BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

STATIC ANALYSIS OF C PROGRAMS

STATICKÁ ANALÝZA PROGRAMŮ V JAZYCE C

PHD THESIS
DISERTAČNÍ PRÁCE

AUTHOR **Ing. VIKTOR MALÍK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL

BRNO 2023

Abstract

This thesis proposes several original contributions to the area of static analysis of software with focus on low-level systems code written in C. The contributions are split into two parts, each related to a different area of static analysis, namely formal verification of (low-level) C code and static analysis of semantic equivalence of different versions of the same software.

The first part proposes new analyses suitable for verification engines that perform automatic invariant inference using an SMT solver. The proposed solution includes two abstract template domains that use logical formulae over bit-vectors to encode the shape of the program heap and the contents of the program arrays. The shape domain is based on computing a points-to relation between pointers and symbolic addresses of abstract memory objects. The array domain is based on splitting the arrays into several non-overlapping contiguous segments and computing a different invariant for each of them. Both domains can be combined with value domains in a straightforward manner, which particularly allows our approach to reason about shapes and contents of heap and array structures at the same time. The information obtained from the analyses can be used to prove memory safety and reachability properties, expressed by user assertions, of programs manipulating data structures. All of the proposed solutions have been implemented in the 2LS framework and compared against state-of-the-art tools that perform the best in the relevant categories of the well-known Software Verification Competition (SV-COMP). Results show that 2LS outperforms these tools on benchmarks requiring combined reasoning about unbounded data structures and their numerical contents.

The second part of the thesis is motivated by existence of software projects that undergo regular refactorings and modifications and yet need to ensure semantic stability of some of their core parts. This part proposes a highly-scalable approach for automatically checking semantic equivalence of different versions of large, real-world C projects, with a particular focus on the Linux kernel. The proposed method uses a novel combination of pattern matching with light-weight static analysis and control-flow transformations. The method checks preservation of the semantics of functions forming the API of the project being analyzed as well as of the semantics of its global variables, which typically hold various control parameters. For the latter, a specialised slicing procedure is proposed to slice out code influenced by these variables and concentrate the further analysis on that code only. Although the method cannot prove equivalence on heavily refactored code, it can compare thousands of functions in the order of minutes while producing a low number of false non-equality verdicts as our experiments show. The method has been implemented over the LLVM infrastructure in a tool called DiffKemp. Our results show that DiffKemp, unlike other existing tools, gives practically useful results even on projects of the size of the Linux kernel.

Keywords

static analysis, formal verification, formal methods, shape analysis, array abstract domain, template-based invariant synthesis, abstract interpretation, abstract domains, SAT/SMT solving, loop invariants semantic equivalence, refactoring, pattern matching, semantics-preserving patterns, code change patterns, refactoring patterns, Linux kernel, program slicing, LLVM IR

Abstrakt

Táto práca prináša niekoľko originálnych príspevkov do oblasti statickej analýzy programov so zameraním na nízkoúrovňový softvér napísaný v jazyku C. Práca je rozdelená do dvoch častí, z ktorých každá sa venuje inej oblasti statickej analýzy, konkrétne formálnej verifikácii a statickej analýze sémantickej ekvivalencie rôznych verzií softvéru.

V prvej časti práce navrhujeme nové analýzy vhodné pre verifikačné nástroje založené na automatickom odvodzovaní invariantov s využitím SMT solveru. Navrhnuté riešenie zahŕňa dve nové abstraktné domény založené na šablónach, ktoré umožňujú popísať tvar programovej haldy a obsahy polí pomocou logických formulí nad bit-vektormi. Doména pre reprezentáciu tvaru haldy je založená na zachytení vzťahov medzi ukazovateľmi a symbolickými adresami abstraktných objektov v pamäti. Doména pre reprezentáciu obsahov polí je založená na rozdelení polí na niekoľko neprekrývajúcich sa spojitých segmentov a odvodení samostatného invariantu pre každý segment. Obidve domény sú navrhnuté spôsobom, ktorý umožňuje ich kombináciu s inými doménami, vďaka čomu je možné abstrahovať tvar a obsah dátových štruktúr zároveň. Informácie získané z týchto analýz je možné použiť na dokázanie bezpečnosti práce s pamäťou a nedosiahnuteľnosti chybových stavov v programoch, ktoré pracujú s dynamickými dátovými štruktúrami. Všetky navrhnuté rozšírenia boli implementované do nástroja 2LS a porovnané s nástrojmi, ktoré sa pravidelne umiestňujú na najvyšších priečkach v relevantných kategóriách v medzinárodnej súťaži vo verifikácii software SV-COMP. Výsledky experimentov ukazujú, že 2LS poráža tieto nástroje na úlohách vyžadujúcich invarianty cyklov kombinujúce popis tvaru a obsahu dynamických dátových štruktúr.

