\sim Sanitizer}}$

(b) $\pi_{\textbf{Sanitizer :: isMalicious}}$

(c) $\pi_{\textbf{Sanitizer :: sanitize}}$; f_0 denotes the first formal parameter and $\langle ret \rangle$ the return value of the **isMalicious** function.



(d) $\pi_{\textbf{Sanitizer :: applySanitizer}}$; f_0 denotes the first formal parameter and $\langle ret \rangle$ the return value of the **getGlobalSan** function.

Figure 5.4: $\Pi_{\textbf{Sanitizer}}$ containing all pointer-assignment graphs of `Sanitizer`.

Callgraphs and Inter-Procedural Points-To Information

After having computed the function-wise pointer assignment graphs, the callgraph is constructed according to Algorithm 1, Algorithm 2 and its resolver routine shown in Algorithm 3. The same algorithm also computes points-to information across procedure boundaries. Since one cannot know upfront what library functions a user is going to call, the callgraph algorithm has to consider every externally visible function definition as a possible entry point [REH+16] (cf. Line 54 of Algorithm 1). We use $CG_{\mathbf{C}}$ to denote a (partial) callgraph of a module C. The algorithm starts at an arbitrary externally visible function f of module C. It then iterates through all call sites cs of f (cf. Line 43). We denote a call site as cs_i where i represents the line number at which the call site is found. In the following, we write $\overline{cs_i}$ for a static call site and $\widetilde{cs_i}$ for a dynamic call site at which a function pointer or virtual function member is called. In case a static call site has been detected, the algorithm adds a new callgraph edge (Line 45). In addition, for the pointer analysis, the algorithm connects the caller's actual pointer parameters and pointer return value with their corresponding formal parameters and return value of the call target using a *stitch* operation (Line 63), thus promoting (intra-procedural) pointer information to inter-procedural information. We formally define the stitch operation in Definition 4 and then discuss its use. In the latter case (Line 47), the algorithm uses points-to information provided by $\Pi_{\mathbf{C}}$ to resolve potential call targets of $\widetilde{cs_i}$ according to Algorithm 3. Starting from the function pointer that is invoked or the pointer variable of the receiver that the virtual member function is being called on at $\widetilde{cs_i}$, we search in $\Pi_{\mathbf{C}}$ for reachable functions

in case of function pointer calls (Line 76) or allocation sites in case of virtual member function calls (Line 87), respectively.

In this process, two situation may occur along with different levels of completeness of points-to information which dictate what (missing) dependencies must be tracked: *Incomplete or partially complete information:* If no functions or allocation sites are reachable yet, the reachable pointers at the function boundaries (i.e., formal pointer parameters or pointer return value of a function whose definition is missing) are marked as dependencies of $\widetilde{cs_i}$ (Line 79 and Line 86). The dependencies are maintained in a bidirectional map from dependent pointer parameters to the respective unresolved call site and vice versa. If only some functions or allocation sites are reachable but also there are some reachable pointers at function boundaries as well, then pointers at function boundaries are added to the dependencies of $\widetilde{cs_i}$ and reachable functions are added as potential call targets to the callgraph (Line 97 and 49). The edges of the callgraph are annotated with $\widetilde{cs_i}$. For virtual member function calls, the call targets of the allocated types at reachable allocation sites are inspected to find the potential targets (Line 95) which are then added to the callgraph. *Complete information:* If no boundary pointers but only functions or allocation sites are reachable starting from the pointer at $\widetilde{cs_i}$, then no dependencies must be tracked.

During the construction of the callgraph we can have situations where a pointer-assignment graph will be amended with new information. To this end, we define a first graph operation which we call *stitch* and which we use to combine pointer information at call sites.

Definition 4. *Stitch: Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$ and $e = (u, v) \notin E$. The stitch of u and v is a new graph $G' = (V', E')$, where $V' = V$ and $E' = E \cup (u, v)$. For convenience, we additionally define the function $stitch : G \times G' \times P \rightarrow G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices (u, v) with $u \in G$ and $v \in G'$ that shall be stitched together to a new graph G''. The stitch function $stitch(G, G', P)$ produces G'' such that $G'' = (V \cup V', E \cup E' \cup P)$.*

For each target function **C::g** that could be successfully resolved, the algorithm stitches $\widetilde{cs_i}$ to $\pi_{\texttt{C::g}}$ (cf. Line 63): Actual pointer parameters are connected with the corresponding formal parameters of the callee function **C::g**. If **C::g** returns a pointer parameter, it is connected as well. All edges are annotated with the corresponding call site.

If this graph stitch affects a pointer that is listed in the dependency map, the algorithm recursively continues resolving the affected call sites. Otherwise, the algorithm recursively continues resolving call sites in the resolved target functions. The algorithms for the interwoven points-to, callgraph computation are shown in Algorithm 1, Algorithm 2, and Algorithm 3. We use the symbol cs in a call to the function $stitch(G, G', cs)$ as shorthand for $\{(a_i, f_i)\}$, the set of pairs of left-hand-site pointer variable/actual pointer parameters and pointer return value/formal pointer parameters of the callee at cs that are stitched together.

```
38 directed graph: CG_C = ∅, T = computeTypeHierarchy(); undirected graph:
     Π_C = ∅; bidirectional map: D = ∅; set: V = ∅;
39 Function constructionWalk(f):
40     if f ∈ V || isDeclaration(f) then
41         return;
42     V∪ = f;
43     foreach callsite cs ∈ f do
44         if cs is static then
45             CG_C∪ =< cs,getCallee(cs)>;
46             updatePointerInfo(f, getCallee(cs));
47         else
48             callees = resolveIndirectCallSite(cs);
49             foreach callee ∈ callees do
50                 CG_C∪ =< cs, callee >;
51                 updatePointerInfo(f, callee);
52     return;
53 Function constructCallGraph():
54     foreach f ∈ C do
55         if !isDeclaration(f) then
56             Π_C∪ = computePointsToGraph(f);
57     foreach f ∈ C \ {internalfunctions} do
58         if f ∉ V∧ !isDeclaration(f) then
59             CG_C∪ = f;
60             constructionWalk (f);
61     return;
```

Algorithm 1: Callgraph construction algorithm

```
62 Function updatePointerInfo(f, callee):
63     Π_C = stitch(Π_C[f], Π_C[callee], cs);
64     modptrs =
        getVerticesInvolvedInGraphOp(stitch, Π_C[f], Π_C[callee], cs);
65     foreach ptr ∈ modptrs do
66         if ptr ∈ D then
67             fmod =getFunctionContaining(D[ptr]);
68             V = V \ fmod;
69             constructionWalk(fmod);
70     constructionWalk(callee);
71     return;
```

Algorithm 2: Procedure for updating the pointer information

```
72 Function resolveIndirectCallSite(cs):
73     callees = ∅;
74     if isFunctionPtrCall(cs) then
75         fptr = getCalledPtr(cs);
76         rfptrs = getReachablePtrs(fptr);
77         foreach fptr′ ∈ rfptrs do
78             if isBoundaryPtr(fptr′) then
79                 D[cs] ∪= fptr′;
80         callees ∪= getReachableFunctions(fptr);
81     else
82         aptr = getAllocationPtr(cs);
83         raptrs = getReachablePtrs(aptrs);
84         foreach aptr′ ∈ raptrs do
85             if isBoundary(aptr′) then
86                 D[cs] ∪= aptr′;
87         allocs = getReachableAllocSites(aptr);
88         foreach alloc ∈ allocs do
89             τ = getAllocatedType(alloc);
90             v_τ = getVTable(T, τ);
91             if ! v_τ then
92                 D[τ] ∪= cs;
93             else
94                 i = getVCallIndex(cs);
95                 callee = getVTableEntry(v_τ, i);
96                 callees ∪= callee;
97     return callees;
```

Algorithm 3: Procedure for resolving dynamic call sites

Example 5.3.3. *The callgraph algorithm starts analyzing the function* ***Sanitizer :: sanitize*** *. At the call site* $\overline{cs}_4$*, the actual parameter is stitched to the formal parameter of* ***Sanitizer :: isMalicious*** *and the algorithm proceeds in* ***Sanitizer :: isMalicious*** *. Since* ***Sanitizer :: isMalicious*** *has now already been visited, the next function to be analyzed is* ***applySanitizer*** *.*

applySanitizer *contains two interesting call sites.* $\overline{cs}_{10}$ *is a static call to* ***getGlobalSan*** *. However, its definition is currently not available and thus, a callgraph node which is marked as a declaration is added to the callgraph. Note that the function causes incomplete points-to information as it returns a pointer value that is stored in variable* s *(cf. Figure 5.4d).*

Furthermore, a virtual function member is called at $\widetilde{cs}_{11}$ *on the receiver pointer variable* s *of type* ***Sanitizer*** $*$*. Due to dynamic dispatch we have incomplete information on the possibly called functions and are not able to resolve this call, because we cannot yet determine the allocation sites that are reachable through* s*. The algorithm marks this call site as incomplete and keeps track of the dependent pointer variable* s*. The call site has to be updated as further information might be discovered later on. For instance, if the definition of* ***getGlobalSan*** *becomes available that provides the required additional points-to*

information. The partial callgraph that can be computed individually on the `Sanitizer` *module is shown in Figure 5.5.*

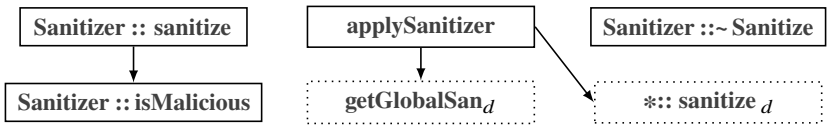


Figure 5.5: Callgraph for `Sanitizer`: $CG_{\text{Sanitizer}}$. f_d denotes the declaration of a function f.

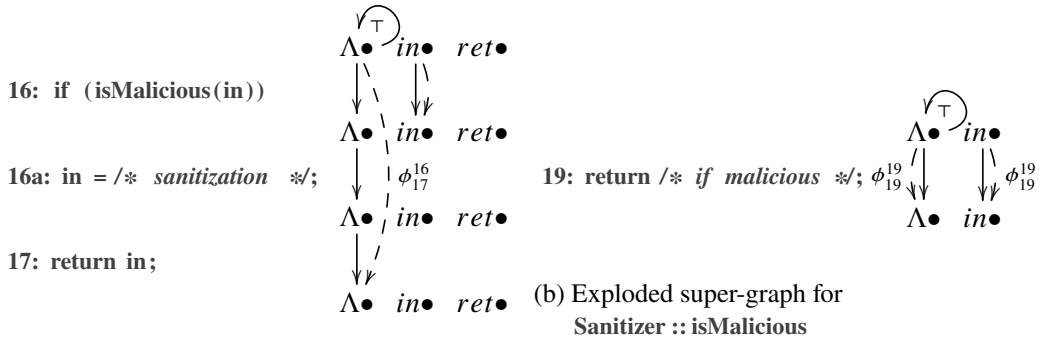
Data-Flow Information

In the next step, the analysis computes the possibly partial data-flow information using IFDS/IDE [RHS95, SRH96, NLR10] according to the description of the data-flow problem to be solved for the available function definitions. Importantly, because the flow functions are assumed to distribute over the merge operator, this summarization is known to be *lossless* [RHS95]. IFDS/IDE problems can thus be solved with full precision, without the need for approximation. In contrast to the information computed before, the data-flow information depends on the configuration of the client analysis because data-flow information is never general and always depending on a specific definition.

We use the flow and edge functions of the client analysis to construct the partial exploded super-graph of the library to be summarized. The partial callgraph is traversed in a depth-first bottom-up manner to maximize the number of functions that can be summarized completely. For a library function f that does not make any calls, the summary information is computed by applying the client's flow and edge functions to each node n of the control-flow graph. The resulting exploded super-graph edges are then combined using composition and meet to construct the *jump functions* $\psi(f)$ that summarize the complete function. For each incoming data-flow fact d_i, its respective jump function $\phi_i(f)$ describes the effect of the analyzed function on d_i.

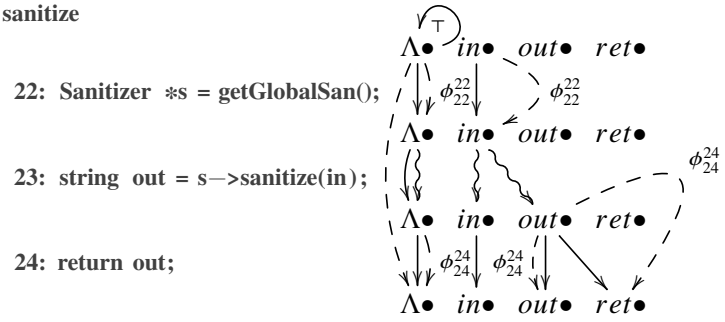
In case a function f contains call sites cs_i, the IFDS/IDE algorithm computes a partial data-flow summary from f's entry node to the first call-site node cs_1, $\psi^{entry}_{cs_1}(f)$. It then computes the summary of the called function f', $\psi(f')$, (if not already computed) and composes it with the partial summary $\psi^{entry}_{cs_1}(f)$ to obtain $\psi^{entry}_{rs_1}(f)$. The algorithm proceeds successively until the complete summary $\psi(f) = \psi^{entry}_{exit}(f)$ has been constructed.

However, in case a library function f contains call sites that are depending on user code, for instance, because of callbacks or incomplete points-to information, a complete summary $\psi(f)$ cannot be computed. In this case, ModAlyzer computes a set of partial summary functions ψ^n_m, where n is a function's entry point or some return site (rs) and m is a function's exit statement or some call site (cs) whose call targets are not or only partially known. This results in *gaps* in the exploded super-graph that represent the unresolved effects of the missing call targets.



(a) Exploded super-graph for **Sanitizer :: sanitize**

(b) Exploded super-graph for **Sanitizer :: isMalicious**



(c) Exploded super-graph for **applySanitizer**

Figure 5.6: Exploded super-graphs for the `Sanitizer` module.

Example 5.3.4. *The data-flow information computed for the* `Sanitizer` *module is shown in Figure 5.6. Individual flow/edge functions are denoted by solid (→) and jump functions by dashed (⇢) arrows. Analyzing* ***applySanitizer*** *leads to an incomplete ESG, because the callgraph for the* `Sanitizer` *module is only partially complete. The definition of* ***getGlobalSan*** *is not yet available and the dynamic call site at Line 11 cannot be resolved with the information available within the* `Sanitizer` *module.*

The call to the unresolved function ***getGlobalSan*** *does not interact* directly *with the data-flow information as it receives no arguments, its return type differs from the type of the data-flow domain (strings), and the string which the variable* ***in*** *refers to is not global as no global declarations are present. Therefore it cannot be modified by the call and one can safely use the identity function here. We will further elaborate on that in Section 5.3.4.*

The call to ****:: sanitize*** *results in a* gap *in the ESG. In Figure 5.6c gaps in the ESG are indicated with squiggled arrows (⇝). We pass* in *and* out *as identity after the gap and also generate other variables, such as the implicit return variable, that depend on* ***out****. Later on, after the merging process, the missing targets of the call site at Line 11 will have been determined and their data-flow summaries can be inserted. Then, the analysis will check whether in, out, and* ret *are reachable from* Λ*, and determine if those variables are tainted.*

The ESGs for the ***Sanitizer :: sanitize*** *and* ***Sanitizer :: isMalicious*** *functions are shown in Figure 5.6a and Figure 5.6b, respectively. For our example analysis we assume that* ***Sanitizer :: isMalicious*** *checks whether the variable* ***in*** *contains malicious data and the function does not modify the data-flow facts.* ***Sanitizer :: sanitize*** *checks if the string referred to by variable* ***in*** *contains malicious data—is tainted—and, if so, replaces it with a sanitized version. Again, to keep our example analysis simple, we assume that the analysis is aware of the special semantics of* ***Sanitizer :: isMalicious*** *and thus, kills the variable* ***in*** *in both branches.*

After having computed the data-flow summaries for the `Sanitizer` module, we have determined any information we need on `Sanitizer` as an individual module. We denote the combination of the partial type-hierarchy graph (and call targets) in Figure 5.3, partial points-to in Figure 5.4 and callgraph in Figure 5.5 and the partial data-flow summaries for `Sanitizer` in Figure 5.6 as $\Xi_{\texttt{Sanitizer}}$.

5.3.3 Merging Analysis Summaries

To complete the picture, we next combine the information obtained by analyzing `Sanitizer` and `DbgSanitizer` with an analysis of the client application `Main`.

For this we need to define a new operation on graphs which we call *contraction*. We use the contraction operation when new information becomes available during a merge, to replace placeholder nodes (that indicate missing information) of a graph by their counterparts that represent the actual information. We apply this operation to combine partial type hierarchy- and callgraphs. For instance, we combine callgraphs by *contracting away* function declaration nodes with their respective definition counterpart nodes: the nodes representing function declarations are removed and all former incoming edges now lead to the corresponding definition nodes. We formally define the contraction operation as follows:

Definition 5. *Contraction: Let $G = (V, E)$ be a (directed) graph containing vertices $\{u, v\} \subseteq V$ with $u \neq v$. Let f be a function that maps every vertex in $V \setminus \{u, v\}$ to itself, and otherwise, maps it to a new vertex w. The contraction of u and v is a new graph $G' = (V', E')$, where $V' = (V \setminus \{u, v\}) \cup \{w\}, E' = E \setminus \{e = (u, v)\}$, and for every $x \in V$, the vertex $x' = f(x) \in V'$ is incident to an edge $e' \in E'$, iff the corresponding edge $e \in E$ is incident to x in G (reproduced from [Ray14]). For convenience, we additionally define the function $contract : G \times G' \times P \rightarrow G''$ that maps the (directed) graphs $G = (V, E)$ and $G' = (V', E')$, and P a set of pairs of vertices $u \in V$ and $v \in V'$ that shall be contracted to a new graph G''. The contraction function $contract(G, G', P)$ contracts the pairs of vertices u_i and v_i and produces a new (directed) graph $G'' = ((V \cup V') \setminus \{u_i\}, (((E \cup E') \setminus \{(t_j, u_i)\}) \setminus \{(u_i, v_i)\}) \cup \{(t_j, v_i)\})$, where all edges incident to u_i with their origin in some vertex t_j are replaced by edges from t_j to v_i contracting away u_i. We use f in $contract(G, G', f)$ as shorthand for $\{(f_{decl}, f)\}$, the set of function declaration/definition pairs and τ in $contract(G, G', \tau)$ as shorthand for $\{(\tau_{decl}, \tau)\}$, the set of type declaration/definition pairs.*

Our merge procedure for two module summaries Ξ_i and Ξ_j is shown in Algorithm 4. In the following, we present all involved steps for each piece of analysis information.

Type Hierarchies

The analysis first merges the type-hierarchy graphs using vertex contraction (cf. Line 102), to remove redundant definitions of the same type. The redundancy is caused by including a type's definition (which usually resides in a corresponding header file) in multiple modules that require a type's exact data layout (e.g. for allocation or subtyping).

Example 5.3.5. *While performing the contraction, the analysis finds that* `Sanitizer`*'s type* $\tau_{Sanitizer}$ *is sub-typed by* $\tau_{DbgSanitizer}$*. The contraction has no immediate effect on the callgraph analysis: As the callgraph uses points-to information to resolve indirect calls, no immediate update is required at this point, because the new type-hierarchy information is not used before new pointer information becomes available. The type hierarchy needs to be queried if a new allocation site has been found. For each newly discovered allocation site, the type hierarchy is used to retrieve the entry of the allocated type's virtual function table.*

Callgraphs and Points-To Information

The analysis merges the callgraphs by using the vertex contraction operation introduced before (Line 109). A contraction is used to remove function-declaration nodes and replace them with their corresponding definition nodes, now linking calls to callees. While performing the contraction on the callgraphs, the corresponding partial pointer-assignment graphs are not contracted but stitched together (cf. Definition 4); through the stitch (Line 113) no nodes of the pointer-assignment graph are replaced to keep information on the parameter mapping. Actual pointer parameters at a call site as well as pointer return values at the respective return site are connected with the corresponding formal parameters of the called function and the left-hand side variables, respectively. The information on the contracted callgraph nodes is used in the next step when *repropagating* data-flows.

Example 5.3.6. *The callgraph contraction of the modules* `Sanitizer` *and* `DbgSanitizer` *is indicated in Figure 5.7. The callgraph contraction triggers the corresponding stitching of PAGs. For instance, the points-to graphs* $\pi_{Sanitizer\,::\,applySanitizer}$ *and* $\pi_{getGlobalSan}$ *are stitched together at* $\overline{cs}_{10}$ *as indicated in Figure 5.8. Through the stitch, the analysis recognizes that the previously marked pointer variable* s *gets new inputs from the resolved callee function* $getGlobalSan$*. As* s *is now able to reach* $getGlobalSan$*'s variable* s *of allocated type* $\tau_{DbgSanitizer}$ *and the receiver object* s *in* $applySanitizer$ *has no other unresolved dependencies, the possible call targets are updated in the callgraph such that* $DbgSanitizer\,::\,sanitize$ *is now the only possible target for the dynamic call site at Line 11. The pointer-assignment graph of the newly discovered callee at Line 11 is stitched to the call site* $\widetilde{cs}_{11}$.

Fixed-Point Iteration for Callgraph and Points-To Graph

Note that there are cases in which the stitch (of two PAGs) of a resolved callee function changes the points-to information in such a way that previously partially resolved indirect call sites must be revised again (cf. Line 63 for summarization, and Line 113 for merges). In these cases, the analysis loops in updating callgraph and points-to information until the

```
98  Function merge(CG_C, T_C, Π_C, D_C, V_C, CG_C', T_C', Π_C', D_C', V_C',):
99      D_C ∪= D_C';
100     V_C ∪= V_C';
101     Π_C ∪= Π_C';
102     T_C = contract(T_C, T_C', τ);
103     modtypes = getVerticesInvolvedInGraphOp(contract, T_C, T_C', τ);
104     foreach τ ∈ modtypes do
105         if τ ∈ D then
106             f =getFunctionContaining(D[τ]);
107             V = V \ f;
108             constructionWalk(f);
109     CG_C = contract(CG_C, CG_C', f);
110     {⟨cs, f⟩} =
          getVertexPairsInvolvedInGraphOp(contract, CG_C, CG_C', f);
111     foreach ⟨cs, f⟩ do
112         f' =getFunctionContaining(cs);
113         Π_C = stitch(Π_C[f'], Π_C[f], cs);
114         modptrs =
            getVerticesInvolvedInGraphOp(stitch, Π_C[f'], Π_C[f], cs);
115         foreach ptr ∈ modptrs do
116             if ptr ∈ D then
117                 f =getFunctionContaining(D[ptr]);
118                 V = V \ f;
119                 constructionWalk(f);
```

Algorithm 4: Merge procedure for callgraphs

```
1  void (*f)();
2  void bar() {}
3  void foo() { f = &bar; }
4  void init(void (*f)()) { f = &foo; }
5  int main() {
6    init(f);
7    f(); // <— indirect call site
8    return 0;
9  }
```

Listing 5.4: Example in which the update of points-to- invalidates callgraph information.

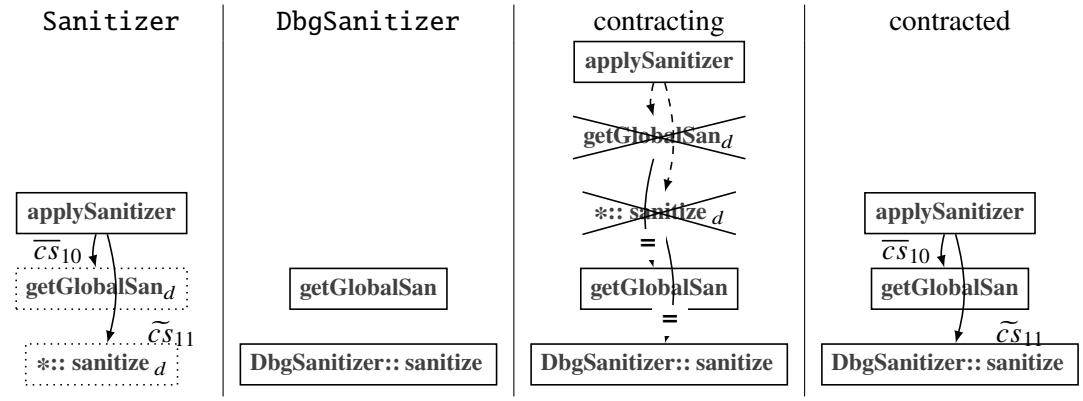


Figure 5.7: Excerpt of the vertex contraction for callgraphs of `Sanitizer` and `DbgSanitizer`. f_d denotes the declaration of a function f.

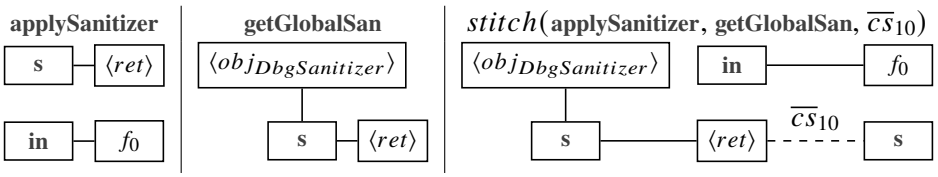


Figure 5.8: Excerpt of the vertex stitch of the PAG's for **applySanitizer** and **getGlobalSan**

callgraph and points-to information stabilize. A constructed yet expressive example of the aforementioned for function pointers is shown in Listing 5.4. When the callgraph algorithm resolves the indirect call to the function pointer **f** using points-to information, it determines **foo** as the call target. However, **foo** manipulates the points-to information such that **bar** becomes a feasible target as well. Thus, the indirect call site has to be revisited and **bar** has to be added as a possible target as well. When analyzing **bar** the callgraph and points-to information stabilize and the algorithm terminates.

Data-Flow Information

Once a callgraph has been updated by a merge, the data-flow information has to be repopulated in order to reflect the changes. Whenever two callgraphs are merged, new function definitions and their respective data-flow summaries become available which have been previously unknown to the other module's data-flow information. The merge procedure for the callgraphs shown in Algorithm 4 issues the contracted nodes (function declarations) and their respective call sites. This information and the newly available function definitions and accompanying data-flow summaries are used to close potential gaps in the ESG. The analysis visits all sub-graphs that have undergone the callgraph contraction procedure in a depth-first bottom-up manner, filling in the newly available data-flow summaries.

Suppose a function f contains a previously unresolved or only partially resolved call site cs and therefore, a pair of partial summaries $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$. If the callgraph

contraction reveals the call target f' and its respective data-flow summary, $\psi_{cs}^{entry}(f)$ and $\psi_{exit}^{rs}(f)$ are composed with $\psi(f')$ to produce a complete summary of f, $\psi_{exit}^{entry}(f) = \psi_{exit}^{rs}(f) \circ \psi(f') \circ \psi_{cs}^{entry}(f)$. The summary $\psi_{exit}^{entry}(f)$ may need to be merged with any existing jump functions that have been obtained along other paths, for instance, call-free-paths (cf. flows for Λ in Figure 5.6c) or paths for other call targets of cs that have been available for analysis already. The complete summary $\psi(f)$ is used to successively fill in potential other gaps in the ESG.

In case a target library to be summarized is depending on code of its user(s) because it uses features such as callbacks, for instance, the static analysis summaries Ξ even for the complete library code will contain gaps. Those gaps are eventually closed once the main application is available, analyzed and merged with the precomputed library summaries to produce the final analysis results.

Example 5.3.7. *As the function definition of* ***DbgSanitizer:: sanitize*** *becomes now accessible to* ***applySanitizer****, its respective data-flow summary can now be plugged into the current gap of* ***applySanitizer*** *to obtain a complete IFDS/IDE summary for it. The sub-graphs that undergo the contraction procedure are visited in a depth-first, bottom-up manner and the data-flow summary for* ***DbgSanitizer:: sanitize*** *is inserted into* ***applySanitizer****. The analysis therefore finds that the values passed as a reference parameter into* ***DbgSanitizer:: sanitize ()*** *and the value returned by it are indeed tainted. Therefore, the return value of* ***applySanitizer*** *is tainted as well. The pre-analysis of the library is now complete and the obtained results can be used by any potential client to the library.*

Analyzing the Main Application

When analyzing the application program `Main` the analysis first constructs `Main`'s type hierarchy, function-wise pointer-assignment and callgraph (cf. Algorithm 1). The type hierarchy-, call- and pointer-assignment graphs for `Main` are merged with the library's respective graphs (cf. Algorithm 4). The data-flow analysis can then start at the entry point **main**. As the data-flow analysis recognizes the call to **applySanitizer** it can directly use the (complete) pre-computed summary and thus keeps the return value as well as the actual reference parameter **input** marked as tainted. Finally, the client analysis is able to query the results and finds that the tainted variable **sanin** leaks at the call to **Statement::executeQuery**.

5.3.4 Removing Dependencies Ahead of Time

While computing the data-flow information for an individual module, information at dynamic call sites or static call sites, where the callee definitions are not available, will be incomplete. However, by using the following shortcuts, ModAlyzer is able to compute a complete and precise data-flow summary nonetheless. We already observed such a situation while computing the data-flow information for **applySanitizer** in the `Sanitizer` module. Because the call to **getGlobalSan** at Line 10 does not have a *direct* impact on the data-flow information (as described in Example 5.3.4), we can model it using the identity flow function. Note, however, that the call still has an *indirect* impact since the function is able to change what function is being called in the next line. When our analysis recognizes

a function f that misses information on potential callees, but where we can ensure that the missing information has no direct or indirect impact on the *data-flow* information, we can nevertheless compute a complete and precise summary for f using the *identity* shortcut denoted as $\overset{\text{id}}{\hookrightarrow}$ and thus fully remove any dependencies on the missing callees. To determine if $\overset{\text{id}}{\hookrightarrow}$ can be applied, different predicates may be applied, depending on the client analysis, e.g. *pass and return by value*. For instance, if a function receives its arguments *by value* they are copied into the callee. Thus, we can be sure that it cannot modify its arguments even if information on the callee's definition is missing.

```
1  std::string foo(bool p) {
2    return p ? sanitize(in) : in;
3  }
```

Listing 5.5: Code allowing the $\overset{\top}{\hookrightarrow}$ *shortcut*.

Another example of a situation in which a data-flow analysis can perform such an optimization is shown in Listing 5.5. Such a treatment for summarization of incomplete data-flow analysis has also been presented in [Klo08]. While analyzing **foo** we assume the information $\top$ for the variable **in**, i.e., **in** is tainted. **foo** sanitizes **in** only in one of the branches (depending on an unknown predicate). Hence, if we assume that we are conducting a may-taint analysis, then it holds that **in** *may* be tainted at the end of **foo** *no matter what* the call to **sanitize** does. It follows that $\top$ will always be associated with **in**. In this case, we can compute a complete summary even with incomplete information by using the $\top$ shortcut $\overset{\top}{\hookrightarrow}$. This is always true for *may*-analyses that use set union as the merge operator, which for instance in IFDS is always the case.

In the presence of global variables, ModAlyzer applies shortcuts *only* if they can be proven sound, which ModAlyzer manages easily if only module-internal global variables are involved. Global variables are often declared as static (in case of C) or within anonymous namespaces (in case of C++) making them internal to the module that declares them. ModAlyzer's shortcuts are not applied if externally visible global variables are involved in the situation, i.e., variables that are used across multiple modules.

Due to C and C++'s modular compilation model, an analysis frequently encounters situations as presented above, in which it can use these shortcuts to compute data-flow information. Functions where these shortcut summaries are used do not need to be revisited, thus, the analysis is able to work more efficiently. Therefore, when summarizing a module, it is desirable to remove as many data-flow dependencies as possible using the $\overset{\text{id}}{\hookrightarrow}$ and $\overset{\top}{\hookrightarrow}$ shortcuts.

5.4 Implementation

We have implemented the strategy described in Section 5.3 in a tool called ModAlyzer, as an extension to PhASAR [SHB19], a static-analysis framework that has been implemented on top of LLVM [LA04]. PhASAR allows to solve arbitrary monotone data-flow problems on the LLVM intermediate representation (LLVM IR) and also provides IFDS/IDE solver implementations.

We extended the existing IDE solver as well as the other infrastructure for type hierarchy, points-to, and callgraph computation and added the necessary summarize, merge, and update functionalities respectively.

ModAlyzer persists the summary results by using a document-oriented store in which it saves the graphs along with the code the analysis is conducted on with help of LLVM's metadata capabilities. LLVM allows for a key-based introduction of custom metadata. Each function that is defined in a module is annotated with its function-wise summaries for the different pieces of static analysis information, i.e., its points-to and exploded super-graph. A module carries the module-wise information that is obtained by merging all information of its enclosed functions as well as type hierarchy and callgraph information. Those module-wise summaries are referred to using the module flags section of the LLVM IR.

For the persistence, we created a bidirectional mapping from LLVM's in-memory representation to a textual representation allowing us to store the graphs comprising pointer values to LLVM IR records as graphs that use the text-encoded version. Additionally, we implemented *import* and *export* functionalities for each graph type that enable us to manage loads and stores of encoded graphs along with the LLVM IR.

An excerpt of an analysis summary created by ModAlyzer for a simplified taint analysis, that has been conducted on module `M` shown in Listing 5.6, is shown in Listing 5.7. The simplified taint analysis considers the integer literal **13** as source and therefore, taints variables to which the literal has been assigned to. Parts of the summary that are too large for presentation here have been abbreviated through **[...]** markers. Each instruction is annotated with a unique id as our textual representation uses an id-based encoding. For instance, the store instruction in Line 8 of Listing 5.7 carries a piece of metadata using the key **psr.id** which refers to the metadata entry **!3** in Line 32. That entry, in turn, stores the instruction's id **1**. Each function that is defined in `M` is annotated with its function-wise summaries for the different pieces of static analysis information. The **taint** function is analogously annotated with its points-to and exploded super-graph using the keys **psr.pt** and **psr.df** (cf. Line 6).[3] The module carries the module-wise information that is obtained by merging the information of its enclosed functions. Those module-wise summaries are referred to using the **llvm.module.flags** in Line 27.

```
1  int *taint(int *v) {
2   *v = 13;
3   return v;
4  }
5
6  int main() {
7   int x = 42;
8   int *y = taint(&x);
9   return 0;
10 }
```

Listing 5.6: Example module `M`

[3] C++ compilers mangle the names of user-defined functions using a scheme that reflects the types of its parameter list to allow for function overloading and thus, **taint** appears as **_Z5taintPi** in Listing 5.7.

```
; ModuleID = 'Module_M.cpp'
source_filename = "Module_M.cpp"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define i32* @_Z5taintPi(i32*) #0 !psr.pt !19 !psr.df !20 {
  %2 = alloca i32*, align 8, !psr.id !2
  store i32* %0, i32** %2, align 8, !psr.id !3
  %3 = load i32*, i32** %2, align 8, !psr.id !4
  store i32 13, i32* %3, align 4, !psr.id !5
  %4 = load i32*, i32** %2, align 8, !psr.id !6
  ret i32* %4, !psr.id !7
}
; Function Attrs: noinline norecurse nounwind optnone uwtable
define i32 @main() #1 !psr.pt !21 !psr.df !22 {
  %1 = alloca i32, align 4, !psr.id !8
  %2 = alloca i32, align 4, !psr.id !9
  %3 = alloca i32*, align 8, !psr.id !10
  store i32 0, i32* %1, align 4, !psr.id !11
  store i32 42, i32* %2, align 4, !psr.id !12
  %4 = call i32* @_Z5taintPi(i32* %2), !psr.id !13
  store i32* %4, i32** %3, align 8, !psr.id !14
  ret i32 0, !psr.id !15
}
attributes #0 = { noinline nounwind optnone uwtable [...] }
attributes #1 = { noinline norecurse nounwind optnone uwtable [...]
    }
!llvm.module.flags = !{!0, !16, !17, !18}
!llvm.ident = !{!1}
!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 5.0.1 (tags/RELEASE_501/final 341448)"}
!2 = !{!"0"}
!3 = !{!"1"}
[...]
!15 = !{!"13"}
!16 = !{i32 5, !"psr.th", !"graph G {}"}
!17 = !{i32 5, !"psr.pt", !"graph G {
            0[label="_Z5taintPi.0"];
            1[label="4"];
            2[label="6"];
            3[label="9"];
            4[label="13"];
            0--1 [label=""];
            0--2 [label=""];
            1--2 [label=""];
            3--4 [label="13"];
}"}
!18 = !{i32 5, !"psr.cg", !"graph G {
            0[label=main];
            1[label=_Z5taintPi];
            0->1 [label="13"];
}"}
!19 = !{!"graph G {
```

```
53              0[label="_Z5taintPi.0"];
54              1[label="4"];
55              2[label="6"];
56              0--1 [label=""];
57              0--2 [label=""];
58              1--2 [label=""];
59  }"}
60  !20 = !{!"{
61              <2 ; lambda ; 7 ; lambda ; TOP>
62  }"}
63  !21 = !{!"graph G {
64              0[label="9"];
65              1[label="13"];
66              0--1 [label=""];
67  }"}
68  !22 = !{!"{
69              <8 ; lambda ; 15 ; lambda ; TOP>
70  }"}
```

Listing 5.7: Simplified ModAlyzer summary of module `M` shown in Listing 5.6.

LLVM's metadata mechanism does not restrict the type of data for annotations. Thus, arbitrary data structures and encodings may be used to persist the analysis information. We use the capabilities of the Boost Graph Library (BGL) [SLL02] to manage type hierarchy, points-to, and callgraph information. The BGL offers of-the-shelf textual import and export functionalities and allows for implementing custom reader/writer concepts. We use the default Graphivz [Gra19] format to store the graphs in metadata records. As PhASAR's IFDS/IDE solver implementation works by incrementally constructing two tables to represent flow functions/jump functions of ever longer sequences of code (c.f. [SHB19,NLR10]), we use the following sets of quintuples for the data-flow summary representation of a function $\psi(f) := \{\langle n_i, d_x, n_j, d_y, l\rangle\}$, where a quintuple represents a jump function (or an edge in the ESG) from data-flow fact d_x to d_y with the corresponding edge function l that summarizes parts of the effects of the region of code that is enclosed by the statements n_i and n_j. The concrete (partial) data-flow summary for the **applySanitizer** function (cf. Figure 5.6c) looks as shown in Table 5.1.

Table 5.1: Data-flow summary for the **applySanitizer** function.

$\langle$	22	Λ	24	Λ	$\top$	$\rangle$
$\langle$	22	*in*	22	*in*	$\top$	$\rangle$
$\langle$	24	*in*	24	*in*	$\top$	$\rangle$
$\langle$	24	*out*	24	*out*	$\top$	$\rangle$
$\langle$	24	*out*	24	*ret*	$\top$	$\rangle$

Note that for IFDS we can use the simple encoding of the binary lattice and the edge functions. We handle the persistence of the difficult-to-handle, general IDE edge functions by creating a record to keep track which edge functions are composed and meet for each jump function while constructing them. We finally persist the record using the extensive

Boost Serialization library [Boo19]. On load, the record can be *replayed* to (re)construct the actual jump functions.

5.5 Experiments

Our empirical evaluation aims to answer the following research questions:

RQ_7	*Does the use of a module-wise static analysis incur a precision loss when compared to a whole program analysis? If so, what causes this loss in precision?*
RQ_8	*Compared to conducting a whole-program analysis, what speed-up can one achieve when applying MWA using pre-computed summaries for type-hierarchy, callgraph, points-to and data-flow information?*
RQ_9	*How frequently can the data-flow shortcuts* $\overset{\mathsf{id}}{\hookrightarrow}$ *and* $\overset{\top}{\hookrightarrow}$ *be applied in MWA?*

To address RQ_7, we compare the analysis results of a whole program analysis with the results obtained by a module-wise analysis. Ideally, the results of both analyses should be identical. To address RQ_8, we measure and compare the runtimes of a client analysis using pre-computed summaries and a version that computes everything on-the-fly. To address RQ_9, we extend PhASAR's IFDS/IDE solver implementation and measure how frequently it makes use of both shortcuts for different client analyses.

5.5.1 Experimental Setup

We have evaluated ModAlyzer using as benchmark subjects the C coreutils (version 8.28) [Cor18] and the PhASAR framework itself.

The GNU core utilities are a collection of C programs that share a common core, providing a library that consists of 251 files. Each coreutil program itself only consists of a small number of C source files that provides the program's entry point, manages the command-line, and makes suitable calls into the common core in order to achieve the desired task. For our evaluation we prepared and analyzed 97 of the coreutils and chose 10 of them at random which to present in this paper in more detail. (However, the figures for the remaining 87 coreutils can be found online [Art21].)

PhASAR is written in C++ and is similarly structured. To provide flexible, reusable software components, the main functionalities of the different components are implemented as libraries. The front-ends (or drivers) themselves represent only a relatively small amount of "glue code" and large amounts of their runtime is spent in library code. Using PhASAR we defined two benchmark subjects: First PhASAR's own command-line client and the PhASAR-based tool *MPT*, a exemplary client that uses PhASAR as a library, both of which can be found alongside PhASAR's examples [Pha18].

We chose those subjects because they have a relatively high amount of virtual calls. This stresses ModAlyzer's points-to based callgraph algorithm. We observed that C++

Table 5.2: Number of compilation units, library/application code ratio, number of statements, pointer variables and allocation sites of the analyzed (completely linked) programs.

Program	Compilation Units	$\frac{IR\ LOC\ lib}{IR\ LOC\ app}$	Statements	Pointers	Allocation Sites
wc	252	41.2	63,166	10,644	396
ls	253	5.9	71,712	13,200	438
cat	252	66.3	62,588	10,584	391
cp	256	10.5	67,097	11,722	443
whoami	252	335.7	61,860	10,433	389
dd	252	16.8	65,287	11,150	408
fold	252	105.8	62,201	10,509	390
join	252	24.9	64,196	11,042	402
kill	253	88.2	62,304	10,527	394
uniq	252	60.1	62,663	10,650	396
MPT	156	13.8	1,351,735	755,567	176,540
PhASAR (driver)	156	56.4	1,368,297	763,796	178,486

developers generally try to minimize the amount of indirect calls to avoid indirect jumps, which degrade performance, especially when implementing performance critical software systems [AH96]. The chosen subjects hence set a relatively high bar when it comes to evaluating analysis performance. The raw as well as the processed data produced in our evaluation is available online [Art21].

All programs and their characteristics are shown in Table 5.2. We prepared all programs presented for analysis with the PhASAR framework by compiling them into LLVM IR with production flags using the Clang compiler. The numbers in Table 5.2 are based on LLVM IR.

We used an uninitialized-variables analysis $\mathbb{U}$ and a taint analysis $\mathbb{T}$ as two concrete client analyses that both impose the information dependencies as shown in Figure 2.8. $\mathbb{U}$ and $\mathbb{T}$ are both implemented in IFDS within PhASAR.

Uninitialized-variables analysis $\mathbb{U}$: $\mathbb{U}$ is an analysis that finds potentially uninitialized variables and tracks them through the program. If the analysis finds an uninitialized variable to be read from, it reports an *illegal* use of that variable. Uninitialized variables propagate through computations and thus, the analysis tracks those as well. $\mathbb{U}$ also tracks the variables across function boundaries making it an inter-procedural analysis.

Taint analysis $\mathbb{T}$: $\mathbb{T}$ is a parameterizable taint analysis that tracks tainted values through the program and reports potential leaks whenever it finds a tainted value that may flows into a *sink* function (or operation). *Sources* and *sinks* are parameterizable. We used PhASAR's default parametrization that treats the command-line arguments passed into **main** as tainted. All standard *input* functions (e.g., **fread**, **fgets**) are treated as *sources* as well. All *output* functions (e.g., **fwrite**, **printf**) are treated as *sinks*.

For each target program shown in Table 5.2 we computed the library and application code ratio based on lines of LLVM IR code. If a module is used by more than one application, we consider it to be part of the library, whereas modules that are only used by one application are considered as application code. We also measured runtimes and

number of leaks/uninitialized variables that each of the analyses reported in a WPA setup as well as an MWA setup. The measurements for MWA are split into a summarization and an actual analysis step. The PhASAR framework implements a reporting system which we use to compare the actual reports to make sure that the findings are identical. We also recorded the number of callgraph updates $\#CG \circlearrowleft$ that had to be performed in the MWA setup, i.e., we counted the number of callgraph edges that have been introduced during the merge process. This is a good indicator of the expense of a merge, as the introduction of a new callgraph edge causes the points-to and data-flow information to be updated as well. In addition, we measured the number of shortcuts that a data-flow analysis was able to use. We measured the runtimes by performing 5 runs for each analysis in each setup on a virtual machine running on an Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz machine with 128GB memory. We removed the minimum and maximum values and computed the average of the remaining 3 values. Table 5.3 shows the results. The first column comprises the programs under analysis, the second column contains the WPA runtimes, column three contains the required runtime for summarization, column four the actual analysis time of MWA. The differences of the runtimes and reports of WPA and MWA are shown in column five. Column six, seven, and eight contain the respective number of callgraph updates, identity shortcuts, and $\top$ shortcuts, respectively. The number of callgraph updates are equal for both analysis as the callgraph information is not affected by the concrete client analyses.

5.5.2 RQ_7: Precision

As the points-to and therefore, call- and control-flow graphs guide an analysis through a program, they may heavily influence the reported results. Therefore, we compared the callgraph obtained in an MWA setting with the one obtained in a WPA setting. We found that the callgraphs only differ at call-sites at which a static function pointer is called. In those cases, our MWA callgraph implementation turns out to be *more* precise as it does not consider every function of the complete program that matches the pointer's signature as a possible target, but only the ones reachable within the module whose address can actually be taken.[4] This reduces the number of infeasible call targets while retaining soundness.

We compared the client analyses precision and recall of WPA and MWA using PhASAR's reporting capabilities. Column Δ in Table 5.3 shows how many result entries differ from a WPA to an MWA setup for each client analysis. We only observed a difference in the reports for the "dd" program while performing the taint analysis. In this case, the analysis in WPA mode reports three leaks in a library function f_L, whereas the analysis in MWA reports none. We investigated the cause of this difference and found that this is actually a false positive in the WPA. The leaking function f_L is not called within the "dd" program. However, "dd" defines a static global function pointer p in the application code and the WPA analysis safely assumes that f_L, which matches the function pointers signature, might be called. When the application code that defines the static function pointer is analyzed in MWA mode, the analysis does not find a declaration of f_L within the application code

[4] Reducing the set of feasible function pointer targets in WPA mode can be easily implemented.

Table 5.3: Runtimes and findings WPA vs. MWA for the taint analysis $\mathbb{T}$ (first half) and uninitialized variables $\mathbb{U}$ (second half).

$\mathbb{T}$: Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\circlearrowleft^{CG}$	$\#\hookrightarrow^{id}$	$\#\hookrightarrow^{\top}$
wc	2.3	5.7	0.5	-1.8 / (0)	47	8,052	78
ls	4.8	5.7	1.3	-3.5 / (0)	166	13,470	11
cat	1.9	5.7	0.2	-1.7 / (0)	21	2,117	269
cp	4.4	5.7	1.8	-2.6 / (0)	197	19,712	1077
whoami	2.0	5.7	0.4	-1.6 / (0)	4	6,065	11
dd	8.1	5.7	5.5	-2.6 / (-3)	58	48,747	90
fold	2.1	5.8	0.4	-1.7 / (0)	12	6,695	11
join	2.4	5.7	0.6	-1.8 / (0)	58	8,979	11
kill	1.9	5.7	0.2	-1.7 / (0)	14	2,079	11
uniq	2.2	5.7	0.4	-1.8 / (0)	29	7,281	11
MPT	2,306	42,847	1,516	-809 / (0)	41	29,061	0
PhASAR	7,176	42,876	598	-6578 / (0)	3	47,736	0
$\mathbb{U}$: Program	WPA [s]	$\Sigma_{m \in lib}$ [s]	MWA [s]	Δ runtimes / (Δ reports)	$\#\circlearrowleft^{CG}$	$\#\hookrightarrow^{id}$	$\#\hookrightarrow^{\top}$
wc	2.6	5.9	0.6	-2.0 / (0)	47	2,413	162
ls	8.4	6.0	3.3	-5.1 / (0)	166	7,173	184
cat	2.0	6.0	0.3	-1.7 / (0)	21	845	12
cp	5.2	5.9	2.2	-3.0 / (0)	197	6,684	1122
whoami	2.0	5.9	0.3	-1.7 / (0)	4	535	0
dd	3.1	5.9	0.9	-2.2 / (0)	58	2,522	16
fold	2.1	6.0	0.4	-1.7 / (0)	12	895	0
join	2.8	6.0	0.5	-2.3 / (0)	58	2,582	171
kill	2.2	6.0	0.4	-1.8 / (0)	14	793	12
uniq	2.5	5.9	0.5	-2.0 / (0)	29	1,433	17
MPT	3,811	53,703	2,958	-826 / (0)	41	137,722	8,136
PhASAR	10,160	53,348	968	-9,192 / (0)	3	210,032	24,446

and therefore, its address cannot possibly be taken, preventing it to be a call target of p. While one could adapt the WPA to be equally precise, the MWA obtains this precision automatically.

Since ModAlyzer does not need to overapproximate information it does indeed also preserve recall. The ModAlyzer approach has been designed to obtain this property by construction. Besides the differing result entries that are caused by the differences in the callgraph, both the results of ModAlyzer and WPA coincide.

The module-wise analysis generally yields the same precision as the whole-program analysis, in some cases even exceeds it.

5.5.3 RQ_8: Performance

Table 5.2 shows that the library/application ratio ranges from 5.9 to 5675.6 and therefore, that the actual application code only comprises a small fraction of the complete program. One expects the MWA runtime to pay off better with increasing code ratios, since more pre-computed summaries can be (re)used for a program's library parts. The runtimes of

both analyses measured in the WPA and MWA setup live up to that expectation. Looking at the programs with an especially advantageous library/application ratio such as whoami, fold, kill, cat, PhASAR, the use of pre-computed summaries saves between 81% and 91% of the analysis time. On average, MWA saves 72% of analysis time compared to WPA while MWA's initial one-time summarization step is, on average, 3.67 times as expensive than the corresponding run in a WPA setup. Thus, computing the initial summarization of the library (or infrequently changing) parts of a program is more expensive that performing a whole program analysis. Computing summaries will always be more expensive compared to computing plain WPA due to the additional overhead required for organizing and maintaining the summaries. In addition, many of PhASAR's critical analysis parts have undergone tremendous amounts of manual optimization while ModAlyzer's implementation for summary generation has not yet been optimized manually. As a concrete example, analyzing PhASAR in an MWA setup outperforms WPA with the seventh run using the taint analysis and after the sixth run for the uninitialized variables analysis—assuming an initial summary must be computed and no changes in PhASAR's library occur after summarization. For the MPT program, that has a larger number of callgraph updates to be performed, MWA pays off with the 54th run for the taint analysis and 64th run for the uninitialized variables analysis, respectively.

In case of PhASAR, runtime savings of 92% can be achieved as the application merely consists of few calls into the library code. This is underlined by the three callgraph updates that are necessary. We manually inspected the program and confirmed that, although the amount of front-end code is certainly large, it performs only very few calls into the corresponding library. A controller class, which is part of the library, is used to dispatch the different tasks to solve into calls to the adequate library functionalities. This shifts large parts of the computation to the offline MWA summarization phase.

The size of the persisted summaries that are stored along with the library code increase a library, on average, by a factor of five in size. The code and summaries for PhASAR require approx. 2.8 GB of memory for persistence and 30 MB for the core utils.

Summaries for static callgraph, points-to and data-flow analysis can be used to capture the analysis effects of libraries. After a one-time pre-computation effort, this allows a runtime reduction of 72%, on average, compared to the runtimes in whole-program mode.

5.5.4 RQ_9: Shortcuts

The number of $\overset{\mathsf{id}}{\hookrightarrow}$ shortcuts taken by an analysis is parameterized by a predicate as described in Section 5.3.4. For the analyses $\mathbb{U}$ and $\mathbb{T}$ we used the predicate *return type is void and uses pass-by-value*. However, different predicates might be useful for other analyses, depending on the specific assumptions that can be made on an analysis's domain. The results in Table 5.3 show that both shortcuts can be frequently applied during analysis. The $\overset{\mathsf{id}}{\hookrightarrow}$ shortcut can be applied between 535 and 210,032 times depending on the client data-flow analysis that is performed. The $\overset{\top}{\hookrightarrow}$ shortcut can be applied between 0 and 24,446 times. We are confident that the number of $\overset{\top}{\hookrightarrow}$ shortcuts could be further increased, if one adjusts

PhASAR's data-flow solvers to favour analyzing branches first that contains fewer (or no) function calls.

Shortcuts can be frequently applied. Hence, to decrease the number of data-flow dependencies and to increase the amount of complete summaries that can be pre-computed offline, it is advisable to make use of shortcuts whenever possible.

5.6 Limitations

In this section, we briefly discuss the limitations of ModAlyzer. ModAlyzer needs to summarize the different pieces of information presented in Figure 2.8 to be able to construct effective module-wise summaries for a given concrete client analysis. Hence, ModAlyzer requires analysis algorithms that produce summarizable results such as IFDS [RHS95], IDE [SRH96] or *Weighted Pushdown Systems* (WPDS) [RSJ03].

For problems that are distributive, hence fit into these frameworks, the summarization is lossless. It is generally also possible to use ModAlyzer to solve non-distributive client analysis problems. As mentioned in Section 5.1, one cannot generally compute summaries for non-distributive data-flow problems. In that case, the approach can only make use of the summaries for type-hierarchy, points-to, and callgraph information, which may still lead to large performance increases as we present in Section 5.5.

We use never-invalidating points-to information computed using an *Andersen* [And94] or *Steensgaard*-style [Ste96] algorithm to be able to produce effective summaries. Again, computing more precise inter-procedural, context-, and flow-sensitive points-to information is a non-distributive problem for which no effective summaries can be computed. However, Späth et al. showed how flow- and context-sensitive pointer analysis can be decomposed into multiple analysis problems each of which, in turn, can be expressed within a distributive framework [SNAB16]—making the overall problem distributive. ModAlyzer's current points-to algorithm could therefore also be replaced by an adjusted version the distributive Bommerang approach proposed by Späth et al. The Boomerang approach—as is—operates in an on-demand manner and does not compute reusable summaries nor does it persist results. It is interesting to see the performance of ModAlyzer with an improved Boomerang-style points-to algorithm, that reuses summaries, presented in [SNAB16], but we consider it as future work.

As described in Section 5.5, ModAlyzer's overall effectiveness degrades with the number of updates that must be performed while merging summaries with the application code. Therefore, ModAlyzer's performance increase may not apply to programs that make excessive use of callbacks.

5.7 Related Work

Several previous approaches address, in part, the difficult problem of compositional static program analysis [OPS92, CDG93, RR01, Dwy97, HR96, RRL99, GRS00, RMR04, XHN05, TWX+17, CCP17]. However, existing techniques for compositional static analysis typically

focus on data-flow or points-to analysis only. As advocated in this paper, a concrete compositional data-flow analysis client requires at the very least a combination of compositional callgraph, points-to and data-flow analysis.

Compositional data-flow techniques rely on the functional approach [SP78] allowing to solve distributive data-flow problems by using summary-based, inherently compositional frameworks such as IFDS [RHS95], IDE [SRH96], or WPDS [RSJ03]. Rountev et al. used IDE data-flow summaries to summarize large object-oriented libraries [RSX08] and showed that a significant amount of time can be saved when using pre-computed summaries. The approach presented by Rountev et al., however, omits to tackle the challenging task of persisting general IDE summaries but rather discards the summaries at analysis shutdown. StubDroid [AB16] is a fully automated approach to generate precise library models for taint-analysis problems for the Android Framework, effectively preventing the re-analysis of the Android Framework for the analysis of different Android apps. Both Rountev's approach and StubDroid assume the existence of whole-program points-to and callgraph information.

Several works use partial points-to information in from of function-local summaries computed using context-free language (CFL-)reachability [WR99, LSXX13, SXX12]. The summaries can be used in various scenarios allowing, among others, for on-demand points-to analysis, pre-analysis, and pointer analysis of partial programs using different sensitivities. These works present individual solutions to individual problems, while this paper presents the first integrated approach and shows its effectiveness on real-world C and C++ applications.

The IDEal [SAB17] approach developed by Späth et al. is an alias-aware extension to the IDE [SRH96] framework. IDEal embeds the alias analysis Boomerang [SNAB16] into the IDE solver implementation HEROS [Bod12] to automatically resolve alias queries on-demand at analysis time while solving a given distributive data-flow analysis problem. However, it does not compute (persisted,) reusable summaries but rather computes analysis queries on-demand and still requires external callgraph graph information.

AVERROES [AL13] uses the *separate compilation assumption* and Java's constant pool [Jav18] to generate sound and precise callgraphs without actually analyzing library code in order to generate a placeholder library. Existing whole-program callgraph construction algorithms can use the replacement to obtain a sound and precise application callgraph. AVERROES supports callgraph construction only. Its summaries cannot be used for precise pointer analysis, nor for precise data-flow analysis.

Other techniques try to improve the scalability of inter-procedural static analysis by using sparse propagation of data-flow facts along def-use chains [SX16b] or demand-driven analysis that only analyze parts of a program that a user is currently interested in [SX16a, SNAB16]. Sparseness is a concept orthogonal to the ones proposed here. Both could be used in combination.

Some tools, including clang-tidy [Cla18c] and CppCheck [Cpp18], trade off scalability for reduced complexity. Thus, they only apply syntactic analysis to retrieve information on the property of interest. Precise, fully-fledged static analysis is replaced by much simpler checks that are capable of analyzing even million lines of code in minutes. However, these checks are often too imprecise to check for interesting properties.

Klohs et al. described the situation for *may*-analysis in which $\top$, representing all information, is obtained along one path in the control-flow graph, and thus, the other path does not have to be analyzed. This allows to remove data-flow dependencies ahead of time [Klo08]. The approach presented here adopts this insight.

ModAlyzer computes the module-level summaries in a completely unrestricted way and does not make any assumptions about missing code. Yet, it may be advisable to compute summaries based on various sensible assumptions in scenarios where the summarization step can be performed ahead of time, e.g. for library pre-analysis. Tree-adjoining languages [TWZ+15] and Dyck context-free language reachability [TWX+17, CCP17] can be used to increase the effective library summarization by computing reasonable conditional summaries that enable greater summary reuse under certain premises checked at analysis time of the application code. Such a strategy allows for more computations to be performed on a module-level. During the merge, the analysis can check whether an assumption that has been made holds and, if so, directly use the corresponding summary that may be much more expressive than one that has been computed without any assumptions about missing code, effectively reducing the amount of work that needs to be done while merging summaries with the application code. ModAlyzer currently does not use such a conditional summarization, however, it provides all required infrastructure to easily integrate the approach. Unfortunately, one cannot rely on programmers specifying pointer or reference parameters as constants using the **const** keyword because C and C++'s type system provides several mechanisms to circumvent constant declarations (e.g. **const_cast** and **mutable** in case of C++). Although writes through **const** are possible, they are used sparingly in real-world software as shown by Eyolfson and Lam [EL16]. Therefore, one reasonable assumption may be *const means const*. Especially **const**-qualified pointer parameters then represent hard inter-procedural boundaries and a data-flow analysis is not concerned with those parameters.

Early versions of Facebook's Infer [CD11] used separation logic to allow for the compositional analysis of heap-based programs. The approach computed bottom-up summaries using bi-abductive inference [CDOY09, BGS18], which could then be used in different calling contexts. Using Infer, one could thus formulate compositional static analyses that are evaluated using abstract interpretation. These analyses, however, were largely restricted to finding cases of memory corruption. Since about 2019—reportedly due to a lack of general applicability and extensibility—Infer thus does not use abductive inference for most of its analyses any longer, and now instead bases its implementation on data-flow analysis using abstract interpretation. This analysis is no longer compositional.

5.8 Conclusions

In this chapter, we presented ModAlyzer, a compositional approach to speeding up static analysis using persisted summaries for callgraph, points-to and data-flow information. We have presented an integrated strategy based on the dependencies as shown in Figure 2.8 that manages all those information and their dependencies, which many useful, concrete client analyses impose to provide precise results. ModAlyzer allows one to compute

static analysis summaries on individual parts of a program without the need to make any assumptions on the missing code. These pre-computed summaries can then be (re)used later on, effectively shifting large parts of the computational effort to an offline phase. In particular, the pre-computed summaries can be persisted using a document-oriented store that then be checked into the software repository of the target project. This has the great benefit of allowing developers to share analysis information that would otherwise require expensive recomputation. In such a scenario, any developer can check out a specific version of the target project along with the already summarized static analysis information of the project's library parts, which can then be directly integrated into an analysis of the actual application code.

Our experiments confirm the finding by previous works that actual application code often only constitutes only a small fraction of the complete program. Thus, ModAlyzer outperforms traditional whole program analysis in both runtime and flexibility.

Until now, we described how to improve static analysis' scalability by exploiting the fact that large portions of a project typically do not change frequently. We will next present how to further adjust static analysis to cope with parts of the code that do undergo frequent changes in a continuous integration/continuous deployment setup, for instance. This will enable us to reduce the computations that must be conducted to produce up-to-date analysis information from one analysis run to another to a minimum.

6 Incrementality

In this chapter, we detail on how to efficiently analyze software projects that are maintained with help of version control systems and developed in a continuous integration/continuous deployment (CI/CD) manner. While a software project's library components can be efficiently handled using ModAlyzer, its non-library components are known to change frequently and hence require special treatment. The changes of individual commits in CI/CD workflows, however, are relatively small and local as this chapter will reveal—a key fact that we are going to exploit.

Many modern software projects are developed incrementally using continuous integration and continuous deployment. Traditional static whole program analysis does not match this development workflow as it has to be rerun for every change made to the code. Hence, various incremental analysis approaches have been developed that reuse previously computed analysis results for code that did not change since the last revision. These approaches allow for more scalable analysis by avoiding unnecessary (re)computations. Yet, existing incremental static analysis approaches do not take advantage of information provided by version control systems that are used in continuous integration. In this work, we present IncAlyzer, a novel program analysis approach that integrates version control system information and deep incremental static analysis. IncAlyzer not only allows for efficiently reusing previously computed analysis results, but also makes static analysis fit for CI/CD workflows. This has great potential to allow developers to check-in persisted analysis results alongside the respective code revision the analysis has been conducted on, making up-to-date analysis results available to, and allowing their reuse by anyone who checks-out the project. We implemented IncAlyzer as an extension to LLVM, VaRA, PhASAR, and ModAlyzer. We applied it to five real-world C projects hosted on Github and analyzed the past 50 commits of each project. Our experiments show that IncAlyzer outperforms traditional whole program analysis, on average, by a factor of 6.99 in runtime without any loss of precision or soundness and that its results are from-scratch consistent.

This chapter presents the design and implementation of IncAlyzer, a novel integrated static analysis approach that computes analysis information incrementally and applies to code that undergoes frequent changes.

6.1 Introduction

Static analysis becomes increasingly important in modern software development. Since many years, static analysis is not only used to aid compiler optimization [Onl18, ICC18], but also to detect bugs [CS18, Cod18, son23b, Son23a] and security breaches [KNR+17, LL05] in software.

However, as we pointed out in the previous chapters, many inter-procedural static analyses, i.e., whole-program analyses that reason about semantic properties of a program, require detailed abstractions of the target program and additional helper analyses to serve a concrete client analysis [Bod18, SLHB21]. Those detailed abstractions allow for very precise analyses, but, in turn, impede scalability [BBC+10, TG17]. Thus, traditional precise whole program analysis becomes prohibitively expensive for frequently changing code.

Modern application development, however, often uses continuous integration and version control systems (VCS) such as Git [Git19], Mercurial [Hg19], or SVN [SVN19] to organize and maintain code bases. CI/CD-style development is notorious for causing frequently changing code. The LLVM project, for instance, usually receives more than 500 commits per week [LLV19]. Extrapolating the number of per-week commits for the LLVM project results in 72 commits per day, on average. Thus, a program analysis, on average, may not exceed a timeframe of 20 minutes if all commits of a day shall undergo an analysis. Traditionally, for every change made to the code, a whole-program analysis would be triggered that ignores all previously computed results. Depending on the size of the program under analysis, it is hard to impossible to run a precise, heavyweight static analysis within a few minutes. Whole program analyses are oftentimes postponed to the end of the day and are run in batch style during the nightly build. In such a scenario they may not exceed the timeframe of eight hours [BBC+10]. Facebook reported that their code basis changes so frequently that they have trouble keeping their analyses up with the code changes such that the results are still relevant when being reported [HO18].

Summarization techniques as presented by [SHB21, AB14, RSX08] are able to summarize libraries and parts of the code that do not change frequently. Those pre-computed summaries can then be used while analyzing the actual application code and may decrease the analysis time by a large factor. Tree-adjoining languages [TWZ+15] and Dyck context-free language reachability [TWX+17, CCP17] are able to increase the amount of useful summaries by computing reasonable conditional summaries for libraries from which the appropriate one is chosen while analyzing the actual application code that uses those libraries. Summarization itself is very expensive, however, which is why it is most effectively applied to parts of a program that do *not* change frequently [SHB21].