Druhá časť práce je motivovaná existenciou softvérových projektov, ktoré vyžadujú zachovanie stability niektorých dôležitých častí, no zároveň sú podrobované pravidelným zmenám. V rámci tejto časti navrhujeme nový prístup pre automatickú analýzu sémantickej ekvivalencie rôznych verzí veľkého priemyselného softvéru napísaného v jazyku C, so špeciálnym zameraním na jadro operačného systému Linux. Navrhnutá metóda používa unikátnu kombináciu vyhľadávania vzorov, rýchlej statickej analýzy a transformácií toku riadenia programov. Tento prístup umožňuje kontrolu zachovania sémantiky funkcií, ktoré tvoria rozhranie analyzovaného projektu ako aj globálnych premenných, ktoré typicky reprezentujú hodnoty konfigurovateľných parametrov systému. Pre porovnávanie globálnych premenných zároveň navrhnujeme špecializovanú procedúru pre prerezávanie programov, ktorá je schopná odstrániť časti programov, ktoré nie sú závislé na hodnotách analyzovaných premenných a obmedziť analýzu iba na závislé časti. Napriek tomu, že metóda nie je schopná formálne dokázať sémantickú ekvivalenciu zásadne upraveného, no ekvivalentného kódu, je schopná porovnať tisíce funkcií v rámci minút a zároveň poskytnúť relatívne malé množstvo nesprávnych výsledkov. Metóda bola implementovaná v nástroji DiffKemp nad infraštruktúrou LLVM. Výsledky experimentov ukazujú, že DiffKemp, narozdiel od iných nástrojov v oblasti, dáva prakticky použiteľné výsledky aj na projekte o veľkosti jadra Linuxu.

Kľúčové slová

statická analýza, formálna verifikácia, formálne metódy, analýza tvaru haldy, abstraktná doména pre popis polí, syntéza invariantov založená na šablónach, abstraktná interpretácia, SAT/SMT solving, invarianty cyklov sémantická ekvivalencia programov, refaktoring, vyhľadávanie vzorov, vzory zachovávajúce sémantiku, vzory zmien v programoch, jadro Linuxu, prerezávanie programov, LLVM IR

Static Analysis of C Programs

Declaration

I hereby declare that this PhD thesis was prepared as an original work by the author under the supervision of prof. Tomáš Vojnar. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Viktor Malík
September 14, 2023

Acknowledgements

There is a lot of people whom I would like to thank and who made this work possible. First of all, big thanks goes to my family—my parents, my brother, and especially my wife. She has supported me throughout my entire PhD study and she has always been patient with me, especially when I was finishing papers over those sunny weekends that I'm sure she imagined we would have spent elsewhere. Another great thanks goes to my supervisor, Tomáš Vojnar, who was always very supportive and contributed a great deal to the ideas introduced in this thesis. Last, but not least, my colleagues and co-workers—Peter Schrammel from DiffBlue who gave a lot of insights into 2LS and greatly helped with the papers, Jan Zelený from Red Hat who made the very existence of DiffKemp possible, and all people from the VeriFIT group at FIT BUT and at Red Hat whom I had the opportunity to meet and work with. All these people have been a great source of inspiration and without them, this thesis would never exist.

Reference

MALÍK, Viktor. *Static Analysis of C Programs.* Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor prof. Ing. Tomáš Vojnar, Ph.D.