Making static analysis incremental can help to improve its scalability for frequently changing code as presented by a large number of previous works [MR14, SVE17, AB14, Ryd83, CNDE05, LHR19]. The idea of incremental analysis is that changes made to a program are usually small [Swa76, HL08, BCSD14] and thus, should only cause invalidation of a small amount of the analysis results, allowing one to reuse large parts of previously computed analysis results. Incrementality has been proven to speed up computations by magnitudes as shown by [BWR+11, SAIM08].

Yet, existing incremental static analysis techniques ignore the information provided by VCSs and are only concerned with data-flow information [AB14, CD11, SEV16, SVE17]. The Reviser [AB14] approach, for instance, assesses the parts of the data-flow results that can be reused based on the inter-procedural control-flow graph. Since the computation of precise callgraph information also requires points-to information and vice versa [Bod18], the information for control-flow, callgraph, and points-to needs to be recomputed from scratch for each increment and thus, the approach only allows for a limited form of incrementality.

The computation of precise control-flow, callgraph, and points-to information can take minutes if not hours to compute on their own. Previous versions of Facebook's Infer [CD11] used separation logic to allow for the compositional analysis of heap-based programs. The approach computed bottom-up summaries using bi-abductive inference [CDOY09, BGS18], which can be used in different calling contexts. However, the approach aims at the verification of memory safety of C and C++ code and is restricted to this use case and has been replaced because of the lack of general application. The IncA [SEV16] approach allows for efficient incremental program analysis using graph patterns and incremental matching algorithms. IncA, however, is restricted to intra-procedural analysis.

In this chapter, we present IncAlyzer, an incremental static analysis approach that uses version control system information to precisely determine what code has changed from one version to another. Using the variability-aware VaRA framework [Sat23, SBS+23] for detecting interactions between code regions that convey a semantic meaning, e.g. detecting blocks of code that belong to a specific commit of a VCS, allows us to propagate the changes made to the source code into the underlying intermediate representation (IR) and access them at the IR level on which the static analysis is conducted on. After an initial computation of whole-program analysis information for the *complete analysis stack* (cf. Section 2.3) using the PhASAR framework, we are able to use ModAlyzer [SHB21] to persist all required information to conduct a concrete client data-flow analysis, i.e., control-flow, callgraph, points-to, class-hierarchy (in case of C++), and data-flow information [Bod18, SLHB21].

In each subsequent analysis run, IncAlyzer uses VCS information prepared by VaRA to precisely determine what piece of code has changed, and therefore, what analysis information must be recomputed and potentially repropagated. Contrary to the Reviser [AB14] approach, which only considers the data-flow parts of a client analysis for its incremental analysis and computes the code delta based on the inter-procedural control-flow graphs, our approach makes the complete client analysis stack (control-flow, callgraph, points-to, type-hierarchy and data-flow information) incremental and uses VCS information to obtain the code delta directly. If IncAlyzer recognizes that a code change has no impact on the semantics of the program while producing commit-annotated IR, it concludes that no (re)analysis is required at all—a situation that occurs more often than one might think as we will show in our evaluation. IncAlyzer has great potential to allow developers to check-in persisted static analysis results directly to the VCS managed code repository for each commit of a project which can then be both kept in sync throughout the continuous integration development of the project (cf. Section 1.3 and Figure 1.1). This has the advantage that each revision only needs to be analyzed once. Any developer can check-out a code revision accompanied by its respective up-to-date analysis results allowing them to check and reuse them for (incremental) analysis locally.

This further allows static analysis information for its corresponding commit to be viewed as a "certificate" that can be checked instantly for each given commit, depending on the precision and capabilities of the underlying client analysis, of course. One may even bind those "certificates" to the code using cryptographic hashing, to avoid accidental or intentional manipulation.

In summary, this chapter makes the following contributions:

- An incremental static analysis approach that utilizes version control information to analyze projects, that use continuous integration, with minimal effort from one increment to another.
- A C++-based implementation of IncAlyzer within the PhASAR [SHB19], ModAlyzer [SHB21], VaRA [Sat23, SBS+23], and LLVM [LA04] frameworks allowing one to incrementally analyze C and C++ projects managed with Git [Git19] using client analyses that can be expressed using distributive data-flow frameworks (e.g. IFDS [RHS95] or IDE [SRH96]).
- An experimental evaluation of our approach that shows its usefulness and effectiveness on real-world software projects.

6.2 Motivating Example

In the following, we present a motivating example which we use to showcase the challenges when statically analyzing projects that are developed using a CI/CD workflow and version control systems.

An excerpt of an exemplary project that uses a VCS is shown in Listing 6.1, Listing 6.2, and Listing 6.3. The project follows the common git flow conventions [Dri10] and its branching model is depicted as a graph in Figure 6.1. The project comprises a development branch named `develop` that consists of the commits D_1 and D_2. In addition, it comprises a feature branch named `sanitize` that consists of the two commits F_1 and F_2.

The commit D_1 shown in Listing 6.1 adds code that implements a small command-line application that retrieves student names associated with an id from a database and prints them to the command line. The id is passed as a command-line parameter to the application and is used to construct the SQL query in Line 15 which is send to a database. The application, however, is vulnerable to SQL injections because the command-line argument `argv[1]`, which may contain malicious contents, is not checked but directly concatenated to the string `q`. Carefully crafted malicious user inputs such as `10 OR TRUE; DROP TABLE name;` may have serious consequences and can alter the database in an undesired manner.

The feature branch `sanitize` is used to implement a fix such that the program is no longer vulnerable to SQL injections. Listing 6.2 shows the commit F_1 that provides parts of a fix by passing the string `input` to the `sanitize()` function—an implementation stub that aims at sanitizing user inputs. The commit F_2 shown in Listing 6.3 eventually provides a concrete implementation for the sanitization function and therefore, disperses the vulnerability. (A realistic SQL sanitization function would be much more complex but has been omitted for brevity.) To avoid cluttering, Listing 6.2 and Listing 6.3 only show excerpts of the respective diffs.

A static analysis that aims at analyzing projects developed in a CI/CD style incrementality needs to perform a full analysis run on a certain commit first and then has to update the analysis results for each subsequent commit. In order to update the analysis results, an incremental approach needs to determine the changes made to the code, recompute the static analysis information for control-flow, callgraph, points-to, type hierarchy, and data-flow that

has been invalidated and finally, needs to re-propagate those updated analysis information to obtain the final results. For a concrete client analysis, we assume the dependencies shown in Figure 2.8 that many realistic analyses impose. When implementing such an approach we face two challenges:

1. We need to update code and corresponding analysis information and keep them in sync for each subsequent commit such as $D_1 \rightarrow D_2$ or $F_1 \rightarrow F_2$.

2. We need to efficiently handle merges of different branches in which code and corresponding analysis information drifted further apart than in two subsequent commits. Figure 6.1 contains a merge of the two branches D_1 and F_2: $\overset{F_2}{D_1}\searrow\nearrow D_2$.

Note that branching can be modeled as two individual subsequent commits.

To detect the vulnerability in Listing 6.1, a taint analysis shall be performed. In the following, we sketch our approach by describing an analysis scenario along the route $D_1 \rightarrow F_1 \overset{D_1}{\searrow\nearrow_{F_2}} D_2$. Initially, no static analysis information for the project under analysis is available. Starting at commit D_1, a full whole-program analysis run is performed in which all analysis information that is required to answer a concrete client analysis, i.e., control-flow, callgraph, points-to, type hierarchy, and data-flow information, is computed. Next, the subsequent commit F_1 (c.f. Listing 6.2) introduces a new function **sanitize()**. In addition, we can observe that the **main()** function has been altered: the call to **Statement::executeQuery()** in Line 15 is replaced by Lines 29–31 in which the user input is passed to **sanitize()** and is then concatenated to **q** after which the resulting new string variable **qi** is passed to **Statement::executeQuery()**. By default, IncAlyzer invalidates analysis information on a function level. However, it can be parameterized to operate on an instruction or basic-block level instead. As F_1 changed instructions in **main()**, its analysis information must be invalidated. **main()**'s pointer information is recomputed and outgoing callgraph edges of the node representing **main()** are updated, such that a new call edge representing the call to **sanitize()** is introduced. When discovering the novel call to **sanitize()**, its points-to information is computed, too. Data-flow information is invalidated for **main()** and data-flows are repropagated along the previously updated callgraph.

The next commit F_2, whose diff is shown in Listing 6.3, replaces the **todo** comment in the **sanitize()** function with an actual implementation that we assume—for the sake of brevity—to sanitize the string referred to by the parameter **s**. Its pointer information is invalidated and updated. The callgraph node representing **sanitize()** has no outgoing edges that need to be followed. However, there is an incoming edge at Line 29. Incoming edges to modified functions tell the data-flow analysis where repropagation is necessary. The analysis detects that the flow fact representing **s** is now killed in the updated version of **sanitize()**. This new insight is now further propagated until the analysis detects the call to the sink **Statement::execute()** at which point there are no tainted variables. The propagation of data-flow facts could, in theory, stop as soon as the analysis detects data-flow facts that are a more coarse-grain representation of the facts of interest or in other words that are located higher up in the underlying lattice. While this would retain soundness of the analysis, it would—over time—introduce an unsustainable amount of imprecision and therefore, false positives: in the example, the analysis would still report a leak at Line 31. The data-flow

facts are hence propagated until the analysis finds that the facts are killed or until it detects merge points that bring data-flow facts on the same level (or higher up) of the lattice.

At last, the commits D_1 and F_2 are merged to produce D_2.

The analysis strategy is described in detail in Section 6.4.

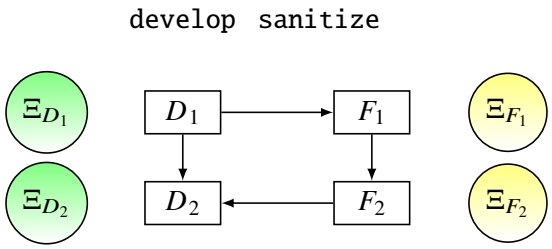


Figure 6.1: An exemplary git flow that shows a project's development and one of its feature branches `develop` and `sanitize`, respectively. Each increment of the project is accompanied by its static analysis information Ξ to answer a concrete client analysis.

```
1  int main(int argc, char **argv) {
2    // check command-line parameters
3    try {
4      Driver *driver;
5      Connection *con;
6      Statement *stmt;
7      ResultSet *res;
8      driver = get_driver_instance();
9      con = driver->connect("tcp://127.0.0.1:3306",
10                           "root", getPassword());
11     con->setSchema("MyDatabase");
12     stmt = con->createStatement();
13     string q = "SELECT name FROM students WHERE id=";
14     string input = argv[1];
15     res = stmt->executeQuery(q + input);
16     res->beforeFirst();
17     while (res->next()) {
18       cout << res->getString("name");
19     }
20     delete res;
21     delete stmt;
22     delete con;
23   } catch (SQLException &e) {
24     // handle exception
25   }
26   return 0;
27 }
```

Listing 6.1: Commit D_1

```
28 - res = stmt->executeQuery(q + input);
29 + sanitize(input);
30 + string qi = q + input;
31 + res = stmt->executeQuery(qi);
32
33 + void sanitize(string &s) {
34 +   /* TODO */
35 + }
```

Listing 6.2: Commit F_1

```
36 -   /* TODO */
37 +   replace(s.begin(), s.end(), '\'', ' ');
```

Listing 6.3: Commit F_2

6.3 Terminology and Notation

In the following, we introduce some terminology and notations that we use to describe our incremental analysis approach.

6.3.1 Model of a Version Control System

We use C_i to denote a commit of a project P, where C_0 represents the initial and C_n the most recent commit of the project. We use $\Delta_{c_i}^{c_{i+1}}(P) = \{\langle i_k, m\rangle\}$, where $m \in \{+, -\}$ to denote the set of instructions in the source code that did undergo changes from commit C_i to C_{i+1}. An instruction i_k might have been added or deleted. The modification of a instruction is modeled in terms of deletion and addition similar to version control systems' semantics.

6.3.2 Model of Analysis Information

The complete static analysis information is denoted as Ξ and thus Ξ_{C_i} represents the static analysis information at the commit C_i. Note that due to IncAlyzer's use of IFDS [RHS95] and IDE [SRH96] as well as ModAlyzer's [SHB21] compositional representations, we are able to express all static analysis information as graphs. Hence, we use $\Delta_{c_i}^{c_{i+1}}(\Xi) = \{\langle e_k, m\rangle\}$ to denote the difference in the static analysis information that hold at commit C_i and C_{i+1} in terms of graph edges e_k that have been added or removed.

6.4 Incremental Update Analysis

In this section, we present our approach to VCS-based incremental update analysis in detail.

6.4.1 Preparing Commit Metadata

State-of-the-art compiler and analysis frameworks do not have repository metadata at their disposal. Hence, before we can utilize repository meta data in our analysis framework, we first need to mine it from the target repository and second, need to integrate it with the intermediate representation (IR) that the static analysis is conducted on.

We achieve this by modifying the compiler such that during the construction of the IR, repository metadata, in the form of commit hashes, is mined from the repository and attached to the corresponding IR. In detail, the modified compiler determines the last change, i.e., commit for each *source-code line* by accessing repository metadata using git-blame, [1] for instance, and then annotates the commit hash to the respective IR instruction. This allows one to determine, for every IR instruction, the commit that introduced or last modified the corresponding source code. Furthermore, we can aggregate this information to determine the last commit which modified a function, by selecting the oldest commit that is not an ancestor of any other commit in the function.

6.4.2 Change Scenarios

Code changes constantly during software development, new source code gets added, existing code is changed or deleted. These code changes, however, often only partially invalidate previously computed analysis results. To reuse these previously computed results, we need to determine which parts of the analysis results can be reused safely and which need to be recomputed. We define four different *change scenarios* (**CS**) that trigger a (partial) reevaluation of the analysis results:

- **CS_1:** New code is added to the code base. In this case, we spawn the analysis at newly added functions. For all other functions, we compute the commit hash that most recently modified each function and reanalyze functions if the commit is newer than the one we stored during the previous analysis run. An example that depicts this situation is shown in Listing 6.4.

- **CS_2:** Existing code is changed. For changes to existing code, we use the same detection as with **CS_1**, by computing for every function the commit hash which most recently modified it. This situation is shown in Listing 6.5.

- **CS_3:** Code is deleted that had data-flow connections to code that still exists in the new revision. Listing 6.6 highlights this situation in which the complete definition for variable `y` is removed. However, code that made use of `y` is also modified and `y`'s usages are removed which allows us to detect this type of deletion. Compared to the previous two cases, deletions, in general, cannot be directly detected by computing the commit hash because the instructions corresponding to the original code are no longer present. However, if there are other instructions that previously depended on the deleted code, their data flows change, which (transitively) causes their commit

[1] Git-blame is a tool that annotates each line with the commit that last modified it.

hash to change, too. This allows us to identify and reanalyze all functions whose commit hash changed.

- **CS_4:** Code is deleted that had no direct data-flow relations to other instructions. In Listing 6.7, the entire definition for variable `z` is removed. This time, however, `z` had no direct users. Compared to the previous deletion case **CS_3**, we cannot obtain updated commit hashes based on other instructions. We detect this case by combining commit information with the number of instructions generated for each function. If the most recent commit hash did not change, but the number of instructions of a function did change, we know that code was deleted and we need to reanalyze this function. Note that simply using the number of instructions does not work, since changed code can lead to the same amount of instructions with different semantics. Only by combining the number of instructions with the commit hash we can be certain that code was deleted. Hypothetically, a compiler bug could generate a function with different semantics but identical commit hash and number of instructions which would not be reanalyzed, resulting in partially incorrect analysis result. However, this problem exists regardless of our approach.

```
1  // ===---- Revision C_i ----------=== //
2  #include <iostream>                  // > 3e8882e
3                                       // > 3e8882e
4  void foo() {                         // > 3e8882e
5    std::cout << "Hello, World!\n";    // > 3e8882e
6  }                                    // > 3e8882e
7  // ===---- Revision C_i+1 --------=== //
8  #include <iostream>                  // > 3e8882e
9                                       // > 3e8882e
10 void foo() {                         // > 3e8882e
11   std::cout << "Hello, World!\n";    // > 3e8882e
12 }                                    // > 3e8882e
13                                      // > ea8426c
14 int add(int i, int j) {              // > ea8426c
15   return i + j;                      // > ea8426c
16 }                                    // > ea8426c
```

Listing 6.4: Change scenario **CS_1**

```
17 // ===---- Revision C_i ----=== //
18 int getValue(int i) {                    // > c4d9b1a
19   if (i > 2) {                          // > c4d9b1a
20     return 42;                          // > c4d9b1a
21   }                                     // > c4d9b1a
22   return 13;                            // > c4d9b1a
23 }                                       // > c4d9b1a
24                                         // > c4d9b1a
25 int main(int argc, char **argv) {       // > c4d9b1a
26   int x = getValue(argc);               // > c4d9b1a
27   return 0;                             // > c4d9b1a
28 }                                       // > c4d9b1a
29 // ===---- Revision C_i+1 ----=== //
30 int getValue(int i) {                    // > c4d9b1a
31   if (i > 9001) {                       // > 0872f49
32     return 42;                          // > c4d9b1a
33   }                                     // > c4d9b1a
34   return 13;                            // > c4d9b1a
35 }                                       // > c4d9b1a
36                                         // > c4d9b1a
37 int main(int argc, char **argv) {       // > c4d9b1a
38   int x = getValue(argc);               // > c4d9b1a
39    return x;                            // > 0872f49
40 }                                       // > c4d9b1a
```

Listing 6.5: Change scenario $\mathbf{CS_2}$

```
41 // ===---- Revision C_i ----=== //
42 int main(int argc, char **argv) {       // > 5ba132e
43   int x = 42;                           // > 5ba132e
44   int y = x * 2 + argc;                 // > 5ba132e
45   int z = y + 9001;                     // > 5ba132e
46   z += 441;                             // > 5ba132e
47   return z;                             // > 5ba132e
48 }                                       // > 5ba132e
49 // ===---- Revision C_i+1 ----=== //
50 int main(int argc, char **argv) {       // > 5ba132e
51   int x = 42;                           // > 5ba132e
52   int z = x + 9001;                     // > 9fc62a4
53   z += 441;                             // > 5ba132e
54   return z;                             // > 5ba132e
55 }                                       // > 5ba132e
```

Listing 6.6: Change scenario $\mathbf{CS_3}$

```
56 // ===---- Revision C_i ----=== //
57 int main(int argc, char **argv) {     // > 74ed397
58   int x = 42;                         // > 74ed397
59   int y = x * 2 + argc;               // > 74ed397
60   int z = 9001;                       // > 74ed397
61   z += 441;                           // > 74ed397
62   return y;                           // > 74ed397
63 }                                     // > 74ed397
64 // ===---- Revision C_i+1 ----=== //
65 int main(int argc, char **argv) {     // > 74ed397
66   int x = 42;                         // > 74ed397
67   int y = x * 2 + argc;               // > 74ed397
68   return y;                           // > 74ed397
69 }                                     // > 74ed397
```

Listing 6.7: Change scenario **CS$_4$**

6.4.3 Compute Whole Program Information

IncAlyzer aims at computing all static analysis information that is required to serve a concrete client data-flow analysis incrementally such as to avoid unnecessary re-computations as much as possible. For the IncAlyzer approach we too assume the dependency model presented in Section 2.3 and shown in Figure 2.8, which many realistic client analyses impose and that also has been successfully applied by [SHB21, SLHB21].

Initially, there is no static analysis information for the project under analysis. Thus, IncAlyzer needs to compute and potentially persist, if desired, the static analysis information Ξ_{C_i} for a given commit C_i that shall serve as the point of initialization of static analysis information using whole program analysis. IncAlyzer uses the ModAlyzer approach presented in Chapter 5 to modularly compute analysis information.

ModAlyzer produces, as a result, a modular function- and module-level in-memory, graph-based summary representation of the complete static analysis stack Ξ_{C_i}. These summaries (that can also be persisted for later use) are typically pre-computed for a program's library parts and then applied while analyzing the actual application code allowing for significant performance improvements. IncAlyzer uses these graph-based summaries to make the full static analysis stack incremental.

6.4.4 Compute Incremental Updates

We next describe how IncAlyzer computes incremental updates for a subsequent commit C_{i+1} based on the static analysis information Ξ_{C_i}. In order to make the analysis run incrementally, IncAlyzer first computes the change set $I = \Delta_{c_i}^{C_{i+1}}(P)$ using VaRA as described in Section 6.4.1 and Section 6.4.2. Using I, we effectively need to map the changes made to the code to the changes in terms of static analysis information in order to compute $\Xi_{C_{i+1}}$ while reusing as much information of Ξ_{C_i} as possible. IncAlyzer computes $\Xi_{C_{i+1}}$ with help of I and Ξ_{C_i} using an (i) *invalidate* – (ii) *recompute* – (iii) *update* strategy.

To assess I's effects on the concrete client analysis and to compute $\Xi_{C_{i+1}}$, we need to make suitable updates to the data-flow information to answer the client analysis a static analysis user is eventually interested in. As the IFDS/IDE frameworks compute the data-flow information by building an exploded super-graph that is based on the inter-procedural control-flow graph, we need to compute the diff of the inter-procedural control-flow graphs of C_i and C_{i+1} to be able to decide what pieces of information need to be invalidated and recomputed. To determine the unknown inter-procedural control-flow graph of C_{i+1}, we need to make suitable updates to C_i's callgraph. This requires information on points-to graphs as per ModAlyzer's compositional representation. To update the points-to graphs, we need information on the source code that is provided by I. We describe IncAlyzer's strategy in what follows.

Type Hierarchy Information First, IncAlyzer iterates I, the set of modified instructions and identifies instructions that allocate types previously unknown in C_i. The types that have been introduced in C_{i+1} are added to the type hierarchy. Virtual function tables are updated according to instructions whose hosting virtual function f_i has been added or removed in C_{i+1}. Indirect callsites that depend on f_i must be invalidated according to the dependencies maintained by ModAlyzer.

Points-to Information Next, I, the set of modified instructions, is iterated and all function-wise pointer assignment graphs that correspond to the individual functions f_i that did undergo changes, i.e, that include at least one $i_k \in I$, are invalidated and recomputed. The points-to information that is based on those pointer-assignment graphs (PAGs) is recomputed anew, too.

Callgraph Information Since modifications made to one piece of analysis information may heavily influence other pieces of analysis information, which, in turn, may influence further information and so on, IncAlyzer computes the incremental update for the various helper analyses using a worklist algorithm that follows the *invalidate*, *recompute*, *update* strategy. The worklist algorithm is shown in Algorithm 5. Once the worklist is empty and the algorithm's fixed point is reached, the helper analyses are up-to-date and represent the state of C_{i+1}. Information exchange between the various types of helper analyses is implemented using a mediator pattern [Gur23]. The worklist algorithm uses a mediator that triggers invalidation, recomputation, and updates for the type hierarchy, points-to, and callgraph analyses, respectively and uses a revise info type $R := I \times T$ for communication. Instances of R carry information about an instruction $i_k \in I$ and its type of information $T := \text{TypeHierarchy}, \text{PointsTo}, \text{Callgraph}, \text{All}$ that must be invalidated and recomputed. The worklist is initialized with $\{i_k, \text{All}\}$, the set of all modified instructions invalidating all previously computed information for these instructions except data flows. The algorithm then removes an element $\langle i_k, t_i \rangle$ from the worklist and triggers the respective helper analyses specified by t_i to invalidate and update information on i_k. Since updated information may also invalidate other analysis information, the algorithms queries each helper analysis for further revise infos to determine what else might have been invalidated through the update.

Any potential revise infos obtained by queries to the helper analyses are then added to the worklist, again.

```
120 Function updateHelperAnalyses(C_i, C_{i+1}, T, P, C):
121     S = computeDelta (C_i, C_{i+1});
122     R = makeReviseInfos (S);
123     while R != ∅ do
124         ⟨i_k, t_k⟩ = R.pop();
125         if t_k == TypeHierarchy || t_k == All then
126             T.invalidateAndRecompute(i_k);
127             R ∪= T.invalidates();
128         if t_k == PointsTo || t_k == All then
129             P.invalidateAndRecompute(i_k);
130             R ∪= P.invalidates();
131         if t_k == Callgraph || t_k == All then
132             C.invalidateAndRecompute(i_k);
133             R ∪= C.invalidates();
```

Algorithm 5: Worklist algorithm that employs the *mediator pattern* [Gur23] to incrementally update the helper analyses.

Data-Flow Information Once IncAlyzer computed the incremental update of the helper analyses using the fixed-point algorithm shown in Algorithm 5, it updates data flows by applying the Reviser approach for incrementally updating IFDS/IDE-based data-flow analysis presented by Arzt et al. [AB14]. The algorithms for incrementally computing data flows are shown in Algorithm 6, Algorithm 7, Algorithm 8, and Algorithm 9.

Reviser's general idea is to consider all ICFG nodes that are transitively reachable from a changed node as *affected*. The data-flow information for all affected nodes must be updated to reflect the new software revision C_{i+1}. Algorithm 6 shows the entry point and initialization of the incremental data-flow computation.

First, all path edges (also known as *jump functions*) of the ESG that correspond to ICFG nodes that have been removed in C_{i+1} are cleared, along with their corresponding value. Also any summary for a node $n \in N^-$ is cleared (cf. Lines 137 – 140). In the event that no control-flow edges have been altered for the software increment from C_i to C_{i+1}, data-flow computation can stop here.

In case an ICFG predecessor node n_1 of a modified control flow is part of a loop, the predecessor node of the loops entry point is chosen as a safe overapproximation. Otherwise, n_1 is added to the set of affected nodes N (cf. lines 149 – 153). Next, the **forwardTabulateSLRPs** subroutine is called in **update** mode to process a worklist of path edges that contain affected nodes (cf. Lines 156 – 161). For efficiency reasons, **forwardTabulateSLRPs** propagates and clears outdated IDE information for changed nodes in one step, instead of running in two individual analysis passes. Once IDE information has been cleared for all modified ICFG nodes and transitively reachable affected nodes, a worklist is scheduled to spawn

recomputation of data flows starting from the predecessors of changed nodes n, which is then processed by another call to **forwardTabulateSLRPs** in **compute** mode, which represents the original IDE iterations (cf. Lines 162 – 168). At this point, all path edges (or *jump functions*) affected by the code changes $\Delta_{C_i}^{C_{i+1}}$ are up-to-date and represent the effects of C_{i+1}.

As a last step, their respective evaluations are cleared (cf. 169 – 172) and their values are recomputed using the original *phase II* of the IDE algorithm as indicated in Algorithm 6 Lines 173 – 173 and shown in Algorithm 9.

Answering the Client Analysis Once the data flows have been updated according to the incremental IDE algorithm described above, IncAlyzer has computed the full update $\Xi_{C_{i+1}}$ for C_i's subsequent commit C_{i+1}. These information can now be used to answer client queries. For the commit F_2 in our motivating example in Listing 6.1, Listing 6.2, and Listing 6.3 a taint analysis would find that variable **qi** is no longer tainted. $\Xi_{C_{i+1}}$ can also be persisted using ModAlyzer and checked into the target program's repository to make the analysis information available to anyone who checks out C_{i+1}. This enables the workflow that has been described in Section 1.3.

6.5 Implementation

We implemented IncAlyzer on top of PhASAR [SHB19], ModAlyzer [SHB21], VaRA [Sat23, SBS+23], and LLVM [LA04].

The VaRA framework [Sat23, SBS+23] has been built for making commit information of a given target project available at the LLVM IR level. The VaRA project is essentially a fork of LLVM that not only integrates repository metadata in form of commits into the compiler's IR, but also provides a modified version of the Clang compiler (Vlang), which automatically adds a project's Git-blame information to the IR using LLVM's metadata mechanism. Git blame provides information on which commit last modified each line of a source file checked into a Git repository. Vlang collects this information and adds it as custom AST nodes into compiler's internal representation. During code generation, VaRA adds the git-blame information to the respective LLVM IR instructions.

For conducting static program analysis on the commit-annotated LLVM IR, we use PhASAR, which provides its users with all required infrastructure to implement deep static program analyses and offers various analyses for computing (inter-procedural) control-flow graphs, callgraphs, type hierarchy, and points-to information. The incremental IDE solver has been implemented by extending PhASAR's existing IFDS [RHS95]/IDE [SRH96] implementations; it is essentially a slightly modified version of the implementation presented by Arzt et al. in [AB14].

We implemented the new incremental helper analyses by providing appropriate overrides for the type hierarchy, points-to and (inter-procedural) control-flow graph, and callgraph interfaces. Most of PhASAR's analysis type implemented for LLVM IR directly operate on LLVM IR types. To make our new implementations incremental, we introduced special updatable wrapper types that allow for exchanging the actual underlying LLVM entity. This

```
134 Function computeIncrementalUpdate(C_i, C_{i+1}):
135     S = Δ_{c_i}^{C_{i+1}}(P);
136     F = getModifiedFunctions (S);
137     forall n ∈ N^-; d_1, d_2 ∈ D; v ∈ V do
138         PathEdge := PathEdge \ ⟨d_1, n, d_2⟩;
139         val := val \ ⟨n, d_1, v⟩;
140         EndSum[⟨n, d_1⟩] = ∅;
141     if E^+ ∪ E^- = ∅ then
142         return;
143     changedNodes := ∅;
144     chgEndSums := ∅;
145     WorkList := ∅;
146     allChangedNs := ∅;
147     oldES := EndSummaries;
148     forall ⟨n_1, n_2⟩ ∈ (E^+ ∪ E^- ∪ E^#) do
149         if isPartOfLoop(n_1) then
150             ls := getLoopStart(n_1);
151             N := getPredsOf(ls);
152         else
153             N := {n_1};
154         forall n ∈ N, d_1, d_2 : (d_1, n, d_2) ∈ PathEdge do
155             WorkList := WorkList ∪ {⟨d_1, n, d_2⟩};
156         forwardTabulateSLRPs(Update, c^new);
157         if ∃e_p ∈ e_{proc(d_2)} : e_p ∈ (N^+ ∪ N^-) ∨ ∃d_1 ∈ D, s_p ∈ s_{proc(d_2)} :
              oldES[⟨s_p, d_1⟩] ≠ EndSummaries[⟨s_p, d_1⟩] then
158             forall c ∈ callSite(proc(d_2)), d ∈ succs(c) do
159                 E^# = E^# ∪ ⟨c, d⟩;
160         allChangedNs = allChangedNs ∪ changedNodes;
161         changedNodes = ∅;
162     forall n ∈ allChangedNs do
163         preds := {m : m → n ∈ c^new};
164         if |preds| ≥ 2 then
165             forall m ∈ preds do
166                 forall d_1, d_2 : (d_1, md_2) ∈ PathEdge do
167                     WorkList := WorkList ∪ {⟨d_1, md_2⟩};
168     forwardTabulateSLRPs(Compute, cfg);
169     forall n_1 ∈ allChangedNs do
170         forall n_i : ∃n_1, ··· , n_i ∈ N; ∀i : n_i → n_{i+1} ∈ cfg do
171             forall d ∈ D; val(n, d) ≠ ∅ do
172                 val(n, d) = ∅;
173     forall n_1 ∈ allChangedNs do
          ; // run original Phase II for (n, d) - Algorithm 9,
           see [SRH96], page 149
```

Algorithm 6: Modified IFDS/IDE algorithm for computing data flows (phase I of the IDE algorithm) incrementally. Reproduced from [AB14].

```
174 Function forwardTabulateSLRPs(mode, cfg):
175     while WorkList ≠ ∅ do
176         Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList;
177         switch n do
178             case n ∈ Call_p do
179                 if d_2 = ϵ then
180                     maybeClearAndPropagate(⟨d_1, retSite(n), ϵ⟩);
181                     continue;
182                 forall d_3 ∈ passArgs(⟨n, d_2⟩) do
183                     propagate(⟨d_3, s_calledProc(n), d_3⟩);
184                     Incoming[⟨s_calledProc(n), d_3⟩]∪ := ⟨n, d_2⟩;
185                     forall ⟨e_p, d_4⟩ ∈ EndSum[⟨s_callee(n), d_3⟩] do
186                         forall d_5 ∈ retVal(⟨e_p, d_4⟩, ⟨n, d_2⟩) do
187                             maybeClearAndPropagate(⟨d_1, retSite(n), d_5⟩);
188                         if retVal(⟨e_p, d_4⟩, ⟨n, d_2⟩) = ∅ ∧ mode = Update then
189                             maybeClearAndPropagate(⟨d_1, retSite(n), ϵ⟩);
190             case n ∈ e_p do
191                 if mode = Update then
192                     if ⟨s_p, d_1⟩ ∉ chgEndSums then
193                         chgEndsums =
                          chgEndSums ∪ ⟨s_p, d_1⟩ EndSum[⟨s_p, d_1⟩] := ∅;
194                 if d_2 ≠ ϵ then
195                     EndSum[⟨s_p, d_1⟩] := EndSum[⟨s_p, d_1⟩] ∪ ⟨e_p, d_2⟩;
196                 if mode = Compute then
197                     forall ⟨c, d_4⟩ ∈ Incoming[⟨s_p, d_1⟩] do
198                         if d_2 = ϵ then
199                             propagate(⟨d_1, retSite(c), ϵ⟩);
200                             continue;
201                         returnVals = returnVal(⟨e_p, d_2⟩, ⟨c, d_4⟩);
202                         forall d_5 ∈ returnVals, d_3 : ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈
                          PathEdge do
203                             propagate(⟨d_3, retSite(n), d_5⟩);
204             case n ∈ (N_p \ Call_p \ {e_p}) do
205                 if d_2 = ϵ then
206                     forall m : n → m ∈ c^new do
207                         maybeClearAndPropagate(⟨d_1, m, ϵ⟩);
208                     continue;
209                 succs = {⟨m, d_3⟩ : n → m ∈ cfg ∧ d_3 ∈ flow(⟨n, d_2⟩, π)};
210                 forall ⟨m, d_3⟩ ∈ succs do
211                     maybeClearAndPropagate((⟨d_1, m, d_3⟩));
212                 if |succs| = 0 ∧ mode = Update then
213                     maybeClearAndPropagate(⟨d_1, m, ϵ⟩);
```

Algorithm 7: Incremental IDE iterations. Reproduced from [AB14].

```
214 Function maybeClearAndPropagate(e := ⟨d1, n, d2⟩):
215     if mode = Update then
216         if n ∉ changedNodes then
217             changedNodes := changedNodes ∪ {n};
218             forall d3 ∈ D : (d1, n, d3) ∈ PathEdge do
219                 PathEdge := PathEdge \ (d1, n, d3);
220     if d2 ≠ ϵ then
221         propagate(e);
```

Algorithm 8: Incremental IDE's clear and propagate procedure. Reproduced from [AB14].

```
222 Function computeValues():
        /* phase II(i)                                                    */
223     foreach n ∈ N# do
224         val(n) = ⊤;
225     val(⟨s_main, Λ⟩) = ⊤;
226     NodeWorkList = {⟨s_main, Λ⟩};
227     while NodeList ≠ ∅ do
228         Select and remove an exploded-graph node ⟨n, d⟩ from NodeWorkList;
229         switch n do
230             case n is the start node of p do
231                 foreach c that is a call node inside p do
232                     foreach d' such that f' = JumpFn(⟨n, d⟩ → ⟨c, d'⟩) ≠ λl.⊤ do
233                         propagateValue(⟨c, d'⟩, f'(val(⟨s_p, d⟩)));
234             case n is callnode in p, calling procedure q do
235                 foreach d' such that ⟨n, d⟩ → ⟨s_q, d'⟩ ∈ E# do
236                     propagateValue(⟨s_q, d'⟩, EdgeFn(⟨n, d⟩ →
                          ⟨s_q, d'⟩))(val(⟨n, d⟩)));
        /* phase II(ii)                                                   */
237     foreach n in a procedure p, that is not a call or start node do
238         foreach d', d such that f' = JumpFn(⟨s_p, d'⟩ → ⟨n, d⟩) ≠ λl.⊤ do
239             val(⟨n, d⟩) = val(⟨n, d⟩) ⊓ f'(val(⟨s_p, d'⟩));
240 Function propagateValue(n, v):
241     v' = v ⊓ val(n);
242     if v' ≠ val(n) then
243         val(n) = v';
244         Insert n into NodeWorkList;
```

Algorithm 9: IDE's original value computation procedure (phase II of the IDE algorithm). Reproduced from [SRH96].

```
1  class Revisable {
2  public:
3    std::vector<ReviseInfo> ReviseInfos;
4    virtual ~Revisable() = default;
5    virtual void invalidate(UpdatableInstruction) = 0;
6    virtual void invalidate(UpdatableBasicBlock) = 0;
7    virtual void invalidate(UpdatableFunction) = 0;
8    virtual inline std::vector<ReviseInfo> revise() {
9      auto ReviseInfosCpy = ReviseInfos;
10     ReviseInfos.clear();
11     return ReviseInfosCpy;
12   }
13 };
```

Listing 6.8: Interface for updating helper analyses.

is required to keep the graph-based information that did not change from one commit to another valid. In that case, when updating to a new commit, we only need to map the wrappers' underlying LLVM pointers to the counterpart of the new IR. Graph nodes of the various involved analyses that did undergo changes is updated by the respective analysis. To model the algorithm Algorithm 5, we added the new **Revisable** interface that we additionally implemented for all of the above helper analyses. The interface is shown in Listing 6.8 and is used to implement the mediator pattern [Gur23] via the exchange of **ReviseInfo** objects. The mediator triggers invalidation and update computations for pieces of code that did undergo changes and communicates the status of the incremental update to the helper analyses. Other pieces of code that are, in turn, invalidated by an update computation are communicated back to the mediator using the **revise** function. This function returns instances of the **ReviseInfo** type that carries information about the pieces of code and their concrete type of helper analyses (potentially all of them) that have been affected by the update and need to be revised. The revise infos are then added back to the worklist by the mediator and iteration continues until a fixed-point is reached. This algorithm is guaranteed to terminate and in the worst case, all previously computed analysis information is invalidated and recomputed from scratch.

6.6 Evaluation

IncAlyzer extends PhASAR and ModAlyzer, and enables client analyses to be run incrementally, reducing the time to analyze a subsequent version of a project. We demonstrate the usefulness of this approach by running three different client analyses on 50 subsequent revisions of five open-source projects to determine analysis speed and precision compared to a traditional matching whole-program analysis.

6.6.1 Research Questions

IncAlyzer aims at reducing the analysis times for continuously running static analysis, like in a continuous integration environment, without compromising correctness. To evaluate and explore the capabilities of our approach, we propose the following research questions:

RQ_{10}	*How much analysis time can be saved by running an analysis incrementally?*
RQ_{11}	*What impact do* change characteristics *have on the applicability of our incremental approach?*
RQ_{12}	*How much do the underlying helper analyses benefit from being run incrementally?*
RQ_{13}	*Are the results produced by IncAlyzer equivalent to those of a matching traditional whole-program analysis?*

We answer these questions in the following sections: RQ_{10} by comparing the running times of IncAlyzer with those of a matching WPA for three different static client analyses. RQ_{11} by a manual investigation of the various project revisions and associated code changes, and by relating those to the analysis times of the corresponding IncAlyzer runs. To answer RQ_{12}, we analyze separately the running times of each analysis involved in the complete static analysis stack and determine how much each of them can benefit from being run incrementally. We answer RQ_{13} by performing an in-memory comparison between the client analyses' reports issued by IncAlyzer run and those of a matching WPA.

6.6.2 Experimental Setup

In order to evaluate our research questions, we use the following experimental setup. We run three different client analyses on 50 subsequent commits for each of the five different projects, once using the traditional whole-program approach and once using IncAlyzer. We then compare the results and the time it took to analyze each commit. To select 50 commits for each project, we used the most recent commit C_i of each project's corresponding *main* branch and the 49 commits that precede C_i. These commits hence may also contain *merge commits*. The commits that we chose, their commit hash's fingerprint as well as a brief description of their nature are shown in Table 6.4, Table 6.5, Table 6.6, Table 6.7, and Table 6.8.

Client analyses. We selected three different types of client analyses commonly used with PhASAR:

- **Typestate**: An IDE based typestate analysis [Str83, SY86]—a typestate analysis or protocol analysis is a static analysis that tracks variables of a certain type and their associated states through the program. Typestates define sequences of operations that may be performed upon a variable. The state information associated with each

variable is used to determine—at compile-time—the validity of operations invoked upon variables.

- **Taint**: An IFDS based taint analysis—a taint analysis tracks variables through the program that have been tainted by one or more so-called *sources* and reports a leak whenever it detects that a tainted variable is used by a *sink*. Sources and sinks can be parameterized and may be functions or instructions.

- **LCA**: An IDE based linear constant analysis—a linear constant analysis track constant variable or variables that linearly ($f(x) = ax+b$, with a and b constant literals) depend on constant values and their respective values through the program.

Besides the IFDS-based taint analysis, we also include two IDE-based analyses that crucially utilize IDE's value domain. This stresses IncAlyzer's implementation of the incremental IDE solver for computing data flows since bugs in IDE's complex edge function components would be directly uncovered when comparing the analyses' result reports.

Experimental process. In general, the goal of our experiments is to determine whether IncAlyzer can produce equivalent results in a shorter time compared to the standard WPA approach. Therefore, our experiments need to run each client analysis once as WPA and once incrementally based on program changes.

In more detail, for each commit C_i, we run the client analysis as WPA on the previous revision C_{i-1} *and* the current revision C_i. This produces the base line analysis results and running time measurements for the change introduced by C_i. Afterwards, we run the client analysis with IncAlyzer, starting from the correct analysis result of the previous revision and computing the new result for the change (new revision C_i) incrementally. By that, we produce the incremental analysis results and running time measurements, which we compare to the base line produced by the traditional matching whole-program analysis. For the 2nd to the 50th commit, we base each incremental analysis on the results of the previous whole-program results.[2]

We automatically compare the analysis results produced by IncAlyzer with those produced by a matching WPA using an in-memory comparison. For each entry in a client analysis' result report, we check whether the findings computed by IncAlyzer and the matching WPA coincide.

The idea behind this experimental process is similar to an inductive proof. We know, by selecting a working client analysis, that we can correctly analyze a single revision. We then, similar to an induction step, show that if we have a correct analysis result for the previous revision, we can produce an equivalent result for the new revision.

Target Subject Selection. We evaluate IncAlyzer on five open-source C projects of various program sizes. Table 6.1 provides an overview of the different projects along with their relevant characteristics.

[2]This is because of a current limitation in the implementation of our evaluation setup.

Table 6.1: Evaluation Targets. Data on lines of code of the software repository's respective main branch and number of commits as of June 9th, 2023. Lines of code measured with Unix' `cloc`.

Program	Domain	LOC	Commits	Samples
BROTLI	Compression	56,377	1,074	50
CURL	Web	264,914	30,518	50
GZIP	Compression	8,909	691	50
HTOP	Visualization	31,445	2,958	50
XZ	Compression	63,874	2,547	50

It is important to note here that the size and characteristics of the change set of two subsequent project revisions are likely having a high impact on how an incremental analysis performs. Since it was initially completely unclear which characteristics influence the analysis by how much, we selected change sets of 50 consecutive commits for each of the five projects. The commit hashes' fingerprints as well as a brief description of their nature are shown in Table 6.4, Table 6.5, Table 6.6, Table 6.7, and Table 6.8 in Section 6.6.

Execution environment. We measured our experiments on an AMD Ryzen 9 7950X 16-Core Processor machine with 128GB main memory running Ubuntu 22.04. We measured each analysis run 3 times and computed the average to determine the time it took to execute the analysis.

6.6.3 RQ_{10}: Performance

We address RQ_{10} by comparing the analysis times of each client analysis run with IncAlyzer to baseline measurements from running a matching WPA. For each change, i.e., commit, we measured the time it took to analyze the specific commit and aggregate the results for each project by computing the mean analysis time.

Table 6.2 depicts the aggregated mean analysis times for each target project and client analysis. In Section 6.10, we show the detailed running times of the five projects for each of the 50 respective target commits in Figure 6.2, Figure 6.3, Figure 6.4, Figure 6.5, and Figure 6.6. With these results, we get a good overview on what an IncAlyzer run does on average in terms of running time and see that analysis times are approximately two and a half times slower (0.43 speedup) in the worst case and up to 44 times faster than a matching WPA run. Interestingly, IncAlyzer appears to slow down client analyses on smaller projects with particularly small data-flow domains. The typestate analysis' runtime performance suffers for three projects except the larger CURL project while it behaves neutral to XZ. Similarly, IncAlyzer's performance degrades for the relatively small BROTLI project for all client analyses except taint analysis which is known for its notorious large data-flow domain. For larger target programs and client analyses with large data-flow domains, however, IncAlyzer starts to excel and is able to speed up analysis times. IncAlyzer achieves a speedup of almost 44 times when conducting a taint analysis on XZ.

Table 6.2: Overview of the average running times and relative speed-ups achieved by running the three client analyses as WPA and with IncAlyzer. All running times in milliseconds.

	Taint			LCA			Typestate		
	WPA	INC	Speedup	WPA	INC	Speedup	WPA	INC	Speedup
BROTLI	1,305	589	2.22	255	344	0.74	130	310	0.43
CURL	12,929,961	882,396	14.65	16,159	3,467	4.66	4,065	979	4.15
GZIP	6,537,968	948,513	6.89	8,501	1,798	4.73	223	360	0.62
HTOP	2,492,921	266,313	9.36	27,372	5,891	4.65	498	581	0.86
XZ	1,812,858	41,349	43.84	2,981	497	6.00	390	375	1.04

While XZ's set of target commits contain only very few commits that change the code directly, one cannot simply thoughtlessly exempt them from analysis entirely. Projects written in C and C++ frequently use code that is generated at build time, for instance, to populate versioning information. Thus, even commits that do not modify source files directly, might still modify the code that is generated—LLVM IR in our case—and eventually analyzed.

To answer RQ_{10}, we can summarize that IncAlyzer allows one to greatly speed up client analyses that would otherwise require vast resources when run using WPA (for the commits shown in Table 6.4, Table 6.5, Table 6.6, Table 6.7, and Table 6.8).

6.6.4 RQ_{13}: Correctness

Each of the client analyses' implementations contains a dedicated data structure that is used to accumulate the relevant analysis results. It is used as an analysis report that will be eventually reported to the user. The typestate analysis, for instance, keeps track of protocol violations for the parametrizable type(s) of interest. While conducting the experiments to answer RQ_{10}, we additionally performed an in-memory comparison of these client analyses' result reports to answer RQ_{13}. Luckily, we could indeed confirm that the results produced by IncAlyzer coincide with those produced by the corresponding analysis run in whole-program mode: In every instance, the analysis reports computed in whole-program mode are identical to those computed by IncAlyzer, which indicates correctness *and* full precision. IncAlyzer results are hence from-scratch consistent [SCS24].

6.6.5 RQ_{11}: Change Characteristics

The changes made to a project can have a significant influence on how many analysis results computed in a previous run can be reused, potentially impacting the usefulness of IncAlyzer. RQ_{11} aims at answering in detail the incremental gains of IncAlyzer and how they correspond with the nature of a commit. We manually analyzed the size and nature of each of the 50 commits for each of the five target subjects.

Our manual investigation of the commits shown in Table 6.4, Table 6.5, Table 6.6, Table 6.7, and Table 6.8 in Section 6.10 shows that most commits are typically small and local, so small, in fact, that they frequently do not change the source code at all or at least

not directly. Injecting code that is generated at build time is common pattern in C and C++ and hence, even an empty commit might still change the compiled code and thus also the analyzed LLVM IR code. The commits we analyzed typically revolved around a handful of functions at most. Changes to pointer variables are rare and even caller-callee relations, i.e., callgraph information, are changed rather infrequently. Merge commits might bring larger source code changes, however, do not occur frequently in our set of target commits. Our manual investigation was able to confirm the results produced in previous works by Swanson [Swa76], Hattori et al. [HL08], and Brindescu et al. [BCSD14] that showed that a program typically only undergoes small changes from one revision to the next. The IncAlyzer approach leverages this to avoid unnecessary repeated re-analysis. However, commits can sometimes also change a large amount of files of the target code base [HL08]. We have been able to confirm that the performance improvements that IncAlyzer provides degrade with the number of functions being changed and their role in the program. If a function that is called several times in the program does change, the performance naturally starts to degrade. We did not encounter the situation frequently in our set of target commits, but if functions that are called many times and under various calling contexts are being changed, IncAlyzer's performance can degrade to a point at which it is eventually cheaper to run a full whole-program analysis (cf. Section 6.10). Still, IncAlyzer has proven highly beneficial in reducing analysis times for the overwhelming majority of commits.

6.6.6 RQ_{12}: Helper Analyses

As visualized in Figure 2.8, each analysis actually comprises different helper analyses, which are commonly reused by different client analyses. The IncAlyzer approach also makes each of the different helper analyses incremental, potentially reducing their analysis time. However, it is unclear which helper analyses profit from being run incrementally and contribute to the overall gains we saw in Table 6.2. Furthermore, since the helper analyses are reused by many different client analyses, it is interesting to see which analysis actually profits and by how much. This gives us an indication how much other client data-flow analyses could profit from IncAlyzer.

We investigate RQ_{12} by analyzing the speed-up distributions of the helper analyses. We added a fine-grain instrumentation using our approach developed in [SLHB19] allowing us to measure the running times of the individual helper analyses. We collected the measurements together with the overall running times of RQ_{10}.

However, since the behavior of the helper analyses is not influenced by the concrete client analysis, we took the average across the measures for the three different client analyses for all commits of a project and computed the running time difference between the WPA and IncAlyzer runs as a percentage. The data is shown in Table 6.3.

The data shows that managing the code in an IncAlyzer setup does come with an overhead. This is because the implementation for the program representation and code management (IRDB) is responsible for determining the code delta by calling into the VaRA framework [Sat23] and for remapping the intermediate representation of the old revision that is not affected by a code change to the intermediate representation of the new revision. The computations of the programs' type hierarchies remain unaffected since the

Table 6.3: Running time differences between WPA and IncAlyzer for the various helper analyses: code management (IRDB), computation of type hierarchy (TH), points-to information (PT), inter-procedural control-flow graph (ICFG).

Helper Analysis	IRDB	TH	PT	ICFG
BROTLI	+627%	0%	-71%	-79%
CURL	+664%	0%	-88%	-76%
GZIP	+589%	0%	-79%	-72%
HTOP	+671%	0%	-83%	-69%
XZ	+624%	0%	-89%	-87%

target programs are all written in C and there is nothing to compute for the type hierarchy analysis. Points-to and callgraph analyses, however, seem to heavily benefit from being run in an incremental manner. Again, most commits only apply minor code modifications and rarely change pointer variables or caller-callee relationships. This is especially beneficial when it comes to points-to information which is notoriously expensive to compute.

6.7 Threats to Validity

Internal Validity. The correct invalidation of static analysis information computed in a previous analysis run is depending on the accuracy and correctness of the mapping of commit metadata and the correct handling of the change scenarios. To ensure that IncAlyzer maps the commits correctly to the corresponding intermediate representation, we implemented an alternative approach that recomputes the commit metadata from debug metadata, provided by the compiler. We determined that our commit mapping is as precise as the compilers own debug metadata. Furthermore, to ensure IncAlyzer handles the commit mapping and the change scenarios correctly, we checked that the static analysis results produced by IncAlyzer are equivalent to the results produced by traditional WPA approach. Our results show that the precision of all evaluated client analyses is not harmed by IncAlyzer, since they produce equivalent results as if they had been run with a matching WPA approach.

External Validity. In our evaluation, we utilized IncAlyzer to run three different client analyses on five real-world open-source projects, analyzing in total 50 changes. We cannot guarantee that the speedups seen in our evaluation generalize to all software projects and to any kind of change. However, our data and the results of previous work of the mining software repositories (MSR) community suggests that there is a high likelihood that IncAlyzer can speed up analysis times in many cases, especially for larger projects.

In addition, we cannot be sure that IncAlyzer's speedups generalize to all client analyses. We used three different client analyses that vary in complexity and our evaluation shows that not every client analysis does profit from being run in an incremental way with IncAlyzer. RQ_{12} shows that the particularly expensive helper analysis that computes points-to

information profits from being run incrementally. This also matches our observation that client analyses with large data-flow domains benefit from IncAlyzer.

6.8 Related Work

An approach that uses path abstraction is presented in [MR14]. The approach encodes program paths as sets of constraints encoded as boolean formulars which are solved using a SAT solver. The satisfiability is used to drive the analysis further. After an initial analysis of the complete source code, the approach manages a mapping of the boolean input constraints to the SAT solver results. If constraints have already been solved in subsequent analysis runs, the results can be used directly. The approach aims at improving the abstraction used for analysis in order to reduce the amount of recomputation necessary for an incremental update.

The DRed$_L$ algorithm presented by Szabo et al. [SBEV18] supports incremental maintenance of recursive lattice-value aggregation in Datalog. It hence lifts the expressiveness of incremental analysis in Datalog by allowing fix points to be computed over lattices other than the powerset lattice. DRed$_L$ has been evaluated for intra-procedural analyses formulated for Java only.

Arzt et al. developed an analysis approach called Reviser. The work formulates an algorithm for incremental updates following a "clear-and-propagate" philosophy based on the inter-procedural control-flow graph [AB14]. The approach clears analysis information of statements that have been changed and statements that are reachable from such modified statements and then, re-propagates analysis information. Reviser's algorithm is formulated as an extension to the IDE framework which allows to solve distributive inter-procedural data-flow problems. However, Reviser is only concerned with data-flow information, it assumes a somehow existing call-graph and points-to information. Call-graph and points-to information still need complete re-computation for each change made to the code. In addition, the Reviser approach omits the challenging task of persisting analysis information. The analysis information are *discarded* after analysis shutdown and thus, the approach cannot be applied directly to compute incremental updates in a real-world setting. IncAlyzer utilizes a modified version of the Reviser approach to compute data flows incrementally.

Conway et al. presents incremental algorithms for the inter-procedural analysis of safety properties [CNDE05]. They use simple incremental automaton-based program analysis to react to changes made to the code. The algorithms, which are incremental versions of standard model checking algorithms, maintain a derivation graph to record the analysis. The graph is successively "repaired" with the incremental changes.

An incremental approach that uses a parallelized algorithm to compute points-to information is presented in [LHR19]. Similar to our approach, the approach presented in [LHR19] uses pointer-assignment graphs. However, it has been implement using the WALA [Wal19] framework and is applicable only for Java programs. Interestingly, our "incrementalized" pointer analysis achieves similar speedups for C and C++ programs.

In contrast to related works, our IncAlyzer approach is concerned with making the whole static analysis stack that is required to answer concrete client analyses incremental.

6.9 Conclusions

In this chapter, we have presented IncAlyzer, a novel approach to incremental static program analysis that is concerned with speeding up the whole analysis stack. We have shown that it can massively reduce analysis times for software projects that follow a CI/CD workflow and utilize version control systems, while providing its users with analysis results that are identical to those of a matching whole program analysis.

Since IncAlyzer's analysis results can also be persisted using the infrastructure described in Chapter 5, they can too be checked into the target project's software repository. Combining the analysis approaches described in this thesis opens up a *combined development and analysis workflow* as sketched in Section 1.3. A developer hence can not only check out the code of a specific commit and its respective static analysis summaries for the library parts of a target projects, but the analysis information for the full software project.

And while the analysis summaries for the non-library parts will be at least partially invalidated at the moment the developer starts making changes to the code, the analysis summaries for the full project can be used to initialize IncAlyzer such that it computes incremental updates from this very state without having to first conduct an initial whole-program analysis. Any local changes made to the code are then handled by IncAlyzer to update the static analysis information accordingly.

Once a new revision is added to the version control system of the project under analysis, the current, up-to-date analysis information can be persisted and checked into the version control system. By keeping code and results of static code analysis synchronized, developers can directly pick up where they left without massive delays caused by long analysis times. In such a scenario, analysis results are always available and accessible as quickly as possible.

Next, we present two projects from academia and one large industry project that underline PhASAR's actual usefulness and relevance for solving real-world problems that could hardly, if at all, be addressed before.

6.10 Incrementality: Data

This section contains the data that is too large to be directly included in the previous sections.

Table 6.4: BROTLI's target commits used as evaluation targets. C_i denotes the initial analysis run and C_{i+n} $(1 \leq n \leq 49)$ the subsequent revisions that are analyzed incrementally based on the analysis results computed in the previous run.

Program	Commit	Brief description
BROTLI		
C_i	f168456	fix typo in java variable name
C_{i+1}	5692e42	change several header and implementation files
C_{i+2}	2a51a85	update readme
C_{i+3}	63be8a9	change python file
C_{i+4}	2f9277f	update CI pipeline
C_{i+5}	bbe5d72	update java files and add branch in encoder function
C_{i+6}	f8c6717	improve decoder performance and update several java files
C_{i+7}	bdcfb12	update comment in header file
C_{i+8}	0a3944c	use VLA arrays at two places
C_{i+9}	ce222e3	update cmake
C_{i+10}	630b508	update several header and implementation files
C_{i+11}	19d86fb	merge-in larger shared directory feature
C_{i+12}	68f1b90	smaller update to few header and implementation files
C_{i+13}	0e42caf	small update to header file
C_{i+14}	a10269c	update readme
C_{i+15}	698e3a7	update readme
C_{i+16}	62662f8	huge update on how to include header files
C_{i+17}	8376f72	midsize update of several header and implementation files
C_{i+18}	27dd726	tiny update to implementation file
C_{i+19}	4ec6703	tiny update to implementation file merge
C_{i+20}	e83c7b8	update cmake
C_{i+21}	f4153a0	update setup file
C_{i+22}	c9eb856	update bootstrap file
C_{i+23}	f09b255	update bootstrap file
C_{i+24}	9801a2c	update cmake
C_{i+25}	6d03dfb	change a function's signature
C_{i+26}	388d0d5	update toml file
C_{i+27}	a8f5813	large update involving large amount of header and implementation files
C_{i+28}	ae212a7	update bootstrap file
C_{i+29}	f842c1b	update cmake
C_{i+30}	3914999	fix type in code comment
C_{i+31}	641bec0	update cmake and test shell script
C_{i+32}	9b53703	update cmake
C_{i+33}	c48ebca	update java build files
C_{i+34}	3152d99	update yml file
C_{i+35}	a7b7839	update git ignore
C_{i+36}	81dc1c8	update cmake and lua files
C_{i+37}	509d441	small update in brotli.c file
C_{i+38}	c8df4b3	update python bindings in implementation file
C_{i+39}	a2cc451	update yml workflows
C_{i+40}	81181ec	update yml
C_{i+41}	0ff6073	update documentation
C_{i+42}	ce92c95	update python file
C_{i+43}	0ea4603	tiny update to implementation file
C_{i+44}	e3ea91d	update java files
C_{i+45}	71fe6ca	tiny update implementation file
C_{i+46}	36533a8	small internal change
C_{i+47}	1e61e97	exchange function implementation in header file
C_{i+48}	04f294b	tiny update in implementation file
C_{i+49}	b2c86d1	mid-size update of several header and implementation files

Table 6.5: CURL's target commits used as evaluation targets. C_i denotes the initial analysis run and C_{i+n} $(1 \leq n \leq 49)$ the subsequent revisions that are analyzed incrementally based on the analysis results computed in the previous run.

Program	Commit	Brief description
CURL		
C_i	a1730b6	update perl file
C_{i+1}	0807fd7	small update several header and implementation files
C_{i+2}	9496d32	update yml file
C_{i+3}	5a02393	update configure-related file
C_{i+4}	1041399	update few macro definitions
C_{i+5}	4efa0b5	update pm file
C_{i+6}	6d45b9c	update several implementation files' error handling e.g., calls to helpf/notef, etc.
C_{i+7}	741f7ed	update header and implementation files re outputting errors
C_{i+8}	6661bd5	continuing on outputting errors
C_{i+9}	1f85420	small changes to header and implementation files
C_{i+10}	bfa7006	tiny update to implementation file
C_{i+11}	10d8404	update to macros in implementation file
C_{i+12}	ec70d14	update readme
C_{i+13}	d45b76e	update readme
C_{i+14}	310eb47	small update of macro definitions in header file
C_{i+15}	93df713	add branch in function's implementation
C_{i+16}	e054a16	update log statement in presence of verbose option
C_{i+17}	f4b5c88	change function's implementation
C_{i+18}	8cf4189	update unrelated test files
C_{i+19}	fff6555	tiny update to header and implementation files
C_{i+20}	6854b6c	update release notes
C_{i+21}	ba669d0	update function's implementation
C_{i+22}	c78a185	update cmake
C_{i+23}	9ad23c3	tiny update header and implementation files
C_{i+24}	0a75964	several very small update to header and implementation files
C_{i+25}	b832cab	update code comment
C_{i+26}	73022b5	add several branches to function's implementation
C_{i+27}	acc0a92	update perl script
C_{i+28}	4317c55	update perl script
C_{i+29}	78d8bc4	update unrelated test files
C_{i+30}	296baf4	update unrelated test files
C_{i+31}	51c22af	update perl file
C_{i+32}	d454af4	update perl file
C_{i+33}	f24b4b9	update perl file
C_{i+34}	6e4fede	update perl and test files
C_{i+35}	cd18e5c	update gitignore
C_{i+36}	0e339b9	update example code
C_{i+37}	e812473	update example code
C_{i+38}	3f8fc25	update large number of implementation files
C_{i+39}	c6d97bc	update unrelated test files
C_{i+40}	92d7dd3	update perl file
C_{i+41}	44296dc	update release notes
C_{i+42}	78886af	add const to several implementation files
C_{i+43}	7af151d	update runner file
C_{i+44}	7d62f0d	small update of implementation file
C_{i+45}	3c9256c	mid-size update to implementation file
C_{i+46}	c72edfa	update docs
C_{i+47}	4894ca6	update code examples
C_{i+48}	67e9e90	update code examples
C_{i+49}	dacd258	add new branch in function's implementation

Table 6.6: GZIP's target commits used as evaluation targets. C_i denotes the initial analysis run and C_{i+n} $(1 \leq n \leq 49)$ the subsequent revisions that are analyzed incrementally based on the analysis results computed in the previous run.

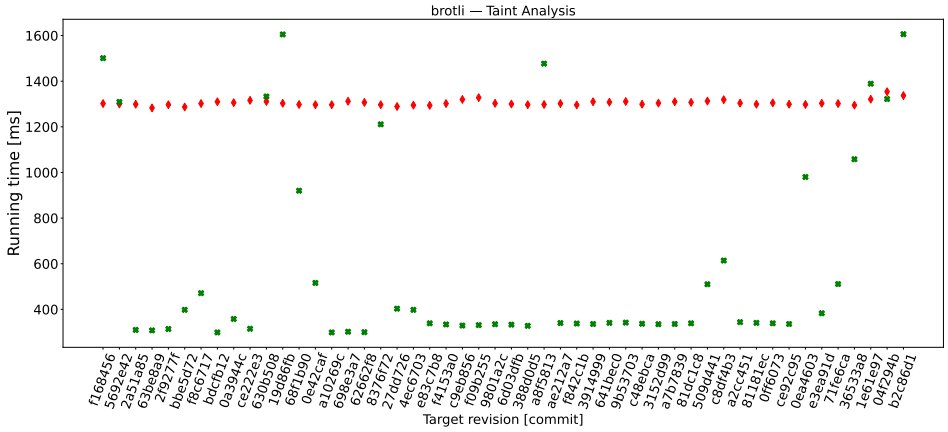
Program	Commit	Brief description
GZIP		
C_i	dc9740d	update build file
C_{i+1}	d74a30d	update build files
C_{i+2}	c99f320	update build file
C_{i+3}	6543c09	update news file
C_{i+4}	0e2d07f	update build file
C_{i+5}	5e1fc8b	update build files
C_{i+6}	9d32487	update build files
C_{i+7}	8000635	update news file
C_{i+8}	e617ae3	update post-release files
C_{i+9}	938c4f5	update build files
C_{i+10}	83c65d1	update build file
C_{i+11}	85e0910	small update in implementation files
C_{i+12}	4b58eee	tiny update in implementation files
C_{i+13}	3e32e3c	update news file
C_{i+14}	2353361	update build file
C_{i+15}	83ce5eb	bump gnulib
C_{i+16}	2801aa3	update build file
C_{i+17}	fa8fac4	bump gnulib
C_{i+18}	430edac	update gitignore
C_{i+19}	dc90550	update docs
C_{i+20}	42efd45	tiny update header and implementation files
C_{i+21}	8ae183f	tiny update in implementation file
C_{i+22}	96cd660	mid-size update to several implementation files
C_{i+23}	d8425fc	mid-size update of implementation files without effect on behavior
C_{i+24}	bcbb260	tiny update of implementation file
C_{i+25}	46ef963	update a function's signature
C_{i+26}	13dab42	update readmes
C_{i+27}	784dddc	update build files
C_{i+28}	9c3f23b	update readme
C_{i+29}	bf5f3ba	update readme
C_{i+30}	114a6bd	remove unnecessary includes
C_{i+31}	a04838d	update build files
C_{i+32}	d3ae215	bump gnulib
C_{i+33}	b9b57fe	update build files
C_{i+34}	3c01cac	update build file
C_{i+35}	f5f931a	small update of print functions
C_{i+36}	b00fcb9	update thanks file
C_{i+37}	00f4ade	update function's implementation
C_{i+38}	7553200	update build file
C_{i+39}	7e6214b	update copyright dates
C_{i+40}	f3267b5	bump gnulib
C_{i+41}	ebe613c	update copyright dates
C_{i+42}	e64e499	update build files
C_{i+43}	2640ce5	update build file
C_{i+44}	54d039e	update docs
C_{i+45}	75dac03	small update exit status broken pipe
C_{i+46}	55890dd	update build files
C_{i+47}	6990880	update unrelated files
C_{i+48}	54042d4	update unrelated files
C_{i+49}	156d0e1	replace function call

Table 6.7: HTOP's target commits used as evaluation targets. C_i denotes the initial analysis run and C_{i+n} $(1 \leq n \leq 49)$ the subsequent revisions that are analyzed incrementally based on the analysis results computed in the previous run.

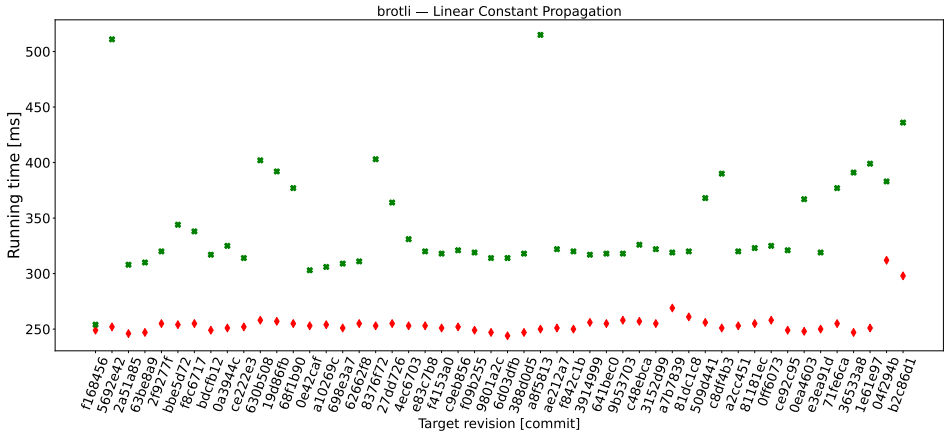
Program	Commit	Brief description
HTOP		
C_i	c66f99b	update code comment
C_{i+1}	11318b5	remove function and corresponding call
C_{i+2}	c803ec6	improve readability by adding constants
C_{i+3}	e207c8a	improve readability by adding constants
C_{i+4}	a5c4650	update macro definition
C_{i+5}	f66f04e	update code comment
C_{i+6}	b618c37	update function's implementation, complex case
C_{i+7}	ab49f3f	rename variables
C_{i+8}	61e7cb1	switch variable
C_{i+9}	c707b0e	remove branch
C_{i+10}	45b334c	changes to vector-set that is frequently used
C_{i+11}	79364ac	further updates to vector
C_{i+12}	c8a6185	update build file
C_{i+13}	e7f447b	larger update of many header and implementation files
C_{i+14}	14da84f	small update on outputting information
C_{i+15}	c878343	update code indentation
C_{i+16}	da255cb	code formatting
C_{i+17}	ccf745e	update several branches
C_{i+18}	467bb78	update variables/function calls
C_{i+19}	7a7c693	update code comment and single statement
C_{i+20}	ed7eac5	update function's implementation, complex case
C_{i+21}	1b640df	implement static assert in header file
C_{i+22}	b2ada27	use constants rather than literals
C_{i+23}	b29b33e	several mid-size updates to multiple implementation files
C_{i+24}	0c8df5f	update code comment and header include
C_{i+25}	71f5a80	several small changes
C_{i+26}	71f2e66	add code comment and branch
C_{i+27}	f50944c	add additional statement
C_{i+28}	8a8df71	fix typo
C_{i+29}	e4ebe18	change several statements
C_{i+30}	0bdade1	large update that spans multiple header and implementation files
C_{i+31}	72235d8	massive update of various header and implementation files
C_{i+32}	290ddba	small update on implementation logic, i.e., branching
C_{i+33}	ab0f68c	minor renaming of types
C_{i+34}	e40daf9	minor renaming of types
C_{i+35}	508d9ce	minor renaming
C_{i+36}	e05a203	syntax-only change
C_{i+37}	1f308b1	update variable names
C_{i+38}	148dfc0	minor variable renaming
C_{i+39}	a393066	minor change of three statements
C_{i+40}	1de7a2b	minor renaming of variables
C_{i+41}	f77ea80	update docs
C_{i+42}	3fc2862	changes in configuration file
C_{i+43}	87db379	add max iterations logic
C_{i+44}	dc883b2	minor update to retain changes
C_{i+45}	4227fbd	update keyboard shortcut
C_{i+46}	f0a7a78	merge
C_{i+47}	b810678	syntax-only change
C_{i+48}	0fb0d75	minor fix task counter
C_{i+49}	d8fe027	remove duplicate zeroing

Table 6.8: XZ's target commits used as evaluation targets. C_i denotes the initial analysis run and C_{i+n} $(1 \leq n \leq 49)$ the subsequent revisions that are analyzed incrementally based on the analysis results computed in the previous run.

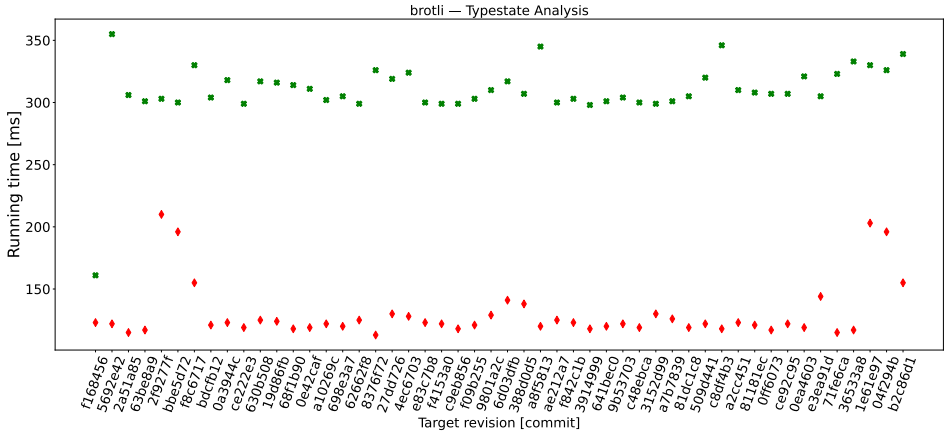
Program	Commit	Brief description
XZ		
C_i	509157c	update build files
C_{i+1}	0007394	update build files
C_{i+2}	5e57e33	update gitignore
C_{i+3}	0cc3313	update build files
C_{i+4}	75c9ca4	update cmake
C_{i+5}	133cf55	update build files
C_{i+6}	76e2315	update bash script
C_{i+7}	8b2f600	update cmake
C_{i+8}	b473a92	update documentation
C_{i+9}	5a7b930	update documentation
C_{i+10}	af4925e	update news file
C_{i+11}	f0c580c	update news file
C_{i+12}	dfe1710	update minor statement
C_{i+13}	5a5bd7f	update build configuration
C_{i+14}	3b8890a	update thanks
C_{i+15}	53cc475	update build configuration
C_{i+16}	8be136f	update build configuration
C_{i+17}	2c1a830	update cmake
C_{i+18}	b089168	update cmake
C_{i+19}	0ba234f	update cmake
C_{i+20}	116e81f	update build file
C_{i+21}	ddfe164	update cmake
C_{i+22}	cf3d1f1	update bash script
C_{i+23}	4fabdb2	update bash script
C_{i+24}	20cd905	update git workflow
C_{i+25}	4d7fac0	update cmake
C_{i+26}	2cb6028	update cmake
C_{i+27}	8be5cc3	update bash script
C_{i+28}	d0faa85	update bash script
C_{i+29}	6549df8	update readme
C_{i+30}	537c6cd	update readme
C_{i+31}	fb9c50f	update thanks
C_{i+32}	0fbb2b8	update security
C_{i+33}	2a89670	remove unused function
C_{i+34}	3938718	update code comment
C_{i+35}	7062348	update preprocessor ifdefs
C_{i+36}	f41df2a	update header includes
C_{i+37}	78ccd93	update thanks
C_{i+38}	16b81a0	update bash script
C_{i+39}	2cf5ae5	update git workflow
C_{i+40}	44c0c5e	update preprocessor ifdefs
C_{i+41}	6be460d	update preprocessor ifdefs
C_{i+42}	9ad64bd	update preprocessor ifdefs
C_{i+43}	713e15e	update build file
C_{i+44}	77050b7	update news file
C_{i+45}	c247d06	update news file
C_{i+46}	d0f33d6	replace test construct by macro
C_{i+47}	8f23657	smaller changes to header and implementation files (branches)
C_{i+48}	3374a53	add lzmd-nothrow annotation
C_{i+49}	f36ca79	update code comment



(a) BROTLI — Taint analysis

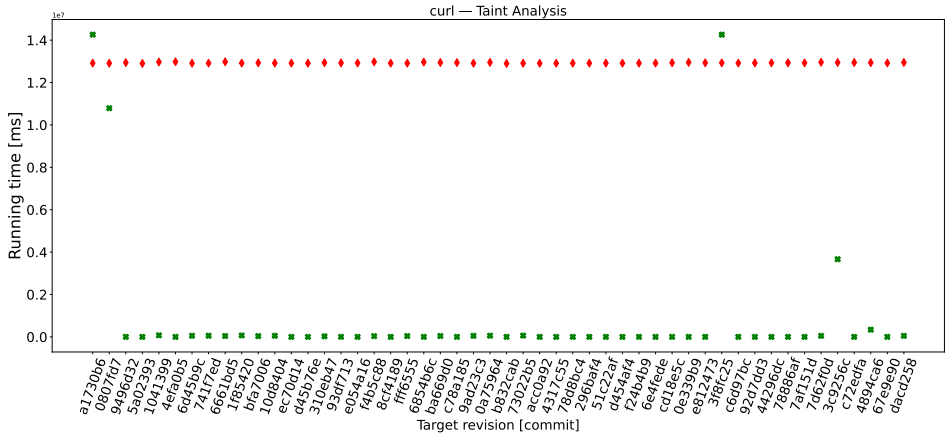


(b) BROTLI — LCA

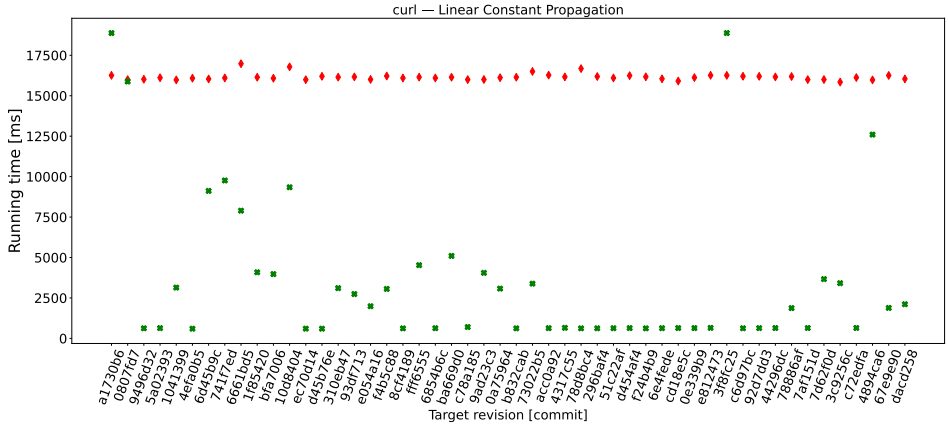


(c) BROTLI — Typestate analysis

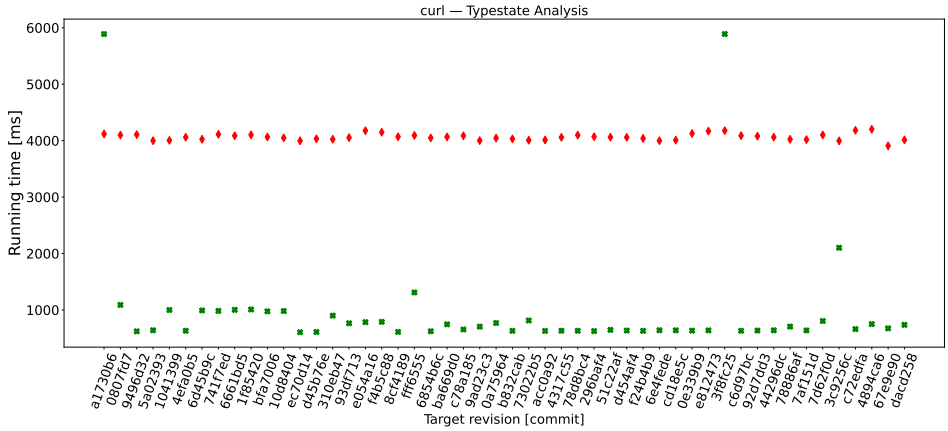
Figure 6.2: Running times for BROTLI. Running times for WPA analysis runs are indicated by a red diamond and running times for IncAlyzer runs are indicated by a green square.



(a) CURL — Taint analysis

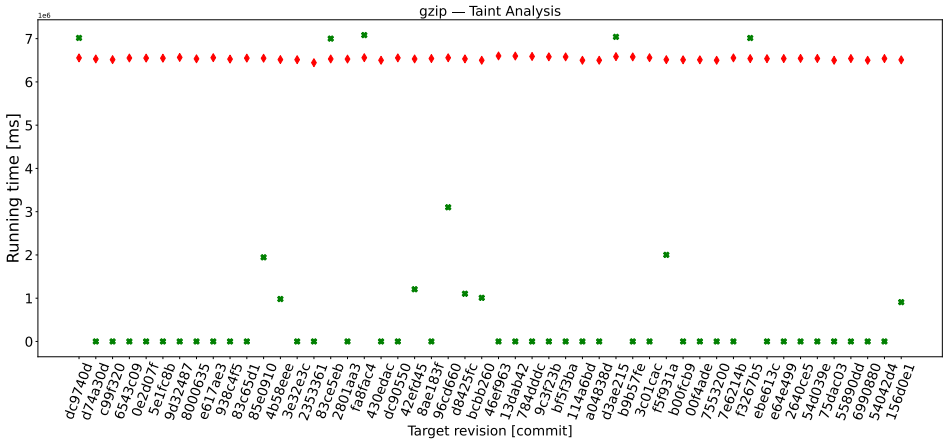


(b) CURL — LCA

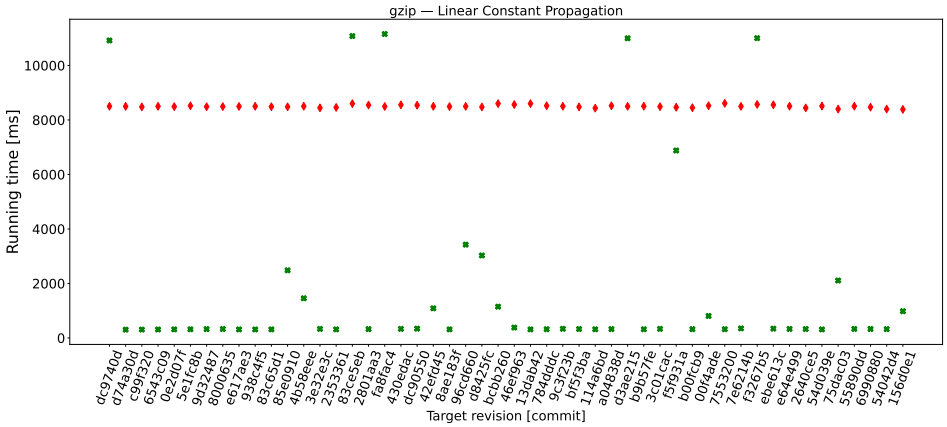


(c) CURL — Typestate analysis

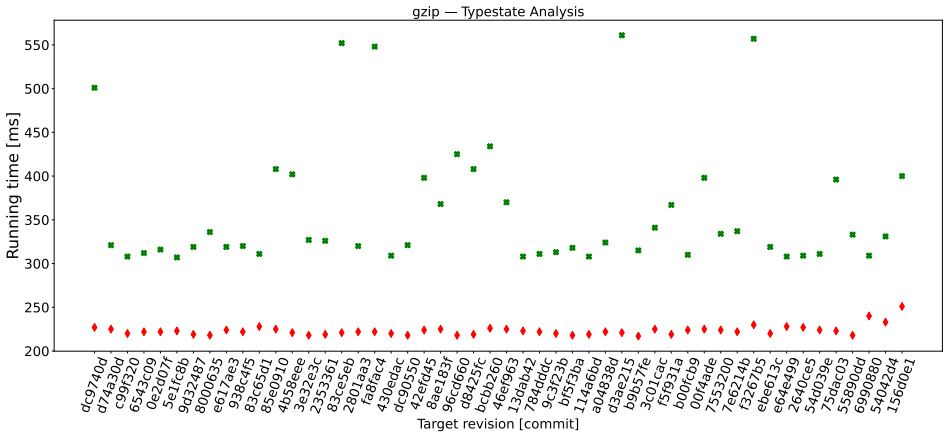
Figure 6.3: Running times for CURL. Running times for WPA analysis runs are indicated by a red diamond and running times for IncAlyzer runs are indicated by a green square.



(a) GZIP — Taint analysis

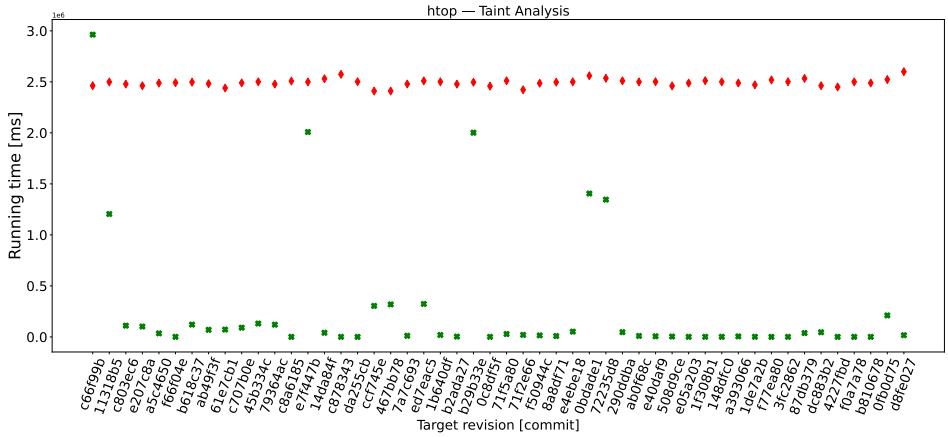


(b) GZIP — LCA

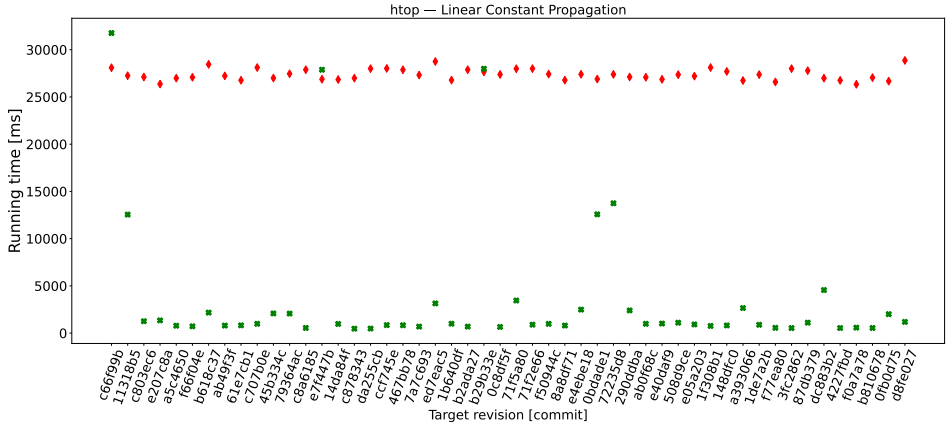


(c) GZIP — Typestate analysis

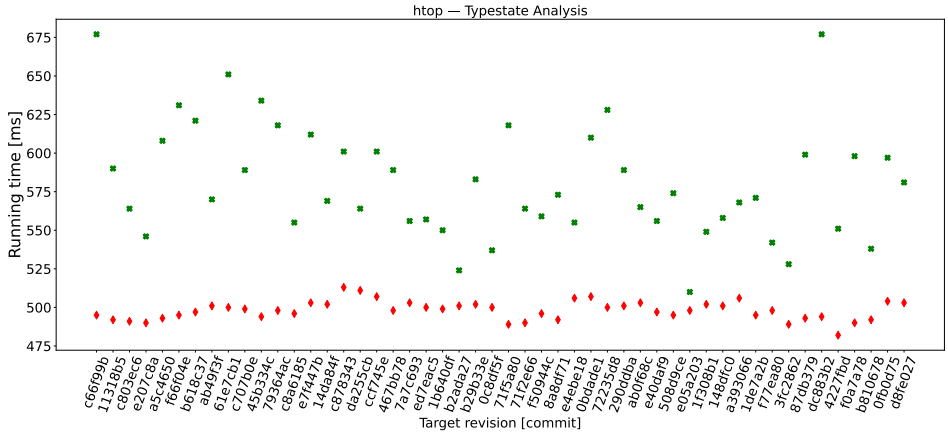
Figure 6.4: Running times for GZIP. Running times for WPA analysis runs are indicated by a red diamond and running times for IncAlyzer runs are indicated by a green square.



(a) HTOP — Taint analysis

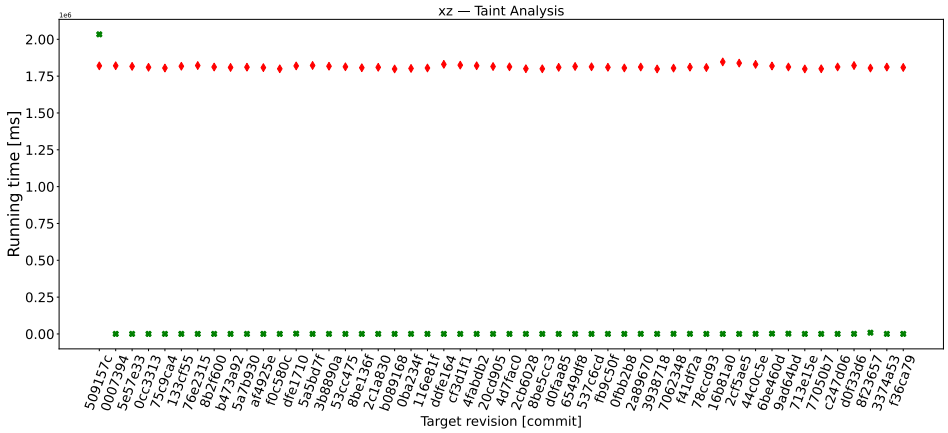


(b) HTOP — LCA

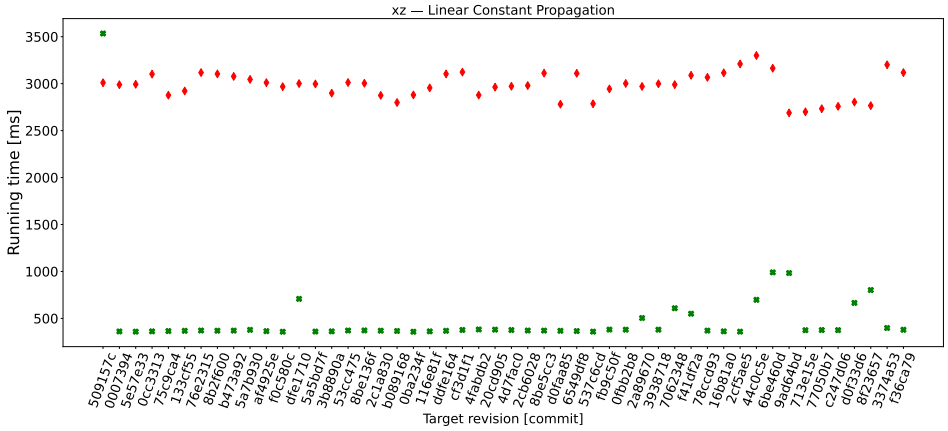


(c) HTOP — Typestate analysis

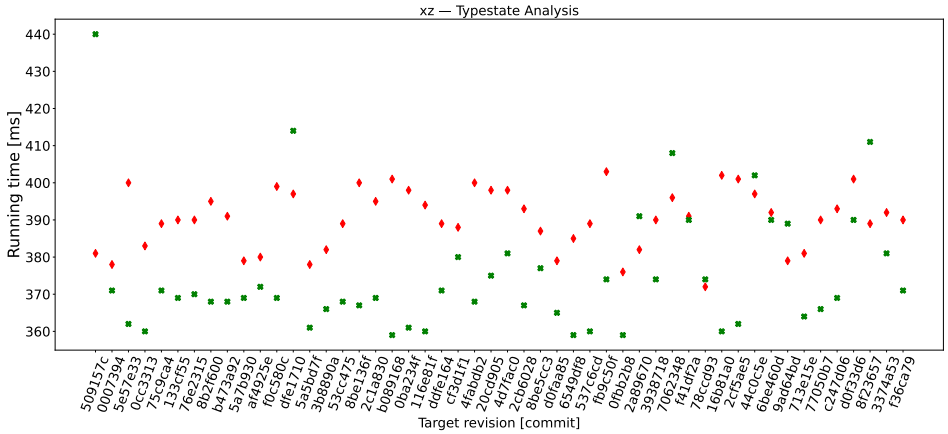
Figure 6.5: Running times for HTOP. Running times for WPA analysis runs are indicated by a red diamond and running times for IncAlyzer runs are indicated by a green square.



(a) XZ — Taint analysis



(b) XZ — LCA



(c) XZ — Typestate analysis

Figure 6.6: Running times for XZ. Running times for WPA analysis runs are indicated by a red diamond and running times for IncAlyzer runs are indicated by a green square.

7 Applications of PhASAR

This chapter presents two projects from academia and one large industry project that we conducted over a time frame of 15 months with one of the world's largest telecommunications companies. This underlines that PhASAR indeed helps opening up new and interesting fields of research and allows for solving actual, real-world problems that could hardly be addressed before.

7.1 Combining Repository Mining and Static Code Analysis

The theory of socio-technical systems (STS) has been developed by Eric Trist, Ken Bamforth, and Fred Emery [Tri81]. It is an approach to complex organizational work design that recognizes the interaction between humans and technology. *Socio-technical* refers to the interrelatedness of social and technical aspects of an organization or a (software) system. The development of a deep understanding of a technical system—with a particular focus on how people interact with it—is one of the theory's main aspects to provide guidance on joint optimization.

Sattler et al. integrate program analysis techniques with the socio-technical inner workings of software by combining high-level information on software projects obtained by repository mining with precise, low-level static data-flow information [SBS+23].

The approach obtains high-level socio-technical information for a given target project using relatively lightweight repository mining techniques. This socio-technical information is then handed over to and is processed by a modified compiler front-end for the programming language that the target software is written in. The customized compiler front-end attaches the information to the compiler's intermediate representation during IR (code) generation. The result is a version of the target project in the compiler's IR that is amended with additional socio-technical information. This enriched IR is then placed at the disposal for further heavyweight, semantic program analysis techniques. A downstream analysis tool can analyze the IR and access the attached socio-technical (meta) data, allowing it to produce attributed results that can provide novel insights.

The approach is formally defined as a conceptual framework called VaRA [SBS+23, Sat23]. Its concrete implementation has been built on top of LLVM and PhASAR. The source of socio-technical information as well as the source of the program analysis information are fully parameterizable and can be specified by VaRA's users. A concrete instance of the VaRA framework that shows its potential is presented in what follows.

The author of this thesis contributed to this piece of research a special-purpose data-flow analysis encoded within the *Interprocedural Distributive Environments* (IDE) [SRH96] framework. The analysis has been implemented in PhASAR and allows one to exhaustively

compute data flows for all program variables of the project under analysis. The data-flow analysis aims at finding interactions between program instructions. Whenever an instruction modifies a piece of memory, i.e., a variable, it becomes associated with this memory. If two or more instructions operate on the same piece of memory, they are considered to interact with each other. When parameterizing VaRA to use information on commits obtained by *git blame* as the source of socio-technical information, each LLVM IR instruction is annotated (by the customized compiler front-end) with the commit hash of the source code line from which it originates. The *git blame* command shows author information of each line of a project's source file that last modified it. This information includes author name, e-mail, commit hash, time stamp, etc. Making this information available to the special-purpose data-flow analysis allows one to automatically determine which of the target project's commits interact with each other *on the data-flow level*. If a taint analysis is chosen as the downstream static analysis, for instance, it can report potential (security) vulnerabilities found within the target project. Using information on the commit interactions computed by the exhaustive data-flow analysis allows one to attribute the taint analysis' findings to a specific project version, author(s), or development team. This opens up the opportunity to determine the authors of the program paths along which the undesired taint flow has been propagated, for instance. In practice, static analysis—especially if conducted in a whole-program manner—might produce lots of potential findings, many of which may be false positives [BBC+10, HO18] such that it has become a real challenge to prioritize and check them. Being able to access commit information at a fine-grained data-flow level enables one to report the analysis' findings directly to the respective authors, who are most familiar with the code for which an issues has been found.

Being able to compute these kinds of information also opens up new opportunities for automating code reviews, for instance. Bugs and security vulnerabilities can be directly and automatically reported to the author who introduced them accidentally (or on purpose [mei22]).

The VaRA approach allows one to obtain novel insights on software projects that were previously locked away and has the great potential to change how we think about software projects and code as it opens up a large variety of useful applications. For more details and a variety of further interesting example parametrizations and applications of the VaRA framework, we refer to reader to [SBS+23].

7.2 Static Configuration-Logic Identification

In another piece of research that is currently under submission [ASR+23], Alhanahnah et al. shows that programs can oftentimes be divided into two parts: a part that configures the program according to the user's parametrization and another part that comprises the program's main computations. Such a program point that separates those two parts not only exists for command-line applications, but for configuration-file programs and server applications, too.

Alhanahnah et al. denotes the program point that separates a program into configuration logic and actual computation as Boundary. A scaled down version of the UNIX `wc`

utility program is shown in Listing 7.1. The program shows that `wc` comprises a configuration part that parses the configuration information provided by the user—here using the command-line arguments—via **main**'s **argv** variable. The program transforms these external configuration information into an internal format by assigning initial (or new) values to certain program-internal configuration variables. These configuration variables are then used to determine which parts of the actual program computations execute.

```
1  #include <stdio.h>
2
3  int count_chars = 0;
4  int count_lines = 0;
5
6  int total_chars = 0;
7  int total_lines = 0;
8
9  int main(int argc, char **argv) {
10   for (int i = 1; i < argc; ++i) {
11     if (argv[i][0] == '-') {
12       switch (argv[0][1]) {
13       case 'c':
14         count_chars = 1;
15         break;
16       case 'l':
17         count_lines = 1;
18         break;
19       default:
20         printf("Invalid flag %s", argv[i]);
21       }
22     }
23   }
24   // ---------- Boundary ----------
25   char buffer[1024];
26   while (fgets(buffer, 1024, stdin)) {
27     if (count_chars) {
28       total_chars += sizeof(buffer);
29     }
30     if (count_lines) {
31       total_lines++;
32     }
33   }
34   if (count_chars) {
35     printf("Total chars = %d\n", total_chars);
36   }
37   if (count_lines) {
38     printf("Total lines = %d\n", total_lines);
39   }
40   return 0;
41 }
```

Listing 7.1: A scaled down version of the `wc` utility. The boundary between configuration logic and main logic has been depicted using `// --- Boundary ---`.

The analysis of program configurations enables one to understand under which circumstances execution may fail or to apply partial evaluation with the goal of speeding up execution or decreasing a program's size. Alhanahnah et al. shows that information on a program's Boundary enables one to solve various software engineering problems. Besides providing useful information for program comprehension to developers, it enables one to conduct software debloating in a fully automated manner. Software debloating aims at automatically removing parts of a program that are not actually required to correctly execute in a certain—user specified—configuration. The scaled down `wc` program in Listing 7.1, for instance, allows one to compute the number of characters and the number of lines of the input provided using **stdin**. If a user, however, is only interested in counting the lines of the input, `wc` can be debloated with respect to the `-l` option to produce a specialized version of the program shown in Listing 7.2. Program debloating provides several advantages as it reduces a program's size and therefore oftentimes also its attack surface, and may leads to improved performance as the compiler's optimizer can perform more aggressive optimizations.

```
1  #include <stdio.h>
2
3  int total_lines = 0;
4
5  int main(int argc, char **argv) {
6    char buffer[1024];
7    while (fgets(buffer, 1024, stdin)) {
8      total_lines++;
9    }
10   printf("Total lines = %d\n", total_lines);
11   return 0;
12 }
```

Listing 7.2: A specialized version of the program shown in Listing 7.1 that only counts the lines of a given input.

Previously, a program's Boundary location had to be identified manually to enable software debloating. This manually identified Boundary can then be used to automatically debloat a program using a tool like LMCAS [AJR+22]. LMCAS performs partial evaluation [JGS93] with respect to the values of the configuration-hosting variables, which are correctly set up and initialized at a program's Boundary location. The approach abstractly operates as follows (reproduced from [ASR+23]):

$$P(x, y) = \text{let } t = \mathit{translate}(x) \text{ in } g(t, y) \tag{7.1}$$

$$\rightarrow P_x(y) = g_t(y) \tag{7.2}$$

P is a two-argument program, $P(x, y)$. $P_x(y)$ is a version of $P(x, y)$ that is specialized on the specific value of x. The body of $P_x(y)$ is obtained by finding and evaluating $t = \mathit{translate}(x)$, and then running a partial evaluator on g with static input t to create $g_t(y)$, which is a version of $g(t, y)$ specialized on the value of t.

A manual field study conducted by Alhanahnah et al. identified the following relevant properties to identify a program's BOUNDARY:

1. Configuration-hosting variables are data dependent or control dependent on **argv**.
2. The BOUNDARY should be located after at least one loop.
3. The BOUNDARY represents an articulation point in the program's control-flow graph.
4. The BOUNDARY should be reachable from the entry point and execute only once.

More formally, this problem can be defined as finding an articulation point (basic block) B of the target program's ICFG G that is reachable from v_{entry}, and is (i) located after a loop, (ii) post-dominates every assignment to a member of C_{host}, and (iii) for each $c \in C_{host}$, all paths from B to v_{exit} are free of definitions to c. If multiple program locations satisfy these conditions, the program location B that is closest—in terms of control-flow edges—to the entry is identified as the BOUNDARY. C_{host} is the set of configuration-hosting variables. v_{entry} is the ICFG node that represents the target program's entry point and v_{exit} is the ICFG node that represents the program's exit point.

The author of this thesis helped identify these properties, and formalized and implemented an algorithm to automatically find a program location that satisfies these properties. The automated BOUNDARY identification algorithm has been implemented in C++ in a tool called SLASH using PhASAR and LLVM. SLASH can automatically identify the BOUNDARY for up to 96% of the 23 target programs used for its evaluation and is able to analyze all of them in 8.5 minutes and with a maximum memory consumption of 4.4 GB.

SLASH can be integrated with the state-of-the-art debloating tool LMCAS [AJR+22] and eliminates the requirement for user-annotated BOUNDARY locations without breaking its functionality. This for the first time allows for a fully automated debloating pipeline. For further details we refer the reader to [ASR+23].

7.3 White-Box Penetration Testing

We next detail how we designed and implemented a PhASAR-based static analyzer to find bugs and (security) vulnerabilities in low-level C and C++ code that runs on routers and embedded systems. This industry project has been conducted with one of the world's largest telecommunications companies. We first present our approach and solution to find bugs and vulnerabilities in low-level C and C++ using white-box penetration testing. Then, we present the insights that we gained in this 15 month project and the challenges that we faced during the project. In the following, we refer to the PhASAR-based static analyzer as *static analysis engine* (SAE) and to the industry partner as *AnonymousCompany*.

The overarching goal of this white-box penetration test project was to combine the strengths of static *and* dynamic program analysis while eliminating their weaknesses (to the extend possible). On the one hand, static program analysis inspects all paths of a program, but frequently suffers from an unduly high false positive rate. This can lower the acceptance rate for developers to use respective tooling to the point at which they reject

it [BBC+10]. To account for a potentially high amount of false positive findings reported by static analysis, fuzzing can be used to generate test cases that aim at dynamically confirming (or disconfirming) the static analysis findings. This automated validation of static analysis findings reduces the workload put onto developers. On the other hand, the findings computed using static analysis provide valuable information to guide a fuzzer towards potentially interesting program locations. Since a fuzzer on its own would otherwise generate random test cases, a combined hybrid approach can make software testing much more efficient. In this project, we developed a PhASAR-based static analyzer that is intended to be used with a fuzzing engine developed by AnonymousCompany.

Techniques for white-box penetration testing include manual code reviews, automated code reviews, and fuzz testing. Manual code reviews, however, do not scale and are tedious and expensive when being applied to large-scale software projects. Hence, they are oftentimes only applied to particularly interesting and especially critical parts of the target software. Automated code reviews counteract these scalability issues by employing automated static analysis tools that aim at detecting suspicious patterns in the source code. These findings are then manually inspected and confirmed or disconfirmed by an expert.

Fuzz testing (or fuzzing) is a technique to automatically detect undesired behaviors and to find potential (security) vulnerabilities in software. Fuzz testing works by generating a plethora of test cases that run the software under analysis on random inputs to provoke interesting program behaviors. In case of white-box penetration testing, the complete source code is known and available for analysis. Hence, additional static program analysis techniques may be used to improve fuzz testing by augmenting the fuzzing algorithm with information computed by static analysis. This improves the fuzzer's effectiveness and makes it less random by targeting potentially interesting program locations identified by the static analysis. The fuzzer, on the other hand, also counteracts one of the great weaknesses of static analysis: false positives. Instead of having the findings computed by a static analysis manually checked by an expert, the fuzzer allows for automated confirmation of findings, which helps developers to endure the noise generated by false positives.

To determine interesting test inputs for the fuzzer, the static taint analysis first computes program slices for program variables and program locations of interest. Whenever the taint analysis detects an undesired data flow for a given variable, the respective program slices that may cause this undesired data flow are analyzed in detail using a subsequent symbolic execution. Analyzing the program slices using a symbolic execution aims at detecting concrete vulnerabilities and yields path condition(s) that must be satisfied to cause these undesired data flow(s) at runtime. Instead of employing symbolic execution only, we use a relatively lightweight upstream static data-flow analysis that records paths to counteract the well-known problem of path explosion. The symbolic execution only gets to analyze the paths reported by the upstream data-flow analysis. The path conditions computed by the symbolic execution are eventually provided to the downstream fuzzer to automatically generate test cases that confirm or disconfirm the undesired data flow(s) and to show their absence once the bug or vulnerability has been fixed.

SAE's implementation is based on PhASAR [SHB19], LLVM [LA04] and Z3 [dMB08], and comprises, besides various (static) helper analyses, two main static analysis components: a precise, parameterizable taint analysis and an extensible symbolic execution. We

built our implementation on PhASAR since it is the only LLVM-based infrastructure that allows the detailed static analysis of programs written in C and C++, and provides all means to develop and solve sophisticated state-of-the-art static data-flow analyses.

Taint analysis, the first main component of SAE, is a very prominent client data-flow analysis that can be used to solve a multitude of analysis problems, depending on the concrete parametrization of sources and sinks. It can be used to find all kinds of code injection vulnerabilities such as SQL or XSS (cross site scripting) injections, for instance, but it can also be used to find memory leaks. In this project, a path-sensitive taint analysis is used to find undesired data flows of sensitive program variables. It produces as a result program slices that are interesting for further analysis by a downstream symbolic execution.

Symbolic execution, the second major component of the analysis engine, is a static analysis technique that reasons about a program path by path. As the number of program paths grows exponentially with every conditional, symbolic execution oftentimes becomes prohibitively expensive for real-world applications. This is also the reason why dedicated tools that implement symbolic executions techniques, such as the Clang Static Analyzer [Cla18b], analyze code inter-procedurally but not across translation units. In this project, we aim at statically analyzing the whole program, which is one of the reasons why an additional upstream taint analysis is necessary to compute program slices in order to reduce the symbolic execution's analysis space. Whenever a runtime value, i.e., a value whose assignment is unknown at compile time, is being processed, symbolic execution assigns a symbolic value. These symbolic values are propagated through the program and modifications made to them as well as comparisons (in conditional branches) are expressed as (path) constraints using logic formulas. The formulas can be simplified and in some cases even (re)solved to constant values using *Satisfiability Modulo Theories* SMT solvers such as Z3 [dMB08]. This allows one to reason about the exact path conditions that must be satisfied to reach a certain point in the target program at runtime.

We also employ symbolic execution to implement custom bug checkers that reason about (i) potential out-of-bounds buffer accesses and (ii) the finiteness of loops to the extend that is theoretically decidable and practically feasible. The result of these symbolic checks are findings of potential programming mistakes and potential vulnerabilities along with their respective data-flow paths and associated path constraints along which they may be triggered at runtime. The (simplified) path constraints for the data flows that lead to potential vulnerabilities issued by the symbolic checks can be subjected to a downstream fuzzer. This allows a fuzzer to create target-oriented test cases. SAE's symbolic checks are extensible: additional checks that aim at finding different kinds of bugs or vulnerabilities can be added to and integrated with SAE.

7.3.1 Running Example

In the following, we present a modified and truncated version of one of the C test programs provided by AnonymousCompany as a running example. The program shown in Listing 7.3 and Listing 7.4, however, still realistically demonstrates the type of bugs and vulnerabilities that AnonymousCompany faces when developing low-level C and C++ code. We use this example to detail on the design, implementation and possible usages of SAE.

Listing 7.3 introduces various macro definitions as well as definitions of the data types **MailBlock**, **MailEnvelope**, and **SubMail** and function declarations that are used in Listing 7.4. The code shown in Listing 7.4 implements different handlers that process instances of these types. Some of these handlers comprise vulnerabilities. An overview of the handlers and their respective vulnerabilities is shown in Table 7.1. The goal of SAE is to automatically detect these vulnerabilities and issue the respective path constraints.

Table 7.1: Overview of the various handlers implemented in Listing 7.4 and their respective vulnerability, if any.

Handler	Vulnerabilities	Vulnerability Location
handler_1	out-of-bound	Line 12
handler_2	out-of-bound	Line 23
handler_3	out-of-bound	Line 36
handler_4	none	
handler_5	infinite loop	Line 59

```
1  #ifndef COMMON_H
2  #define COMMON_H
3  #define SUB_MAIL_ENV_SIZE 255
4  #define INDEX_DIVIDER 4
5  #define MAX_CONTENT_SIZE 1024
6  #define SUB_LOOP_OFFSET 20
7  typedef struct MailEnvelope {
8    unsigned int mailType;
9    unsigned int sendId;
10   unsigned int recId;
11 [[clang::annotate("psr", "SizeGuard", "MailBlock::Contents")]]
12   unsigned int contentSize;
13 } MailEnvelope;
14
15 typedef struct MailBlock {
16   MailEnvelope envelope;
17   unsigned int mailBlockSize;
18 [[clang::annotate("psr", "SizeGuard", "MailBlock::Contents")]]
19   unsigned char content[];
20 } MailBlock;
21
22 typedef struct SubMail {
23 [[clang::annotate("psr", "SizeGuard", "SubMail::SubMailEnvelope")]]
24   unsigned char subMailEnvelope[SUB_MAIL_ENV_SIZE];
25   unsigned int subContentSize;
26   unsigned char subContent[];
27 } SubMail;
28
29 typedef void (*MailHdlFuncPtr)(MailBlock *);
30 #endif // COMMON_H
```

Listing 7.3: Header file defining the data types required by Listing 7.4.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "common.h"
4
5  void handler_1(MailBlock *mail_p) {
6    int totalSize = mail_p->envelope.contentSize;
7    unsigned char *content_p = mail_p->content;
8    SubMail *subMail_p = (SubMail *)content_p;
9    while ((totalSize > 0) && (subMail_p != NULL)) {
10     totalSize = totalSize - sizeof(SubMail) - (subMail_p->
           subContentSize);
11     // VUL: no check of totalSize before accessing next message body
12     content_p += sizeof(SubMail) + subMail_p->subContentSize;
13     subMail_p = (SubMail *)content_p;
14   }
15   return;
16 }
17
18 void handler_2(MailBlock *mail_p) {
19   unsigned int i;
20   for (i = 0; i < mail_p->envelope.contentSize; i++) {
21     // VUL: no check of array index before accessing
22     unsigned int index = (i + mail_p->envelope.mailType) /
           INDEX_DIVIDER;
23     unsigned char value = mail_p->content[index];
24   }
25   return;
26 }
27
28 void handler_3(MailBlock *mail_p) {
29   unsigned int i;
30   for (i = 0; i < mail_p->envelope.contentSize; i++) {
31     unsigned int index = (i + mail_p->envelope.mailType) /
           INDEX_DIVIDER;
32     // VUL: wrong pretection
33     if (index > MAX_CONTENT_SIZE) {
34       break;
35     }
36     unsigned char value = mail_p->content[index];
37     // do something
38   }
39   return;
40 }
41
42 void handler_4(MailBlock *mail_p) {
43   unsigned int i;
44   for (i = 0; i < mail_p->envelope.contentSize; i++) {
45     unsigned int index = (i + mail_p->envelope.mailType) /
           INDEX_DIVIDER;
46     // Correct protection
47     if (index >= mail_p->envelope.contentSize) {
48       break;
49     }
```

```
50      unsigned char value = mail_p->content[index];
51    }
52    return;
53  }
54
55  void handler_5(MailBlock *mail_p) {
56    SubMail *subMail_p = (SubMail *)mail_p->content;
57    unsigned char *subContent_p = subMail_p->subContent;
58    unsigned int len = 0;
59    while (len < subMail_p->subContentSize) {
60      unsigned char subLoopSize = *(subContent_p + SUB_LOOP_OFFSET);
61      // VUL: could be infinite loop if subLoopSize is zero
62      len += subLoopSize;
63      subContent_p += subLoopSize;
64    }
65    return;
66  }
67
68  // Find message handler
69  MailHdlFuncPtr findMailHdl(MailBlock *mail_p) {
70    if (mail_p == NULL) {
71      return NULL;
72    }
73    // Complex logic to retrieve and return the suitable handler
74    return mailHdlTable[i];
75  }
76
77  // Post incoming message to its handler
78  void postMail(MailBlock *mail_p) {
79    MailHdlFuncPtr myMailHdl_p = findMailHdl(mail_p);
80    if (myMailHdl_p != NULL) {
81      (*myMailHdl_p)(mail_p);
82    }
83  }
84
85  // Test message handler callbacks
86  void test(MailBlock *mail_p) {
87    // Further complex logic
88    postMail(mail_p);
89  }
90
91  int main(int argc, [[clang::annotate("psr.source")]] char **argv) {
92    printf("Initializing...\n");
93    // Detailed setup of function pointer tables, initialization, etc.
94    [[clang::annotate("psr.source")]] MailBlock *mail_p =
95      (MailBlock *)(argv[1]);
96    mail_p->envelope.mailType = strtol(argv[1], NULL, 10);
97    test(mail_p);
98    return 0;
99  }
```

Listing 7.4: One of the target program provided by AnonymousCompany that implements various (erroneous) handlers.

7.3.2 Overview of the Static Analysis Engine

In this section, we give an overview of SAE's workflow and its individual components. A schematic of the analysis engine is shown in Figure 7.1.

In the following, we refer to analysis concepts that we implemented (as types, i.e., C++ classes) within SAE by typesetting them using **this source code highlighting**.

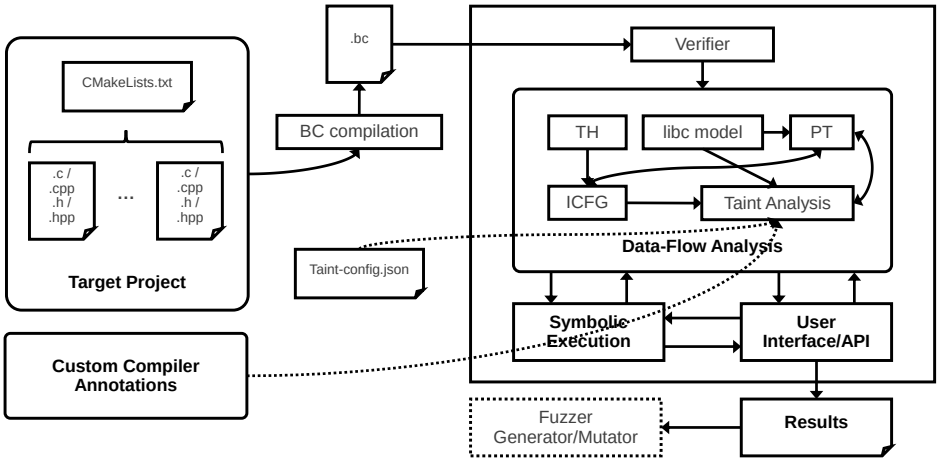


Figure 7.1: Overview of the Static Analysis Engine. *TH* denotes type hierarchy information, *PT* denotes points-to information and *ICFG* denotes information on inter-procedural control flows (which includes callgraph information).

Pre-Processing

SAE has been built on top of PhASAR and therefore analyzes LLVM's intermediate representation. To produce LLVM IR (sometimes called LLVM bitcode) for real-world projects, one needs to extract the exact compile *and* link commands encoded in the build system that is used by the target project. Doing this manually is an infeasible task due to the multitude of different build systems such as Makefile, CMake, Bazel, etc. As explained in Section 3.8.3, a solution to produce whole-program LLVM IR is to use compiler wrappers such as WLLVM [WLL21] and GLLVM [GLL21].

Since the various analyses that are part of SAE all operate on LLVM IR, their raw results are expressed on LLVM IR level, too. However, based on the analysis results we eventually wish to make statements on the target project under analysis that are useful and clear to developers. Therefore, SAE needs to ensure that the results on LLVM IR level can be mapped back to source code level to provide meaningful insights to its users. To ensure that a re-mapping of the results is possible, SAE requires the source code to be compiled using the `-g` compiler flag to enable debugging information. By enabling debug information, the compiler preserves information on original variable names, line numbers, columns, etc., which are required for mapping the results back to source code level. PhASAR provides

APIs to access this information and to map IR constructs to their respective counterparts on the source code level.

Taint Analysis

In this section, we are giving an overview of SAE's taint analysis. For SAE's taint analysis we assume the dependencies and helper analyses as shown in Section 2.3.

We need to base SAE's taint analysis on a highly precise, inter-procedural control-flow graph. This is because the target program (cf. Listing 7.4) makes use of indirect function calls in form of calls to function pointers. Other test programs provided by AnonymousCompany are written in C++ and contain (indirect) calls to virtual function members. To compute precise ICFGs, we require additional points-to information to resolve function pointers and allocated types of receiver objects upon which virtual functions are being called. Fortunately, the indirect function calls in the test programs provided by AnonymousCompany can be successfully resolved using the points-to information provided by PhASAR and LLVM.

Since the target programs are written in low-level C and C++, an analysis has to deal with lots of calls to the C runtime. When analyzing LLVM IR, calls to the C runtime, i.e., libc functions, such as **malloc**, **free** or **printf** are left unresolved. These functions are only available as declarations within the respective IR. This is because their implementations are usually not available, and even if they would be, these functions are typically implemented in low-level C or assembly making them particularly hard to analyze. LLVM models many of libc's functions that are particularly low-level such as **memcpy** or **memmove** as so-called *intrinsic* functions. LLVM's intrinsic functions only describe the semantics of a function albeit there is no actual implementation for that function. LLVM's code generator decides at code generation time how to implement calls to intrinsic functions for the given target architecture. It may even use a hardware implementation to correctly handle the semantics, if available. In this project, we use PhASAR's extensible mechanism described in Section 3.4.5 that precisely models, i.e., summarizes the semantics of the libc and LLVM intrinsic functions to be able to capture the target program's behavior. This summary model directly affects the points-to and data-flow analysis and indirectly affects all other pieces of static analysis information.

In this project, we designed and implemented **IDEExtendedTaintAnalysis** to compute program slices for variables of interest such as array typed variables that may not overflow. Whenever the analysis detects an interesting (or illegal) data flow, the paths that lead from the source to the very sink at which the leak is detected are extracted for further, more detailed analysis using symbolic execution.

Since the target software is implemented in C and C++, it represents a particularly hard-to-analyze target and we have to take great care to avoid what is known as *overtainting* [SAB10], which prevents the analysis to compute any useful results. A taint analysis that is based on imprecise points-to information, for instance, may generate a large amount of tainted variables when being set up to overapproximate information. This often results in virtually all program variables being marked as tainted, which does not provide any helpful information. To compute precise data-flow (or taint-flow) paths, the analysis has to be

flow-, context-, field- *and* path-sensitive. Generally, precise, inter-procedural, and context-sensitive data-flow analyses are notoriously hard to scale for large applications. However, since taint analysis can be expressed using distributive flow functions [Bod18], it can be encoded within distributive data-flow frameworks, e.g., IFDS [RHS95], IDE [SRH96] or W(PDS) [RSJ03]. We thus implemented **IDEExtendedTaintAnalysis** as an IDE [SRH96]-based analysis that obtains its flow- and context-sensitivity due to IDE [SRH96] and models (k-limiting) field-sensitivity explicitly.

Taint analysis requires a configuration that specifies sources, sinks, and sanitizers. Our taint analysis is configurable in two ways as described in the following: (i) a configuration file in JSON format allows one to specify functions and their parameters and/or return values as sources, sinks or sanitizers. In addition, we support (ii) custom compiler annotations that can be used to specify sources, sinks, and sanitizers directly in the target application code. Custom compiler annotations are introduced with **[[clang::annotate()]]** as shown in Listing 7.3 and Listing 7.4. The compiler annotations made to the source code are automatically preserved and made available for analysis in LLVM's IR with help of LLVM's Clang compiler. Custom compiler annotations present an additional source of information for code analysis, but other than that have no effect on a program's semantics. In particular, they do not affect the target program's performance. The support for the compiler annotations is implemented in **Annotation**. SAE's taint analysis recognizes these compiler annotations while the software can still be compiled in production without modifying the behavior of the resulting executable. The taint configuration is managed by the **TaintConfig** type.

The interesting data flows computed by the taint analysis are subjected to a subsequent symbolic execution to retrieve information on potential vulnerabilities and their respective path constraints. The program slices are computed using **PathSensitivityManager**.

Symbolic Execution

In the following, we give an overview of the symbolic execution that we use to determine detailed information on the path constraints that lead to the data leaks computed by the taint analysis.

Symbolic execution assigns symbolic values to the input variables and variables whose values cannot be determined statically at compile time, and computes information on how they are being manipulated and used along the various execution paths of the program under analysis. This, however, is exactly why symbolic execution does not scale well: since it reasons about the program path by path, and the number of paths in a program grows exponentially in the number of conditional branches. We counteract this scalability issue by analyzing only the relevant paths that comprise the interesting program slices that have been overapproximated by the comparably more lightweight taint analysis. To compute constraints to be fed into AnonymousCompany's downstream fuzzer, we are mainly interested in the path constraints that the symbolic execution yields. These constraints are logic formulas describing the conditions that must be satisfied in order to reach a certain point in the program. The accumulated path constraints can be solved and simplified using *satisfiability* (SAT) solvers and *satisfiability modulo theories* (SMT) solvers. We use the

SMT solver implementation Z3 [dMB08] to determine concrete input values that trigger the execution path(s) that cause the data-flow paths of interest determined by the taint analysis. SAE implements a symbolic execution engine with help of Z3 [dMB08]. The generation and handling of path constraints are implemented in **LLVMPathConstraints**.

Besides being able to compute the path constraints for the data-flow paths issued by the taint analysis, SAE's symbolic analysis also provides a mechanism to implement custom symbolic checks. In this project, we implemented checks that allow one to detect potentially infinite loops and potential out-of-bounds buffer-access vulnerabilities. The checks are implemented in **LoopGuardCheck** and **SizeGuardCheck**, respectively.

The solved and/or simplified path constraints computed by SAE's symbolic execution engine for the illegal data flows that may trigger a potential vulnerability as determined by the symbolic checks constitute the result of SAE. These constraints are meant to be used by AnonymousCompany's fuzzer. However, SAE's findings can also be helpful on their own. In particular, the paths issued for a potential vulnerability allow software developers to manually check the finding and makes the finding explainable.

7.3.3 Design and Implementation

Next, we elaborate on the design and implementation of SAE in more detail (cf. **Engine**). We use our running example from Section 7.3.1 to discuss the various usages modes of SAE.

Taint Configurations

SAE's taint analysis has two purposes in this project. First, it allows one to compute ordinary taint flows that represent undesired data flows—so-called *leaks*—in the *traditional* sense as described in Section 2.2 and Section 7.3.2. Second, it can also be used to compute program slices that lead from the target program's entry point(s) to potential vulnerabilities detected by SAE's symbolic analyses. SAE implements two symbolic analyses (or checks) to detect the vulnerabilities shown in Table 7.1: LoopGuard, a symbolic analysis that aims at proving finiteness of program loops and SizeGuard, a symbolic analysis that aims at detecting out-of-bounds buffer accesses. We next detail on how the taint analysis must be configured to run the usage modes:

1. Taint mode,
2. LoopGuard mode,
3. SizeGuard mode,

or any combination of these.

An overview of the connection between SAE's taint analysis and symbolic execution parts as well as the symbolic checks is shown in Figure 7.2.

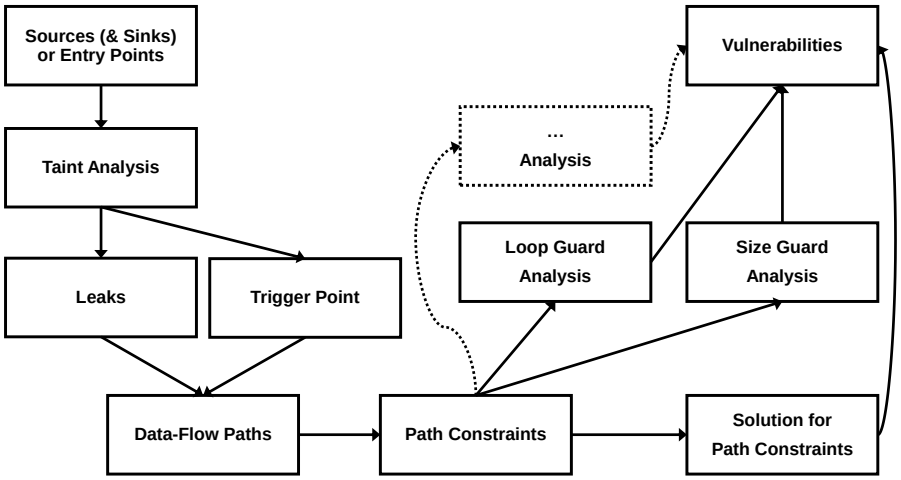


Figure 7.2: Overview of the interaction of SAE's taint analysis and symbolic analysis components.

Taint Analysis

Although the terms *sources*, *sinks*, *tainted* and *sanitized* originate from traditional taint analysis that is designed to find privacy leaks, a taint analysis can be parametrized to find any potentially interesting data flow from a given set of sources to a set of sinks. In the context of this project, our taint analysis can also be configured to compute program slices for potential out-of-bounds buffer accesses and infinite loops.

To support the *ordinary* Taint mode in the traditional sense, SAE's taint analysis uses the description of sources, sinks, and sanitizers as specified using a JSON configuration file or the compiler annotations as described in Section 7.3.2. In our running example in Listing 7.4, the taint analysis would treat the pointer typed variable **argv** in the **main** function that is annotated using a custom compiler annotation as a source and thus, spawn a taint analysis at Line 92 to track **argv** through the program. The taint analysis propagates this variable through the program and once the taint analysis encounters that the tainted variable (directly or transitively) reaches a call to a function whose parameters have been annotated as sinks, the analysis reports a leak.

To avoid overtainting as described in Section 7.3.2, the taint analysis has been implemented to be as precise as technically feasible. Therefore, SAE's taint analysis has been implemented to be flow-, context-, field- and path-sensitive. It has been implemented as a distributive data-flow problem within PhASAR and is solved within the IDE [SRH96] framework which enables unbounded flow- and context sensitivity by design.

Field sensitivity has been explicitly implemented using the *k-limited field-access paths* approach. The analysis hence does not propagate plain values, i.e., variables as data-flow facts, but larger structures that represent abstract memory locations more precisely. In this project, we modeled an abstract memory location as a base value, e.g. an instance variable of type **MailBlock** and a vector of byte offsets that are applied to the base value in

order to reach the tainted value that is represented by this abstract memory location. The byte offsets are separated by memory loads and allow one to distinguish between different fields of struct or array typed variables. Let us observe the field access **mail_p->content**, for instance. Its base value is **mail_p** which is of type **MailBlock**. To reach the field **content**, its byte offset must be added to the base pointer **mail_p**. In this case, the byte offset is **sizeof(MailEnvelope)** which is 16 bytes (assuming, an **unsigned int** is represented using four bytes) plus four bytes for the **mailBlockSize** field (assuming no padding between the fields has been added). After adding the byte offset, the value at this memory location can be loaded. As no further pointer arithmetic is applied, the value that is loaded is an integer, and the next byte offset is 0. (Otherwise, more entries for representing byte offsets would be added to the vector component of the data-flow fact.) The taint analysis would thus propagate the field-access path $\langle \texttt{mail\_p} \mid 20, 0 \rangle$ to model the memory location **mail_p->content**. Similarly, for **mail_p->envelope.contentSize**, the analysis computes the access path $\langle \texttt{mail\_p} \mid 12, 0 \rangle$, which does not overlap with the array's contents as this representation shows. Hence, the analysis is able to distinguish **mail_p->envelope.contentSize** from **mail_p->content[index]**, for instance. The abstract memory locations are implemented in **AbstractMemoryLocation**.

The length of the access paths, however, must be limited to a constant value k, which is 3 by default in our setting, to ensure termination. When field-accesses lead to access paths longer than k, the analysis applies an overapproximation and considers the whole "remaining struct" that exceeds three field accesses as tainted. The k-limiting field-access paths are precise enough to handle all of AnonymousCompany's test examples. While maintaining full context-, flow- and field-sensitivity at the same time is generally undecidable [Rep00], a k-limiting approach for modelling field-sensitivity can still produce precise data-flow results for realistic programs.

To compute the data-flow path(s) along which a data flow may occur, we implemented a **PathSensitivityManager** that allows one to record the individual flow functions and corresponding instructions. The **PathSensitivityManager** is integrated with PhASAR's IDE solver implementation and records flow functions while the client data-flow analysis problem, i.e., the taint analysis, is being solved. By integrating the functionalities for path recording directly with PhASAR's generic IDE solver, we enabled PhASAR's users to compute the precise data-flow paths for all IFDS- and IDE-based analysis problems. Our implementation also offers an interface to query the data-flow paths for any combination of instructions and data-flow facts that have been computed.

To compute program slices in LoopGuard and SizeGuard mode, the taint analysis is parameterized in a special manner. The symbolic checks for detecting dedicated program vulnerabilities are required to provide a special callback function that identifies what we call *trigger points* (cf. Figure 7.2). Trigger points represent points in the program, i.e., instructions, that may cause a vulnerability and should be inspected in more detail using symbolic execution. The checks' callback functions are registered to the **TaintConfig** type that describes a concrete taint configuration and are treated as sinks by the taint analysis. Thus, whenever the taint analysis finds a data flow that reaches a trigger point, a *leak*—or in this case, a *trigger*—is detected that will be used to spawn symbolic analyses that aim at confirming the vulnerabilities that the taint analysis found.

The LoopGuard check tries to prove finiteness for *all* loops in the target program. Therefore, the LoopGuard check uses the taint analysis and specifies the tautological Λ data-flow fact (also known as *zero value* cf. Section 2.2) as a source value and propagates it through the program starting at all specified entry points. This allows the taint analysis to compute all reachable paths from the target program's entry points to the LoopGuard check's respective trigger points. The LoopGuard check considers as trigger points all LLVM IR instructions that introduce loops.

The SizeGuard check aims at detecting out-of-bounds buffer accesses. It uses as sources for the taint analysis all buffer variables that SAE's users have marked as a source. In the example in Listing 7.4, the **mail_p** variable would be considered as a source and the taint analysis would propagate it through the program. The SizeGuard check requires a second input, i.e., an annotation that describes the buffer and its respective size that is to be "guarded". Line 19 and Line 12 in Listing 7.3 represent such SizeGuard annotations for the array typed variable **content** and its respective size that is kept track of with help of the **contentSize** variable. These size guard annotations allow the SAE to detect all places in the code that use annotated arrays. The SizeGuard analysis treats all places at which an annotated array is dereferenced as trigger points, since they may cause a vulnerability when the respective index is not in the interval $[0, \mathit{contentSize} - 1]$. Array dereferences hence act as sinks in SizeGuard mode and are communicated as such to the taint analysis. This way, it can compute the respective program slices of interest.

Path Sensitivity and Performance Optimizations

The taint analysis described in Section 7.3.3 produces as a result a set of leaks—or data flows of interest—which may be *ordinary leaks* (mode 1) or *trigger points* for the symbolic checks (mode 2 and 3) (cf. Figure 7.2).

To enable verification of the findings reported by the taint analysis and the symbolic checks that are conducted in the subsequent steps, the exact data-flow paths are required. Functionalities to record paths have been implemented in **PathSensitivityManager** as part of PhASAR's IDE solver implementation as described in Section 7.3.2.

Unfortunately, recording data-flow paths is hard to scale to large target programs. SAE implements a suitable solution based on the approach presented in [LHBM14], which uses a sparse representation for data-flow paths to counteract path explosion. We sketch our solution for computing the paths by presenting an example in what follows.

Figure 7.3 shows an exemplary C (or C++) program and its respective ICFG that serves as a target for a taint analysis. We assume that the taint analysis generates variables **x** and **y** as data-flow facts and propagates them through the program.

Our path sensitivity implementation works in several stages: first, each edge in the exploded super-graph (ESG) constructed by PhASAR's IDE [RHS95] solver implementation is recorded explicitly. We use the sparse graph representation described in [LHBM14] to keep the memory consumption low. For any leak detected by the taint analysis, we locate the corresponding node in the sparse ESG (that represents the program point at which the leak ocurred and the leaked value) and follow all edges in backward direction until we reach the source that initially generated the leaked value. In our example, the tainted variable **y** is

```
1  int compute(int i) {
2    int r = i;
3    return r;
4  }
5
6  int main(int argc,
7           char *argv[]) {
8    int x = source();
9    int y = source();
10   if (argc > 1) {
11     x = 42;
12   } else {
13     y = compute(x);
14   }
15   int r = x + y;
16   sink(y);
17   return 0;
18 }
```

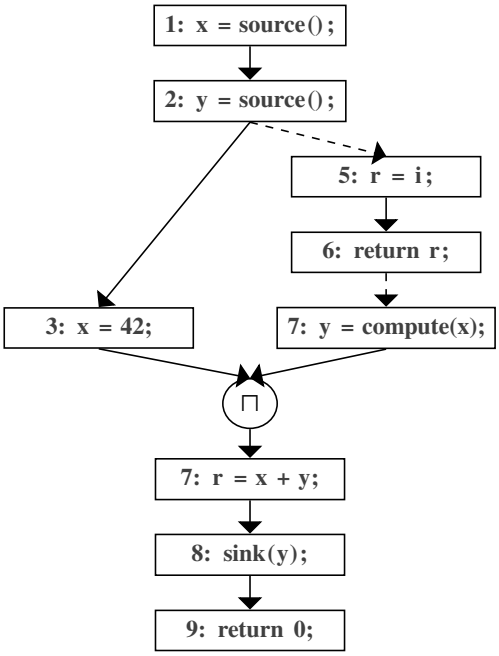


Figure 7.3: An exemplary program and its ICFG. Each node of the ICFG is labeled with an id for later reference. Solid edges (→) denote intra-procedural and dashed edges (⇢) denote inter-procedural control flows.

leaked at statement 8. The corresponding ESG computed by the taint analysis is shown in Figure 7.4. Red edges denote the paths that the path-sensitive query traverses for the leak at statement 8. Due to the merge point at statement 7, there are two possible program paths that potentially cause the leak. The ESG nodes for statements 7 and 8 are collapsed for each data-flow fact, because they do not represent (conditional) branches.

During the traversal of the ESG, we build a new data structure: a directed acyclic graph (DAG). This DAG contains the visited parts of the ESG in an even more condensed manner. Consecutive ESG nodes that do not contain branch instructions are merged into a single node. While traversing the exemplary ESG, the DAG shown in Figure 7.5 is built and represents the same set of paths in a much more compact way. The edges and partial paths of the DAG nodes are in reverse direction of control-flow. This is because the ESG was traversed backwards w.r.t. the program's control flow.

For each loop in the program's (inter-procedural) control-flow graph, the ESG contains a cycle for each data-flow fact that holds within the loop, making it difficult to traverse. Hence, we construct the DAG while ensuring that instructions within loop structures along the data-flow paths are not overlooked. For each loop, we compute two paths: One path that skips the loop and another path that visits the loop's body exactly once. This way, we do not miss any program instructions, but are able to build a loop-free sub-graph of the ESG. Since we are only considering single loop iterations, we potentially introduce unsoundness to the path constraints. However, we need to apply some bound if we wish to avoid

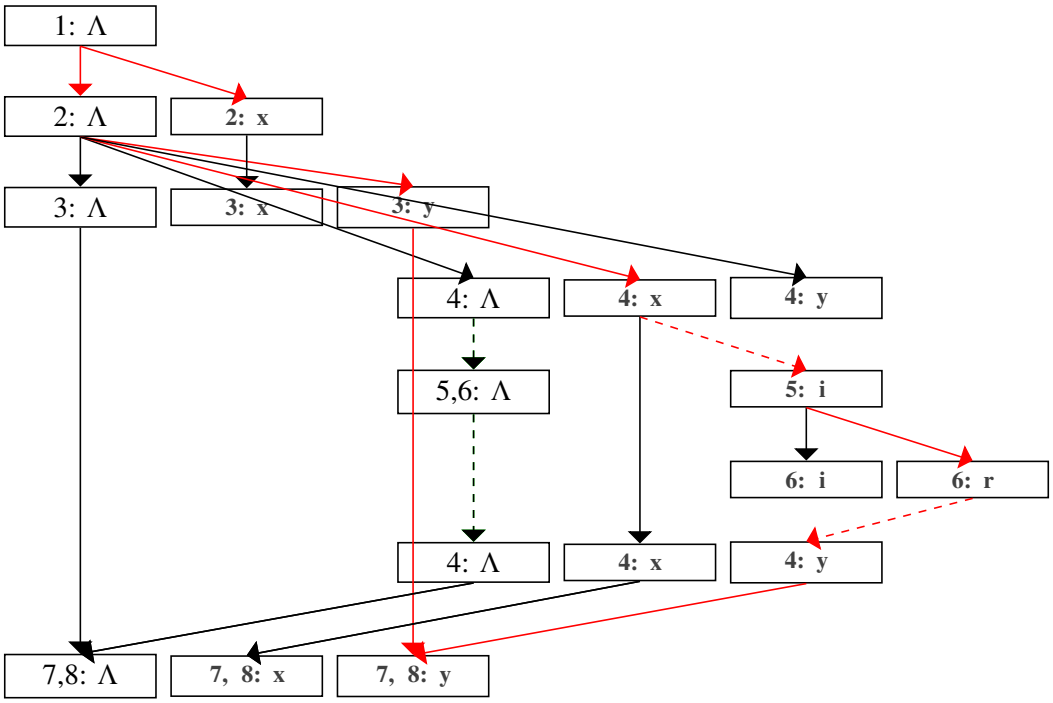


Figure 7.4: Example ESG for the C (or C++) program shown in Figure 7.3. Red edges denote the leaking paths, solid edges (⟶) denote intra-procedural and dashed edges (⇢) denote inter-procedural data flows.

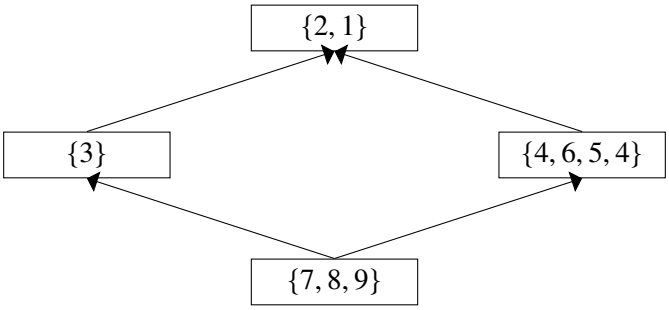


Figure 7.5: Example DAG for the leak at statement 8 (`sink(y);`) in the C (or C++) program shown in Figure 7.3.

introducing sound constraints that would likely prevent us from computing results that are actually useful. The Clang Static Analyzer's symbolic execution implementation [Cla18b], for instance, will analyze a loop at most four times in its default setting.

Symbolic Execution

SAE's symbolic execution parts analyze the data-flow paths issued by the taint analysis for every interesting finding and creates their corresponding constraints as Z3 [dMB08] expressions. The Z3 expressions obtained this way describe the constraints under which the different paths execute at runtime. These constraints can be fed into AnonymousCompany's downstream fuzzing infrastructure. In addition, (parts of) the constraints can be used to further manipulate fuzzing inputs to allow for diving deep into the target program.

Path Constraints Similar to statically finding all program paths that may lead to a vulnerability, analyzing and solving the associated path constraints is computationally expensive, too. It is essential to use an affordable representation when performing operations on paths. We hence use the DAG-based representation obtained by the ESG transformation described in Section 7.3.3 for efficient path extraction.

We reverse the direction of the DAG's edges such that the edges now point *in* direction of control flow, again, before we then extract the Z3 constraints x_n for each node n in the DAG from the LLVM IR branch instructions in n. In addition, we collect the constraints y_n for each DAG node under which the vulnerability location is reachable from that node. This is done by combining the constraints y from all successor nodes using disjunction ($\vee$) in a bottom-up fashion and conjoining them with x_n. For each node n with successors $s_{n,1}, \ldots s_{n,k}$ the reachability constraint is computed as shown in Equation (7.3).

$$y_n = x_n \wedge (\bigvee_{i=1}^{k} y_{s_{n,i}}) \tag{7.3}$$

In case the condition y_n is contradictory, we remove the node n from the DAG. The collection of constraints is still path insensitive and therefore, very fast as we incrementally combine constraints from incoming edges using disjunction.

To achieve path sensitivity, we need to collect the constraints per path without merging them at control-flow merge points. We thus traverse the DAG in a depth-first-search manner. This way, the stack of nodes $\langle n_1, \ldots, n_k \rangle$ that are currently visited represents the exact path we are "looking at", which is a prefix of one or more paths leading to the vulnerability location. While recursively descending towards the vulnerability location, we perform multiple filtering steps to reduce the number of feasible paths as quickly as possible such as to avoid state space explosion to the extend possible. Our previously conducted taint analysis based on the *Interprocedural Distributive Environments* (IDE) [SRH96] framework is context-sensitive, but the computed ESG on its own is context insensitive. To retrieve context-sensitivity for the computed paths, we check call- and return sites on the fly and discard a path early whenever call and return sites do not match. Furthermore, we collect the previously computed constraint x_n for the currently visited DAG node n and incrementally combine it with the constraints of the other nodes $n_1, \ldots, n_k$ of the current path-prefix using conjunction; hence, the tentative constraint c_n is computed as shown in Equation (7.4).

$$c_n = x_n \wedge \left(\bigwedge_{i=1}^{k} x_{n_i}\right) \tag{7.4}$$

This allows us to incrementally build the constraint for the complete current path. c_n's definition differs from y_n's definition, because c_n is path-sensitive while y_n is not. For each node n that we add to the path that is currently explored, we query Z3 to check whether the path prefix that is currently explored has contradictory constraints. In this case, we discard the infeasible path early. If we would not call Z3 at all to decide whether we can drop an infeasible path, we would start enumerating the constraints for all possible paths which would seriously impact SAE's performance in terms of both, running time and memory consumption.

Calling the Z3 solver for each step in the DAG traversal, on the other hand, adds a significant amount of overhead to the computations. Therefore, we use a heuristic to decide whether it is worth calling the Z3 solver. The heuristic makes use of the fact that assuming a single constraint x_n is satisfiable, the conjunction of several satisfiable constraints can only become unsatisfiable if some atoms (i.e., Z3 expressions for single variables) are repeated across different x parts of the conjunction. If a new constraint of a node x_n only consists of atoms that do not occur in the rest of the path constraints' $x_{n_1}, \ldots, x_{n_k}$ satisfiability cannot change and hence, SAE does not need to call the Z3 solver again.

Finally, whenever the DAG traversal reaches the vulnerability location, we save the path $\langle n_1, \ldots, n_k, n \rangle$ with its constraint c_n and Z3's solution model such that we can forward it to AnonymousCompany's fuzzer. Note that although a DAG node n can be visited multiple times during the depth-first-search, the path constraint c_n might differ every time, since it depends on the complete path that leads to n.

Using the algorithm described in the above, our path-sensitive symbolic execution is significantly more scalable compared to our earlier implementations that did not discard

paths with contradictory constraints early. However, the number of paths that need to be explored in the final depth-first-traversal is still exponential in the number of branches in the program slice(s) of interest. We therefore implemented two additional measures to keep the running times manageable for larger programs.

First, we offer the option to limit the depth of the DAG. If SAE's command-line option `-path-length-threshold` is set, we cut off the DAG such that its depth is limited by a user-specified threshold. This optimization is unlikely to miss any important path constraints as—in the common scenarios we observed in the test programs—the reachability of a vulnerability location is determined by the program parts that are "close" in terms of control-flow distance. Yet, a residual risk in missing path constraints remains.

Second, we offer a fallback solution, which applies a path-insensitive symbolic execution whenever the number of nodes in the DAG exceeds the threshold value provided by the command-line option `-path-insensitivity-fallback-threshold`. When computing path constraints in an path-insensitive manner, we are still able to report a Z3 constraint (the ORed y_n constraints for all starting points of the analysis) with its corresponding solution model in a reasonable amount of time. This optimization is guaranteed to provide a sound overapproximation of the path constraints. If one of these two parameters is not specified explicitly by the user, the parameters are set to a reasonable default that we determined with help of the test programs provided by AnonymousCompany.

Additionally, we also provide the command-line option `-path-count-threshold` to limit the total number of reported paths for a vulnerability location to a constant threshold that is set to 1000 by default. However, we discovered that this threshold is less useful when using a sensible parametrization of the DAG depth and the path-insensitivity fallback threshold. Thus, limiting the depth and specifying a fallback threshold should be the primary points of interaction for SAE's users.

Next, we detail on how we designed the symbolic checkers that aim at proving finiteness of loop constructs and detecting out-of-bounds buffer accesses.

Symbolic Loop Finiteness Check It is generally undecidable to automatically determine whether a loop will terminate or not. However, in parsing code such as provided in the running example in Listing 7.4 loops have the following properties that we exploit to attempt proofs of termination (or non-termination): First, the loop condition consists of relational operators that compare a *loop-variant* and a *loop-invariant* value. Second, the loop-variant value is *monotonic*, that is, in every loop iteration it changes only towards one direction, if it changes at all.

Consider the `while` loop in `handler_5` in Listing 7.4. In its condition `len` is the loop-variant value and `subMail_p->subContentSize` is the loop-invariant value.

For proving the loop's finiteness, one first needs to check whether these conditions are met. We consider a variable loop invariant if and only if it is not changed during loop execution. This is the case if no store in the loop alters the variable directly or indirectly via an aliasing pointer. For the loop-variant variable in the loop condition, we recursively define a Z3 function that describes the effects of the loop to this variable. Then, we compose the inner-loop operations for the loop-variant condition variable. For any non-invariant

operand in inner-loop operations for the loop-variant condition variable, we introduce an opaque Z3 function and introduce constraints for its domain.

With this recursive function defined we query Z3 to prove its monotonicity. These proofs are done indirectly: we query Z3 to check whether the negation of the claim is satisfiable. If that is the case, then Z3 provides a possible valuation which constitutes a counterexample. Otherwise, the claim is proved.

For the `while` loop in `handler_5` monotonic growth for `len` is proved, as every loop iteration adds the unsigned value `subLoopSize` to it.

Monotonicity direction is determined by the operator. For `<`, `<=`, `>=`, `>`, we choose the direction that eventually allows the condition not to be fulfilled anymore. For `!=`, we determine for both directions whether they are feasible.

If monotonicity is successfully proven, we proceed to prove *strict monotonicity*. That is, the loop-variant variable cannot be the same for two loop iterations. If that condition is met, termination of the loop is guaranteed. The proof query is similar to the proof query for non-strict monotonicity, except equality is not allowed.

For the `while` loop in `handler_5` a counterexample against strict monotonicity for `len` is found: if `subLoopSize == 0`, `len` remains unchanged and no progress is made, showing that this loop is possibly infinite.

Checking loop finiteness is implemented as a symbolic analysis in **LoopGuardCheck**. Since SAE targets LLVM intermediate representation, **LoopGuardCheck** does not only consider plain `while` loops as shown in the above, but loops with irregular loop termination via break, return, and goto, too.

Symbolic Out-Of-Bound Buffer Access Check Out-of-bounds buffer accesses present a frequently occurring programming error in low-level C and C++ that may cause undefined behavior and can lead to serious (security) vulnerabilities. Especially in the C programming language, there is no dedicated array type and thus, arrays are managed as a pointer to a contiguous block of memory and its respective size. Since the pointer and the size are two separate entities, chances are high that an incorrect size is being used while processing an array.

To statically check if a memory access is within valid bounds, we developed the SizeGuard symbolic analysis in **SizeGuardCheck**. Besides requiring a specification of the variable whose memory accesses should be protected or *guarded* (cf. Line 95 in Listing 7.4), this check requires an additional piece of information. To be able to check if buffer accesses are indeed within valid bounds, it must be parameterized with the array that is to be protected (cf. Line 19 in Listing 7.4) and the array's respective size (cf. Line 12 in Listing 7.4). This is done by compiler annotations that qualify the pointer-size pairs as relevant to the SAE and assigns a unique id to them, **MailBlock::Contents** in the running example. Note that without a size-guard annotation, there is no reliable way in recognizing a connection between a buffer and its respective size. Such connections could potentially be inferred by means of machine learning and AI techniques, both of which are out-of-scope for this project.

Whenever an annotated array is accessed in the program, this access comprises a trigger point for the SizeGuard check. Once the taint analysis propagated all variables to be

protected through the program under analysis and all trigger points have been collected, a SizeGuard check is spawned for each trigger point to verify that the memory access is within valid bounds. The symbolic analysis considers the path constraints along all paths that lead to the trigger point under analysis to create an *environment* that represents a situation similar to the one that is established at runtime once this program point is reached. The analysis then aims at constructing a logical expression using Z3 that expresses the pointer offset. If the offset comprises more complex computations, these are modeled, too, to the extend that is technically feasible. Once all available information for a certain trigger point has been collected as a Z3 expression, the vulnerability condition that states that the memory access is not within valid bounds is checked. If the Z3 solver finds that the constructed solver instance comprising the collected constraints is actually solvable, a potential out-of-bound access has been found and a respective solution that states the actual constraints that may enable the program paths that lead to and trigger the vulnerability at runtime is issued.

Checking out-of-bounds buffer accesses is implemented as a symbolic analysis in **SizeGuardCheck**.

Implementing Custom Symbolic Checks We designed and implemented SAE such that it is open to extension for further symbolic checks as indicated by the dotted box **... Analysis** and dotted edges shown in Figure 7.2.

A custom symbolic check must fullfil the following requirements to be integrated with SAE: It must implement a trigger, i.e., a callback that is used by the taint analysis to determine if a particular data flow is of interest to the symbolic check. The callback that communicates trigger points to the taint analysis must have the function signature **std::set< llvm::Value const *>(llvm::Instruction const *) const**. The taint analysis will call this callback function for each instruction of the program and will check if one or more data-flow facts that it computed can be found within the set of values returned by the callback. If the taint analysis detects that a data-flow fact occurs in the returned set, it will treat this instruction as a trigger point. The taint analysis will query the analysis engine to compute (possibly all) program slices (depending on SAE's configuration) and their respective path constraints that lead from the analysis' starting points to the trigger point in question.

It will then use this information and query the symbolic check's second requirement: an implementation of **analyze(llvm::Instruction const *, LLVMPathConstraintsBase &)** to spawn the actual symbolic check. The symbolic check then symbolically analyzes the instruction of interest and is free to use the information on path constraints that is provided by SAE and made available in form of the reference typed parameter **LLVMPathConstraintsBase**. The symbolic check's **analyze** function has to return the result of its analysis. The **SizeGuardCheck**, for instance, returns a Z3 expression and the **LoopGuardCheck** returns an enum to communicate the loop's finiteness, but other return values are possible, too.

7.3.4 Results and How to Access Them

The static analysis engine provides various APIs to query different pieces of analysis information. These APIs are already integrated in the command-line tool `sa-engine` and are accessible using the command-line parameters. SAE can thus be used as a library and

as a command-line parameterizable static analyzer. In the following, we briefly discuss SAE's most important parameters and their output.

Path Collection

The **IDESolver** provides access to path information via a rich interface. It provides direct access to the raw data-flow analysis results via the **SolverResults getSolverResults()** function. Via the **SolverResults** type, a **PathBuilder** object can be created using the **PathBuilder pathBuilder (LLVMPathConstraintsBase*)** function.

The **PathBuilder** provides various customization options. In particular, the optimization thresholds explained in Section 7.3.3. Each customization option returns a new copy of the **PathBuilder** object with the corresponding option enabled. The **PathBuilder** type offers the functions **FlowPathSequence pathsTo(n_t, d_t)** and **MaybeFlowPathSeq pathsOrConstraintTo(n_t, d_t)** that provide their callers with the paths that lead to the queried ESG node. The ESG node and hence also the program location of interest is specified as LLVM IR instruction (**n_t**) and data-flow fact (**d_t**). Only **n_t**-**d_t** pairs are valid that have been obtained via the **getAllLeaks(const SolverResults &)** function of the **IDEExtendedTaintAnalysis**. Depending whether a path-(in)sensitive computation of constraints is desired, **pathsTo(n_t, d_t)** or **pathsOrConstraintTo(n_t, d_t)** can be used, respectively. When the fallback option is set, **pathsOrConstraintTo(n_t, d_t)** is being called, since it is able to return either a **FlowPathSequence** or the constraints in form of a single **z3::expr**. Otherwise, **pathsTo(n_t, d_t)** is used to indicate that such a fallback to path-insensitive analysis is disallowed. Details can be found in the implementation of the **PathSensitivityManager** type.

Emitting the Exploded Super-Graph

SAE allows one to export the exploded super-graph (ESG) that is constructed on-the-fly while solving the taint analysis problem using PhASAR's IDE solver. As output format, we support the DOT format.

The respective command-line flag `-emit-esg-as-dot` expects as a parameter a file path to which the DOT output should be written to. SAE's C++ API provides a member function **void printAsDot(std::ostream &)** that serves the same purpose in **IDESolver.h**.

Emitting the Inter-Procedural Control-Flow Graph

Similar to the ESG, the inter-procedural control-flow graph (ICFG) can be exported to a file. As output format, SAE supports an incidence-list in JSON.

The respective command-line option `-emit-icfg` accepts the output-file path as argument. For the C++ API, the **LLVMBasedICFG** (declared in **LLVMBasedICFG.h**) provides a member function **nlohmann::json exportICFGAsJson()**, which returns the JSON incidence-list representation for the receiver ICFG object.

Each edge in the JSON is represented as JSON object with two elements, *from* and *to* which are either LLVM IR instructions or more complex JSON objects modeling the source-code location. An example of the JSON with IR instructions is shown in Listing 7.5.

```
[
  {
    "from": "%i.addr = alloca i32, align 4 | ID: 5",
    "to": "store i32 %i, i32* %i.addr, align 4 | ID: 6"
  },
  {
    "from": "store i32 %i, i32* %i.addr, align 4 | ID: 6",
    "to": "%i.addr1 = bitcast i32* %i.addr to i8* | ID: 8"
  },
  ...
]
```

Listing 7.5: Format of the inter-procedural control-flow graph.

Emitting Analysis Coverage

SAE supports emitting information on file coverage and instruction coverage by the static analysis. The coverage information is output in YAML representation.

On the command line, the file path can be specified using the `-analysis-coverage-output-file` option. Otherwise, the standard output stream is used by default. The options `-emitanalysis-file-coverage` and `-emit-analysis-inst-coverage` enable the output of file and instruction coverage, respectively. Listing 7.6 shows an example in which both, file and instruction coverage options are enabled.

```
Files covered by the analysis:
    - "demo_pathsensitivity/demo_04.cpp"
Number of files covered by the analysis: 1
Number of total files in the project: 1
Amount of files covered by the analysis[%]: 100.00
Number of instructions covered by the analysis: 10
Total number of instructions in the project: 46
Amount of instructions covered by the analysis[%]: 21.74
```

Listing 7.6: File and instruction coverage reported by static analysis engine.

The coverage information can be retrieved by calling
std::vector<std::string> getSourceFilesCoveredByAnalysis() and
size_t getNumInstructionsCoveredByAnalysis() defined as part of the **ExplicitESG** type, when using the corresponding C++ APIs. To compute relative coverage numbers, the total files and instructions in the project under analysis can be retrieved by calling **std::vector<std::string> getAllSourceFiles()** and **size_t getNumInstructions()** defined as part of the **ProjectIRDB** type in **ProjectIRDB.h**.

Emitting Full JSON Reports

SAE outputs detailed information on the analysis results to the command-line. However, sometimes a plain command-line output is not desired. The command-line output can be disabled using the `-silent` command-line flag. All analysis results can also be written to a file in JSON format. An example of such a JSON report is shown in Listing 7.7.

```
{
    "Vulnerabilities": [
        {
            "VulnerabilityLocation": {
                "InstructionId": "39",
                "LLVMInstruction": "call void @_Z5printi(i32 %3), !
                    dbg !89, !psr.id !90 | ID: 39",
                "Paths": [
                    {
                        "PathConstraint": {
                            "Constraint": ";
                                (set-info :status unknown)
                                (declare-fun argc () Int)
                                (assert
                                    (not (<= argc 1)))
                                (check-sat)",
                            "Model": "(define-fun argc () Int  2)"
                        },
                        "ProgramSlice": [
                            "main - int main(int argc, char **argv)
                                {",
                            "main -   int a = 42;",
                            "main -   int b = a;",
                            "main -   int c = b;",
                            "main -   int a = 42;",
                            "main -   if (argc > 1) {",
                            "main -     a = getPasswd();",
                            "main -   } else {",
                            "main -   int b = a;",
                            "main -   int c = b;",
                            "main -   print(c);"
                        ]
                    }
                ],
                "SourceCodeInformation": {
                    "Column": 3,
                    "Line": 17,
                    "SourceCodeFilename": "demo\_03.cpp",
                    "SourceCodeFunctionName": "main",
                    "SourceCodeLine": "print(c);"
                },
                "SourceFilesCovered": [
                    "demo\_03.cpp"
                ]
            },
            "VulnerabilityType": "ordinary leak"
        }
    ]
}
```

Listing 7.7: Exemplary report in json format generated by SAE.

The JSON output can be enabled via the `-emit-json-report` command-line option which accepts a file path. The output is structured as follows: At the top-level, there is a JSON object containing a single key **Vulnerabilities** that maps to an array of JSON objects representing all vulnerabilities of a specific vulnerability category, respectively. A vulnerability object consists of two JSON keys, **VulnerabilityType** and **VulnerabilityLocation**. The **VulnerabilityType** specifies the kind of vulnerability found, i.e., "ordinary leak", "out-of-bounds access" or "infinite loop" as described in Section 7.3.3. In **VulnerabilityLocation**, the concrete vulnerability is represented as JSON object with keys describing the vulnerability location as well as the paths, path constraints and file coverage in detail. Most importantly, it contains the keys **Paths** and **SourceCodeInformation**. The **SourceCodeInformation** contains details on the coordinates in the source code at which the potential vulnerability has been identified. The **Paths** consists of an array of JSON objects each of which represents a path leading to the vulnerability location. Each path consists of two parts: the covered program slice and the corresponding path constraint in SMT2 format. The program slice is stored as JSON array of strings and each string is split by the – character into two parts: the first part is the function that hosts the statement and the second part is the program statement.

7.3.5 Insights and Lessons Learned

We next detail the key insights and experiences that we made throughout this 15-month industry project.

Developers of static program analyzers know that virtually all algorithms involved in static program analysis can be heavily parameterized and that these parameterized algorithms even affect each others results and running times in non-trivial manners [SLHB21, SLHB19, Bod18]. Therefore, it is clear that static analysis developers implement the respective algorithms in a highly parameterizable manner. The more information on a given target project the users of static analyzers provide, the more qualified parameters can be chosen for the algorithms and the better the findings reported by those analyzers. During this project, however, we quickly learned that—from a user's perspective—the opposite is desired. Users from industry, even when being expert-level developers, most often do not know how static analysis conceptually works and thus, even when desired additional information on the target project is theoretically available, cannot provide it to the analyzer in a way that is useful to improve its results. Static analysis writers hence rather need to provide sensible default parameters, if possible and otherwise should implement lightweight pre-analysis to determine suitable parameters for the analysis algorithms. Even non-optimal parameters determined by a heuristic are still better than inputs of uninformed analysis users.

Specifying *sources* and *sinks* to parameterize a taint analysis, for instance, is necessary to run SAE. Ideally, however, sources and sinks could be determined automatically. Several research approaches aim at automating the sensible selection of *sources* and *sinks* [PDB19, RAB14] to relief analysis users from this burden. The same holds for specifying buffer-size pairs for our symbolic SizeGuard checker. In the C programming language, there are no dedicated array types (that carry an array's size) and instead, an array is usually resembled by a pointer to the first element and a separate entity that (hopefully) carries its correct respective size. Due to the definition of this project and its time schedule, we require

SAE's users to manually specify sources and sinks, and the arrays to be protected and their respective variables that carry the sizes. In future work, it would be intriguing to check to what extend buffer-size pairs can be automatically mined from a software project written in C or C++.

When requiring manual user inputs for a static analyzer, communication between developers and analysis setup is necessary. Initially, we implemented the taint configuration as well as the configuration of buffer-size pairs for the symbolic check of buffer overflows as source code annotations. Modern compilers oftentimes allow for custom source code annotations that can be used and examined downstream in the compiler pipeline. The Clang compiler, used to generate the LLVM IR that PhASAR is targeting, for instance, allows developers to use custom source code annotations. While code annotations using implementation-defined language extensions have been available in the major compilers for years, *attribute specifier sequences* provide the unified standard syntax and are officially part of C since C23 and C++ since C++11. This enables developers to pass additional information to a downstream analysis tool. Clang's code generator preserves this information when generating LLVM IR and an analysis tool can then consume the parametrization as part of the code analysis. This approach follows the "code is documentation" principal and generally allows developers to communicate additional information precisely and clearly to a program analyzer. For some critical pieces of the target software, however, such source code annotations are undesired or even prohibited by company policy as we learned in this project. This required us to provide an additional option for an external parametrization; we choose configuration files in JSON format. It is hence advisable to provide several sensible configuration options for the various analysis parametrizations, depending on the requirements for the target source code.

Another aspect that cannot be overstated is the cognitive complexity involved in developing such analyzers based on state-of-the-art static analysis concepts. We can confirm the many horrors static analysis writers face when building industrial-strength static analyzers as described by Toman and Grossman [TG17]. Not only are the textbook implementations of IFDS [RHS95], IDE [SRH96] or WPDS [RSJ03] data-flow solvers and symbolic execution hard to get right, they do not suffice to solve actual analysis problems in industry when being used in their plain vanilla version. We had to apply various implementation tricks to get SAE running on the test programs provided by the industry partner within the given time and memory budgets. A few of such optimizations for path sensitivity and symbolic execution have been described in Section 7.3.3. Additionally, we had to heavily optimize the data-flow domain to encode field sensitivity as efficiently as possible such as to generate as little data-flow facts as possible. Since the time complexity of IFDS/IDE is $O(|N| \cdot |D|^3)$, when exhaustively computing all source-to-sink data flows—where N is the set of nodes of the target program's ICFG and D is the data-flow domain—it is advisable to keep the size of D as small as possible. The exploded super-graph built by IFDS/IDE is bounded in size by $O(|N| \cdot |D|)$. This, again, required us to take great care to keep the number of data-flow facts small and use as few bytes as possible to represent an abstract memory location as a data-flow fact (cf. Section 7.3.3) to avoid running out-of-memory during analysis of larger programs. Each byte—also due to potential padding—quickly adds up and can significantly increase the memory requirements of an analysis. Running

times are affected as well by the size of the data-flow facts because of hardware caching of the machine the analysis is running on.

The complexity could be (at least) partially tamed by assigning members of our team to the individual components and making them the experts of those components during the project. With help of clean interfaces we defined strict bounds of the various complex analysis algorithms involved such that each team member could mainly concentrate on their own analysis algorithms and topics. While the cognitive complexity could be managed within the team this way, we encountered difficulties while shipping the project. Since the result of our project is not a piece of ready-to-use, end-user software, but rather a piece of complex special static analysis infrastructure, it required a lot of time to sort out issues on how to use and parameterize SAE and to explain its internals to the customer such that is could be integrated with AnonymousCompany's internal fuzzing infrastructure. In future projects, we will allocate a sufficiently large and dedicated amount of project time to provide support for the roll-out of such analyzers at the customer's site.

7.4 Conclusions

In this chapter, we presented two projects from academia and a large industry project all of which used PhASAR as a major component to solve interesting and relevant problems.

In Section 7.1, we showed that by combining high-level information obtained by repository mining with information obtained by low-level data-flow analysis opens up a variety of novel applications. Both fields, repository mining and static analysis, which have been previously successful on their own, can benefit from another and when combined, provide new interesting insights on software projects. Without the infrastructure provided by the PhASAR framework, our work on integrating program analysis and repository mining would not have been possible, since there are no alternative inter-procedural data-flow analysis frameworks targeting C and C++ that could have been used. And building a static data-flow analysis framework from scratch for the sole purpose of using it for implementing a single new research approach is too expensive.[1]

PhASAR could also benefit the research on static configuration-logic identification as described in Section 7.2. In this research project, we were able to use a variety of PhASAR's features to compute callgraphs and data-flow information to eventually implement a novel algorithm that computes Boundary locations that separate a program's configuration logic from its main logic that is responsible for the actual computations. This ultimately allows for fully automating software debloating approaches like LMCAS [AJR+22] or the `temporal-specialization` tool [GPMP20], for instance. Previously, these approaches could only be used in a semi-automated manner and required its users to manually specify the target program's Boundary location. Again, this work would too be virtually infeasible without the static analysis toolbox provided by the PhASAR project.

Lastly, we could successfully apply PhASAR in an industry project to find real buffer overflows and infinite loop vulnerabilities in router software written in low-level C and C++ as described in Section 7.3. We used PhASAR to implement a precise, parameterizable,

[1] . . . and requires a PhD thesis on its own as this documents shows.

inter-procedural, context-, flow-, field- and path-sensitive taint analysis that serves as a filter for a subsequent symbolic analysis that we implemented with help of Z3 [dMB08]. The PhASAR-based taint analysis represents a major part of the SAE analyzer that we developed within this project.

All of the applications presented in this section show that PhASAR is a framework that can indeed be used to solve demanding problems in both, academia and industry.

8 Conclusions and Future Work

Static program analysis can help to find bugs and (security) vulnerabilities, but only if being used in a precise and inter-procedural setup. Traditional static analysis offers a range of scalable approaches and algorithms for computing points-to, callgraph and data-flow information. However, even though these approaches can scale relatively well up to midsize programs, they are rendered unusable when being directly applied—in whole-program mode—to modern software development that is characterized by extensive library usage and frequent code changes, or to target programs that comprise several million lines of code. Existing approaches are also only concerned with computing individual pieces of static analysis information such as points-to, callgraph or data-flow information. All this becomes an issue when trying to conduct concrete client analyses that aim at finding complex bugs and (security) vulnerabilities in modern software development workflows and require a multitude of different analysis information.

In this thesis, we have presented PhASAR, the tremendous engineering efforts that have been necessary to develop this infrastructure, and the VarAlyzer, ModAlyzer and IncAlyzer approaches built on top of PhASAR that address these issues.

PhASAR is a core element of this work and provides the main building blocks to implement complex static analysis approaches. It allows its users to implement concrete client analyses, hack on existing analysis implementations or prototype completely new tools and analyzers. PhASAR provides its users with implementations for all algorithms that are required to conduct precise, inter-procedural static analysis. Besides having provided a detailed description of the PhASAR analysis framework, we also elaborated on poor design choices we made in early version of PhASAR and detailed on how we have overcome them. We also shared important lessons learned and presented particular difficult design and implementation challenges.

Our VarAlyzer approach, the second contribution of this thesis, for the first time enables one to analyze software product lines written in C using inter-procedural data-flow analysis. It extends the class of data-flow problems that can be efficiently solved on software product lines by enabling its users to solve general, distributive problems formulated in the Interprocedural Distributive Environments framework. We have further shown how an existing, complex typestate analysis, which can only be efficiently formulated within IDE, can now be solved in a variability-aware manner using VarAlyzer. VarAlyzer's analysis time is only affected by the size of the target program, but not its features, whereas the analysis time of the product-based approach grows exponentially in the number of features. Our empirical evaluation shows that VarAlyzer's results coincide with the results computed by the product-based approach. We were also able to identify several fundamental challenges that still must be overcome to make the analysis of full software product lines in C a reality and provided ideas on how to address these remaining challenges.

ModAlyzer, another contribution of this thesis, is an integrated approach that aims at speeding up concrete client analyses by summarizing parts of the code that do not change frequently. It computes analysis summaries for all the various pieces of information required to solve a concrete client analysis, mediates their mutual interactions and persists these summaries. These summaries can be computed in a separate offline phase and can then be loaded at analysis time to massively speed up the analysis of the actual application code by avoiding expensive and unnecessary reanalysis of the program parts previously summarized. Our empirical evaluation confirms that the greater the parts of the program that can be analyzed and summarized upfront, the greater the speedup that can be achieved using ModAlyzer. This can bring analysis times down to the point at which they can be integrated into a project's build pipeline. Our approach is only effective, however, if the target project contains parts that do not change frequently, of course. We also showed that ModAlyzer provides the same results as a matching whole-program analysis that computes everything from scratch.

In our final contribution, we made static analysis incremental with help of IncAlyzer. IncAlyzer is built on top of ModAlyzer and helps to further reduce analysis times by incrementally analyzing the parts of a program that do change frequently. Like ModAlyzer, IncAlyzer's results are equal to those computed by a matching whole-program analysis that ignores information already computed in previous analysis runs. The IncAlyzer approach has the potential to bring analysis times of deep, inter-procedural static analysis to sub-build-pipeline levels, allowing them to be potentially run on a developer's machine to compute analysis findings within a few minutes or even seconds, depending on the extend of the changes made to the code. The combination of ModAlyzer and IncAlyzer also allows for new software development workflows in which static analysis information is attached to and persisted with the program code under analysis. This capability can potentially save lots of analysis time and thus also electrical energy by avoiding unnecessary repeated reanalysis of the same code across different machines.

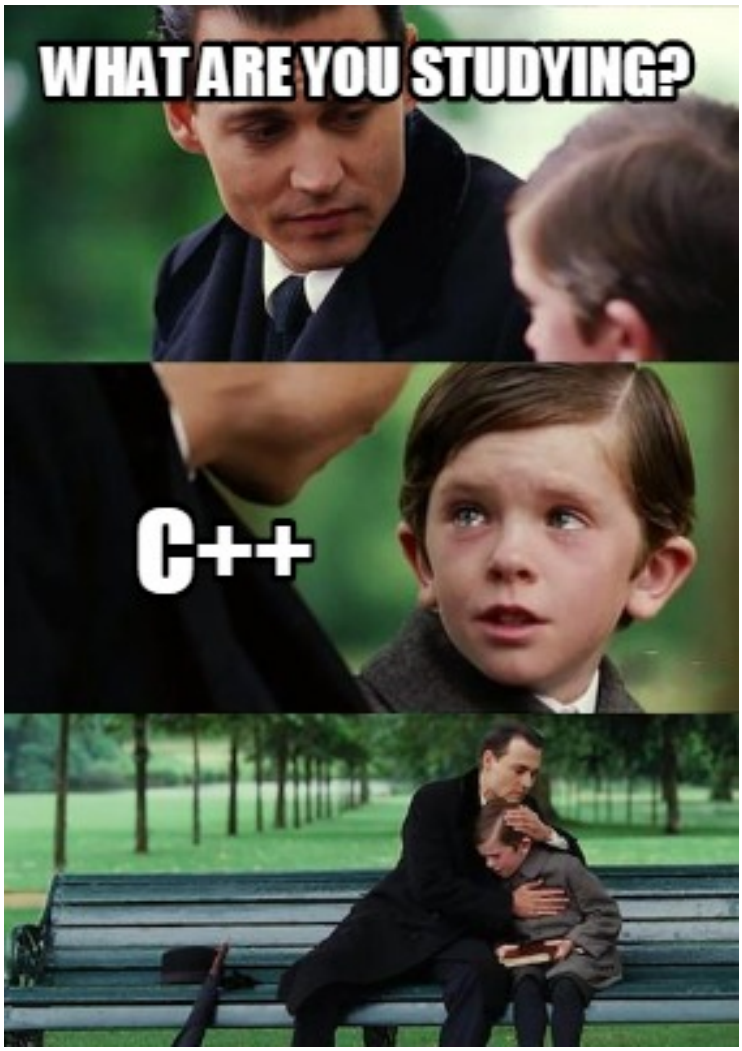
All of our contributions help to make static program analysis more accessible in practice by simplifying the task of writing a static analysis and by providing new integrated analysis strategies that match modern software development workflows.

Finally, we closed out the thesis by highlighting two research projects and one large industry project that PhASAR made possible. The author of this thesis selected these particular projects for presentation here, since he was heavily involved with these projects. As Github's data on PhASAR suggests, there are a lot more interesting projects built on top of PhASAR and we hope that we could help the corresponding authors of these projects.

> Precise, semantic whole-program analysis can be designed to fit modern software development in the 21th century and scaled to large, real-world software.

However, there are further directions that seem worth to be looked at in the future. One of these directions targets a summary format for more precise points-to information. Pointer analysis unfortunately cannot be expressed as a distributive data-flow problem. And while flow-insensitive points-to information can still be summarized using pointer-assignment graphs, flow-sensitive points-to information is difficult to handle. Several previous works

aim at decomposing pointer analysis into smaller problems that can then be expressed in distributive data-flow frameworks, again. It would be intriguing to evaluate possible summary formats for these approaches to potentially enable integration with the ModAlyzer and IncAlyzer approaches and evaluate the effects of more precise points-to information on the whole analysis stack and therefore also on the client analysis to be solved. Our approaches have been designed such that individual parts of the analysis stack can be easily changed and replaced. Another direction for future work concerns link-time variability of software product lines. While VarAlyzer is able to successfully conduct variability-aware, inter-procedural data-flow analysis, the build system and linking steps of a project can introduce further variability, which is currently not considered by our approach. To be able to consider these additional two sources of variability, one would first require a generic model of a project's build system. Second, one would need to design and construct a variability-aware symbol table that exceeds compilation times of individual translation units and instead, mediates information on symbols in a variability-aware manner until the program is completely linked.


WHAT ARE YOU STUDYING?
C++

Bibliography

[AB14] Steven Arzt and Eric Bodden. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 288–298, New York, NY, USA, 2014. ACM.

[AB16] Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 725–735, New York, NY, USA, 2016. ACM.

[ADA09] RED SOFT ADAIR. Is there a working c refactoring tool?, September 2009. `https://stackoverflow.com/questions/1388469/is-there-a-working-c-refactoring-tool`.

[AH96] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in c++ programs. In Pierre Cointe, editor, *ECOOP '96 — Object-Oriented Programming*, pages 142–166, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[AJR+22] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroSP)*, pages 251–269, 2022.

[AL13] Karim Ali and Ondřej Lhoták. Averroes: Whole-program analysis without the whole program. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 378–400, Berlin, Heidelberg, 2013. Springer-Verlag.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, May 1994. `https://www.cs.cornell.edu/courses/cs711/2005fa/papers/andersen-thesis94.pdf`.

[ARF+14] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[Art21] Artifacts. Supplementary material, 2021. https://drive.google.com/drive/folders/1ESiSu5iKsFTrM2XqN3Oj4fhIqVfdQ93W?usp=sharing.

[ASR+23] Mohannad Alhanahnah, Philipp Schubert, Thomas Reps, Somesh Jha, and Eric Bodden. slash: A technique for static configuration-logic identification, October 2023. https://arxiv.org/abs/2310.06758.

[AST23] *AST Matcher Reference*, January 2023. https://clang.llvm.org/docs/LibASTMatchersReference.html.

[AWS22] Amanzon aws codeguru reviewer, 2022. https://aws.amazon.com/codeguru/.

[Bab18] Personal communication with domagoj babic, google, 2018.

[Bay72] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, Dec 1972.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010.

[BCSD14] Caius Brindescu, Mihai Codoban, Sergii Shmarkatiuk, and Danny Dig. How do centralized and distributed version control systems impact software changes? In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 322–333, New York, NY, USA, 2014. Association for Computing Machinery.

[BGH+06] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BGS18] Dirk Beyer, Sumit Gulwani, and David A Schmidt. Combining model checking and data-flow analysis. In *Handbook of Model Checking*, pages 493–540. Springer, 2018.

[Bis20] Gnu bison, 2020. https://www.gnu.org/software/bison/.

[Bod12] Eric Bodden. Inter-procedural data-flow analysis with ifds/ide and soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 3–8, New York, NY, USA, 2012. ACM.

[Bod18] Eric Bodden. The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them). In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 85–93, New York, NY, USA, 2018. ACM.

[Boo18] Boost proto, August 2018. `https://www.boost.org/doc/libs/1_68_0/doc/html/proto.html`.

[Boo19] Boost serialization, August 2019. `https://www.boost.org/doc/libs/1_70_0/libs/serialization/doc/`.

[BRTB12] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 13–24, New York, NY, USA, 2012. Association for Computing Machinery.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 243–262, New York, NY, USA, 2009. ACM.

[BS16] George Balatsouras and Yannis Smaragdakis. Structure-sensitive points-to analysis for c and c++. In *International Static Analysis Symposium*, pages 84–104. Springer, 2016.

[BTR+13] Eric Bodden, Társis Tolêdo, Márcio Ribeiro, Claus Brabrand, Paulo Borba, and Mira Mezini. Spllift: Statically analyzing software product lines in minutes instead of years. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 355–364, New York, NY, USA, 2013. ACM.

[BWR+11] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[CCP17] Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *Proc. ACM Program. Lang.*, 2(POPL):30:1–30:30, December 2017.

[CCS+13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-Francois Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking. *IEEE Trans. Softw. Eng.*, 39(8):1069–1089, August 2013.

[CD11] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of c programs. In *NASA Formal Methods Symposium*, pages 459–465. Springer, 2011.

[CDG93] Michael Codish, Saumya K. Debray, and Roberto Giacobazzi. Compositional analysis of modular logic programs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 451–464, New York, NY, USA, 1993. ACM.

[CDOY09] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 289–300, New York, NY, USA, 2009. ACM.

[CEW12] Sheng Chen, Martin Erwig, and Eric Walkingshaw. An error-tolerant type system for variational lambda calculus. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 29–40, New York, NY, USA, 2012. Association for Computing Machinery.

[CG94] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in c++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 397–408, New York, NY, USA, 1994. ACM.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, page 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[CKLS09] Cristina Cifuentes, Nathan Keynes, Lian Li, and Bernhard Scholz. Program analysis for bug detection using parfait: Invited talk. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '09, page 7–8, New York, NY, USA, 2009. Association for Computing Machinery.

[Cla18a] Clang: a c language family frontend for llvm, July 2018. `http://clang.llvm.org/`.

[Cla18b] Clang static analyzer, August 2018. `https://clang-analyzer.llvm.org/`.

[Cla18c] Clang-tidy, August 2018. `http://clang.llvm.org/extra/clang-tidy/`.

[CNDE05] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 449–461, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Cod17] CodeMonkey. Clang static analyzer doesn't find the most basic problems, March 2017. `https://stackoverflow.com/questions/42696759/clang-static-analyzer-doesnt-find-the-most-basic-problems`.

[Cod18] Grammatech codesonar, December 2018. `https://www.grammatech.com/products/codesonar`.

[Cor18] Coreutils - gnu core utilities, July 2018. `https://www.gnu.org/software/coreutils/coreutils.html`.

[Cpp18] Cppcheck - a tool for static c/c++ code analysis, August 2018. `http://cppcheck.sourceforge.net/`.

[CS18] Coverity-(SAST). Coverity static application security testing (sast), December 2018.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, jul 2019.

[DH96] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 306–323, New York, NY, USA, 1996. Association for Computing Machinery.

[Dim16] Aleksandar S. Dimovski. Symbolic game semantics for model checking program families. In Dragan Bošnački and Anton Wijs, editors, *Model Checking Software*, pages 19–37, Cham, 2016. Springer International Publishing.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Doo18] Doop, August 2018. `http://doop.program-analysis.org/`.

[Dri10] Vincent Driessen. A successful git branching model, January 2010. `https://nvie.com/posts/a-successful-git-branching-model/`.

[Dro19] Droidbench, April 2019. `https://github.com/secure-software-engineering/DroidBench`.

[Dwy97] M. B. Dwyer. Modular flow analysis for concurrent software. In *Proceedings of the 12th International Conference on Automated Software Engineering (Formerly: KBSE)*, ASE '97, pages 264–, Washington, DC, USA, 1997. IEEE Computer Society.

[EBN02] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of c preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[EH14] Michael Eichberg and Ben Hermann. A software product line for static analyses: The opal framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, pages 1–6, New York, NY, USA, 2014. ACM.

[EHMG15] Michael Eichberg, Ben Hermann, Mira Mezini, and Leonid Glanz. Hidden truths in dead software paths. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 474–484, New York, NY, USA, 2015. ACM.

[EHRS00] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In *International Conference on Computer Aided Verification*, pages 232–247. Springer, 2000.

[EL16] Jon Eyolfson and Patrick Lam. C++ const and Immutability: An Empirical Study of Writes-Through-const. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[FHJ+06] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Goanna—a static model checker. In *International Workshop on Parallel and Distributed Methods in Verification*, pages 297–300. Springer, 2006.

[GCC18a] Gcc, the gnu compiler collection, July 2018. `https://gcc.gnu.org/`.

[gcc18b] Gnu compiler collection (gcc) internals, July 2018. `https://gcc.gnu.org/onlinedocs/gccint/`.

[GG12] Paul Gazzillo and Robert Grimm. Superc: parsing all of C by taming the preprocessor. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 323–334. ACM, 2012.

[Git19] git –fast-version-control, April 2019. `https://git-scm.com/`.

[GJ05] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a c program. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, page 379–388, USA, 2005. IEEE Computer Society.

[GLL21] Gllvm: Whole program llvm in go, March 2021. `https://github.com/SRI-CSL/gllvm`.

[GPMP20] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[Gra19] Graphviz, August 2019. `https://www.graphviz.org/`.

[GRS00] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 334–344, New York, NY, USA, 2000. ACM.

[Gur23] Refactoring Guru. Mediator pattern, 2023. `https://refactoring.guru/design-patterns/mediator`.

[Har23] Ajay Harish. When nasa lost a spacecraft due to a metric math mistake, November 2023. `https://www.simscale.com/blog/2017/12/nasa-mars-climate-orbiter-metric/`.

[Hea14] HeartBleedBug. Heartbleed: a serious vulnerability in the popular openssl cryptographic software library. `https://heartbleed.com/`, 2014.

[Her20] Hercules, June 2020. `https://github.com/joliebig/Hercules`.

[Hg19] Mercurial, April 2019. `https://www.mercurial-scm.org/`.

[HHL+17] P. Holzinger, B. Hermann, J. Lerch, E. Bodden, and M. Mezini. Hardening java's access control by abolishing implicit privilege elevation. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 1027–1040, May 2017.

[HKR+20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. Modular collaborative program analysis in opal. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 184–196, New York, NY, USA, 2020. Association for Computing Machinery.

[HL08] Lile P. Hattori and Michele Lanza. On the nature of commits. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*, pages 63–71, 2008.

[HO18] Mark Harman and Peter O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, 2018.

[HP00] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, page 113–123, New York, NY, USA, 2000. Association for Computing Machinery.

[HR96] Mary Jean Harrold and Gregg Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Softw. Eng.*, 22(7):442–460, July 1996.

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

[ICC18] ICCOptimizeOptions. Intel® c++ compiler 19.0 developer guide and reference: Interprocedural optimization (ipo), December 2018.

[IMD+17] Alexandru Florin Iosif-Lazar, Jean Melo, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Effective analysis of C programs by rewriting variability. *CoRR*, abs/1701.08114, 2017.

[iOS15] iOSUnicodeBug. Bug in ios unicode handling crashes iphones with a simple text. `https://appleinsider.com/articles/15/05/26/bug-in-ios-notifications-handling-crashes-iphones-with-a-simple-text/`, 2015.

[Jav18] Java virtual machine specification: The constant pool, December 2018. `https://docs.oracle.com/javase/specs/jvms/se10/html/jvms-4.html#jvms-4.4`.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., USA, 1993.

[JKC20] Swati Jaiswal, Uday P. Khedker, and Supratik Chakraborty. Bidirectionality in flow-sensitive demand-driven analysis. *Science of Computer Programming*, 190:102391, 2020.

[JSS16] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification*, pages 422–430. Springer, 2016.

[Käs10] Christian Kästner. *Virtual separation of concerns: toward preprocessors 2.0*. PhD thesis, Otto von Guericke University Magdeburg, 2010.

[KATS12] Christian Kästner, Sven Apel, Thomas Thüm, and Gunter Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol.*, 21(3), July 2012.

[KGR+11] 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 *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, page 805–824, New York, NY, USA, 2011. Association for Computing Machinery.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[KKHL10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. Typechef: Toward type checking #ifdef variability in c. In *Proceedings of the 2Nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 25–32, New York, NY, USA, 2010. ACM.

[Klo08] Karsten Klohs. A summary function model for the validation of interprocedural analysis results. In *Proceedings of the 7th International Workshop on Compiler Optimization meets Compiler Verification, COCV'08*, 2008.

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

[KOE12] Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, page 773–792, New York, NY, USA, 2012. Association for Computing Machinery.

[KSK09] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., USA, 1st edition, 2009.

[KTS+09] Christian Kastner, Thomas Thum, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. Featureide: A tool framework for feature-oriented software development. In *2009 IEEE 31st International Conference on Software Engineering*, pages 611–614. IEEE, 2009.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, Sep 1977.

[LA04] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[LBLH11] Patrick Lam, Eric Bodden, Ondřej Lhoták, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Cetus Users and Compiler Infrastructure Workshop (CETUS 2011)*, October 2011.

[LH15] Johannes Lerch and Ben Hermann. Design your analysis: A case study on implementation reusability of data-flow functions. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2015, pages 26–30, New York, NY, USA, 2015. ACM.

[LHBM14] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. Flowtwist: Efficient context-sensitive inside-out taint analysis for large codebases. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 98–108, New York, NY, USA, 2014. Association for Computing Machinery.

[LHR19] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. Rethinking incremental and parallel pointer analysis. *ACM Trans. Program. Lang. Syst.*, 41(1):6:1–6:31, March 2019.

[LKA11] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of c code. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, AOSD '11, page 191–202, New York, NY, USA, 2011. Association for Computing Machinery.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[LLV19] Llvm git commit activity, May 2019. `https://github.com/llvm/llvm-project/graphs/commit-activity`.

[LP14] Wei Le and Shannon D. Pattison. Patch verification via multiversion interprocedural control flow graphs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 1047–1058, New York, NY, USA, 2014. Association for Computing Machinery.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[LSXX13] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 61–81, Berlin, Heidelberg, 2013. Springer-Verlag.

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 282–298, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[LTMS18] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 129–140, New York, NY, USA, 2018. ACM.

[LvRK+13] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 81–91. ACM, 2013.

[LWWX16] Hongliang Liang, Lei Wang, Dongyang Wu, and Jiuyun Xu. Mlsa: a static bugs analysis tool based on llvm ir. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 407–412. IEEE, 2016.

[MB05] Bill McCloskey and Eric Brewer. Astec: A new approach to refactoring c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, page 21–30, New York, NY, USA, 2005. Association for Computing Machinery.

[MDBW15] Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wasowski. Systematic derivation of correct variability-aware program analyses. *Sci. Comput. Program.*, 105:145–170, 2015.

[mei22] meijer. Researchers introducing [linux] kernel bugs on purpose, November 2022. `https://www.reddit.com/r/programming/comments/mvek9m/researches_introducing_linux_kernel_bugs_on/`.

[Mer03] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *in Proc. GCC Developers Summit, 2003*, pages 171–180, 2003.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005.

[Mey14] Scott Meyers. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[MKK07] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[MLI23] Multi-level ir compiler framework, January 2023.

[MR14] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering*, pages 125–139, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[NKH+18] Lisa Nguyen, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. Visuflow, a debugging environment for static analyses. In *International Conference for Software Engineering (ICSE), Tool Demonstrations Track*, 1 January 2018.

[NLR10] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavy-weight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[Onl18] GCC Onlinedocs. Options that control optimization, December 2018. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

[Onl21] GCC Onlinedocs. Cpp 3.4 stringizing, November 2021. `https://gcc.gnu.org/onlinedocs/gcc-11.2.0/cpp/Stringizing.html#Stringizing`.

[Ope08] OpenSSLRandomNumberGeneratorBug. Dsa-1571-1 openssl – predictable random number generator. `https://www.debian.org/security/2008/dsa-1571/`, 2008.

[OPS92] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In Ole Lehrmann Madsen, editor, *ECOOP '92 European Conference on Object-Oriented Programming*, pages 329–349, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[PDB19] Goran Piskachev, Lisa Nguyen Quang Do, and Eric Bodden. Codebase-adaptive detection of security-relevant methods. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*,

ISSTA 2019, page 181–191, New York, NY, USA, 2019. Association for Computing Machinery.

[Pha18] Phasar, July 2018. `https://phasar.org`.

[PK13] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in soot using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.

[Pro12] Emil Protalinski. Apple security blunder exposes lion login passwords in clear text, May 2012. `https://www.zdnet.com/article/apple-security-blunder-exposes-lion-login-passwords-in-clear-text/`.

[Pro18] The programming languages beacon, December 2018. `https://www.mentofacturing.com/vincent/implementations.html`.

[RAB14] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, volume 14, page 1125, 2014.

[Ray14] Santanu Saha Ray. *Graph Theory with Algorithms and Its Applications: In Applied Science and Technology*. Springer Publishing Company, Incorporated, 2014.

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

[Rep00] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, jan 2000.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society, 1953, 74, 2, 358*, 1953.

[RLJ+18] Alexander Von Rhein, JöRG Liebig, Andreas Janker, Christian Kästner, and Sven Apel. Variability-aware static analysis at scale: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 27(4), November 2018.

[RMR04] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, June 2004.

[Rou14] Personal communication with atanas (nasko) rountev, 2014.

[RR01] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 20–36, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering — ESEC/FSE '99*, pages 235–252, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[RSJ03] Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted pushdown systems and their application to interprocedural dataflow analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 189–213, Berlin, Heidelberg, 2003. Springer-Verlag.

[RSX08] Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 53–68, Berlin, Heidelberg, 2008. Springer-Verlag.

[Ryd83] Barbara G. Ryder. Incremental data flow analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 167–176, New York, NY, USA, 1983. ACM.

[SAB10] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.

[SAB17] Johannes Späth, Karim Ali, and Eric Bodden. Ideal: Efficient and precise alias-aware dataflow analysis. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[SAB19a] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[SAB19b] Johannes Späth, Karim Ali, and Eric Bodden. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proc. ACM Program. Lang.*, 3(POPL):48:1–48:29, January 2019.

[SAE+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018.

[SAIM08] Ozgur Sumer, Umut Acar, Alexander T. Ihler, and Ramgopal R. Mettu. Efficient bayesian inference for dynamically changing graphs. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1441–1448. Curran Associates, Inc., 2008.

[Sat23] Florian Sattler. Vara: a variational region analyzer, 2023. `https://github.com/se-sic/vara-llvm-project/`.

[SB09] Asia Slowinska and Herbert Bos. Pointless tainting: Evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 61–74, New York, NY, USA, 2009. ACM.

[SBEV18] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing lattice-based program analyses in datalog. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[SBL11] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[SBS+23] Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. Seal: Integrating program analysis and repository mining. *ACM Trans. Softw. Eng. Methodol.*, 32(5), July 2023.

[SCS24] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive abstract interpretation with demanded summarization. *ACM Trans. Program. Lang. Syst.*, 46(1), March 2024.

[Sec19] Securibench, April 2019. `https://suif.stanford.edu/~livshits/work/securibench/intro.html`.

[SEV16] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: A dsl for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 320–331, New York, NY, USA, 2016. Association for Computing Machinery.

[SGP+22] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. Static data-flow analysis for software product lines in c. *Automated Software Engineering*, 29(1):35, Mar 2022.

[SHB19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Phasar: An inter-procedural static analysis framework for c/c++. In Tomáš Vojnar and

Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 393–410, Cham, 2019. Springer International Publishing.

[SHB21] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:31, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[SKB14] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM.

[SLHB19] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Know your analysis: How instrumentation aids understanding static analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2019, pages 8–13, New York, NY, USA, 2019. ACM.

[SLHB21] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Into the woods: Experiences from building a dataflow analysis framework for c/c++. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 18–23, 2021.

[SLL02] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library - User Guide and Reference Manual*. C++ in-depth series. Pearson / Prentice Hall, 2002.

[SMM15] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. Test suites for benchmarks of static analysis tools. In *Proceedings of the 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, ISSREW '15, pages 12–15, Washington, DC, USA, 2015. IEEE Computer Society.

[SNAB16] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for java. In *European Conference on Object-Oriented Programming (ECOOP)*, 17 - 22 July 2016.

[Sof18] Black Duck Software. 2018 open source security and risk analysis. `https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2018-ossra.pdf`, 2018.

[Son23a] SonarCloud. clean code in your cloud workflow with sonarcloud, November 2023. `https://www.sonarsource.com/products/sonarcloud/`.

[son23b] sonarqube. clean code for teams and enterprises with sonarqube, November 2023. `https://www.sonarsource.com/products/sonarqube/`.

[SP78] M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., New York, NY, 1978.

[SRH96] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, October 1996.

[SSS+21] Philipp Dominik Schubert, Florian Sattler, Fabian Schiebel, Ben Hermann, and Eric Bodden. Modeling the effects of global variables in data-flow analysis for c/c++. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 12–17, 2021.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[Str83] Robert E. Strom. Mechanisms for compile-time enforcement of security. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 276–284, New York, NY, USA, 1983. ACM.

[Str18] C++ applications, December 2018. `https://www.stroustrup.com/applications.html`.

[SVE17] Tamas Szabo, Markus Volter, and Sebastian Erdweg. Incal: A dsl for incremental program analysis with lattices. In *International Workshop on Incremental Computing (IC), 2017.*, 2017.

[SvGJ+15] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, page 598–608. IEEE Press, 2015.

[SVN19] Subversion, April 2019. `https://subversion.apache.org/`.

[Swa76] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

[SX16a] Yulei Sui and Jingling Xue. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 460–473, New York, NY, USA, 2016. ACM.

[SX16b] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM.

[SXX12] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 264–274, New York, NY, USA, 2012. ACM.

[SY86] R E Strom and S Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, January 1986.

[Syn18] Synk. The state of open source security, December 2018. `https://snyk.io/stateofossecurity/`.

[Sza21] Tamás Szabó. *Incrementalizing Static Analyses in Datalog*. PhD thesis, University of Mainz, Germany, 2021.

[TA12] Thomas Thüm and Sven Apel. Analysis strategies for software product lines. *none*, 2012. `https://www.cs.cmu.edu/~ckaestne/pdf/tr_analysis12.pdf`.

[TG17] John Toman and Dan Grossman. Taming the Static Analysis Beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, volume 71 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18:1–18:14, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Thi18] Thinlto: Scalable and incremental lto, July 2018. `http://blog.llvm.org/2016/06/thinlto-scalable-and-incremental-lto.html`.

[TL:18a] Llvm language reference manual, July 2018. `http://llvm.org/docs/LangRef.html`.

[TL:18b] Llvm users, July 2018. `http://llvm.org/Users.html`.

[Tra19] Gregory Travis. How the boeing 737 max disaster looks to a software developer, April 2019. `https://spectrum.ieee.org/how-the-boeing-737-max-disaster-looks-to-a-software-developer`.

[Tri81] Eric L Trist. *The evolution of socio-technical systems*, volume 2. Ontario Quality of Working Life Centre Toronto, 1981.

[TWX+17] Hao Tang, Di Wang, Yingfei Xiong, Lingming Zhang, Xiaoyin Wang, and Lu Zhang. Conditional dyck-cfl reachability analysis for complete and efficient library summarization. In Hongseok Yang, editor, *Programming Languages*

and Systems, pages 880–908, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[TWZ+15] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 83–95, New York, NY, USA, 2015. ACM.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[Wal19] Wala, April 2019. `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[Win17] Titus Winters. C++ as "live at head" language. `https://www.youtube.com/watch?v=tISy7EJQPzI&ab_channel=CppCon`, 2017.

[WKE+14] Eric Walkingshaw, Christian Kästner, Martin Erwig, Sven Apel, and Eric Bodden. Variational data structures: Exploring tradeoffs in computing with variability. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SPLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 213–226. ACM, 2014.

[WLL21] Wllvm: Whole program llvm, March 2021. `https://github.com/travitch/whole-program-llvm`.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, page 187–206, New York, NY, USA, 1999. Association for Computing Machinery.

[WS73] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2):28–34, February 1973.

[WSR00] Reinhard Wilhelm, Shmuel Sagiv, and Thomas W. Reps. Shape analysis. In *Proceedings of the 9th International Conference on Compiler Construction*, CC '00, pages 1–17, London, UK, UK, 2000. Springer-Verlag.

[XHN05] Jingling Xue and Phung Hua Nguyen. Completeness analysis for incomplete object-oriented programs. volume 3443, pages 271–286, 04 2005.

[YMX+10] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 143–154, New York, NY, USA, 2010. ACM.