Contents

1 Introduction **4**
1.1 Verification of Programs with Data Structures 5
1.2 Static Analysis of Semantic Equivalence for C Projects 7

I Template-Based Verification of Programs with Data Structures **9**

2 Template-Based Verification of Software **10**
2.1 Internal Program Representation 11
2.1.1 The Static Single Assignment Form 12
2.2 k-induction and k-invariants 13
2.2.1 Template-Based Predicate Inference 15
2.2.2 Abstract Domains in 2LS 18
2.2.3 Incremental Bounded Model Checking 19
2.2.4 Incremental k-Induction 19
2.3 Implementation and Architecture of the 2LS Framework 20
2.4 Running Example 22

3 Verification of Heap-Manipulating Programs **24**
3.1 Memory Model 25
3.1.1 Static Memory Objects 25
3.1.2 Dynamic Memory Objects 25
3.2 Representation of Memory-Manipulating Operations 27
3.2.1 Dynamic Memory Allocation 27
3.2.2 Reading through Dereferenced Pointers 27
3.2.3 Writing through Dereferenced Pointers 28
3.3 Abstract Domain for Heap Shape Analysis 29
3.4 Abstract Domain Combinations 30
3.4.1 Product Templates 31
3.4.2 Templates with Symbolic Paths 31
3.5 Memory Safety Verification 32
3.5.1 Safety from Dereferencing/Freeing a `null` Pointer 33
3.5.2 Safety from Dereferencing/Freeing a Freed Pointer 33
3.5.3 Memory Leaks Safety 33
3.6 Related Work 34
3.6.1 Logic-based Methods 34
3.6.2 Methods Using Automata and Graphs 35
3.6.3 Methods Using Storeless Semantics 35

4 Verification of Array-Manipulating Programs **37**
4.1 Array Domain Template . . . 38
4.2 Computing Array Segment Borders . . . 39
4.3 Array Domain Invariant Inference . . . 40
4.4 Running Example . . . 41
4.5 Related Work . . . 42
4.5.1 Methods Based on Array Segmentation . . . 42
4.5.2 Methods Based on Analysis of Array-Manipulating Loops . . . 43
4.5.3 Predicate Abstraction and Non-Automatic Methods . . . 43

5 Experimental Evaluations **44**
5.1 SV-COMP Organization and Rules . . . 44
5.2 Scores of 2LS in SV-COMP . . . 45
5.2.1 Heap-related Categories . . . 46
5.2.2 Array-related Categories . . . 46
5.2.3 Comparison to Other Tools . . . 47
5.3 Alternative Rankings . . . 47
5.3.1 Speed of Verification in 2LS . . . 47
5.3.2 Power Consumption and Correctness Rate . . . 48

II Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects **49**

6 Static Analysis of Semantic Equivalence **50**
6.1 Program Representation . . . 52
6.2 Function Equality . . . 53
6.3 Analysis of Function Equality . . . 54

7 Built-in Semantics-Preserving Change Patterns **58**
7.1 Supported Semantics-Preserving Changes . . . 59
7.1.1 Change Patterns in the Linux Kernel . . . 59
7.2 Handling the Supported SPCPs . . . 61
7.2.1 Changes in Structure Data Types . . . 61
7.2.2 Moving Code into Functions . . . 64
7.2.3 Changes in Enumeration Values . . . 65
7.2.4 Changes in Source Code Location . . . 65
7.2.5 Inverse Branch Conditions . . . 66
7.2.6 Code Relocations . . . 67

8 Custom Change Patterns **70**
8.1 Representation of Custom Change Patterns . . . 70
8.1.1 Formal Definition of Custom Change Patterns . . . 70
8.1.2 Encoding Change Patterns with LLVM IR . . . 71
8.2 Custom Change Pattern Matching . . . 71
8.2.1 Pattern Detection . . . 72
8.2.2 Determining Successor Synchronisation Points . . . 75
8.2.3 Semantic Equality Detection . . . 75
8.2.4 Updating the Variable Mapping . . . 76

9 Comparing the Use of Global Variables **77**
9.1 Comparing Functions w.r.t. a Variable 78
9.2 Slicing Algorithm 78

10 Implementation and Experiments **81**
10.1 KABI Functions 81
10.2 Refactoring Commits in the Linux Kernel 82
10.3 The *musl* Standard C Library 83
10.4 A Comparison with Other Tools 84
10.5 Effectiveness of Program Slicing 85
10.6 Application of Custom Change Patterns 85
10.6.1 The List of Used Patterns 86
10.7 Efficiency of CCP Matching 87

11 Related Work **89**

12 Conclusion **92**

Bibliography **94**

Chapter 1

Introduction

Computer science and software engineering count among the youngest fields of human endeavour, yet they already influence practically every aspect of our lives. According to some reports [68], there were 33.6 million active software developers in the world in 2022. With every developer being able to produce thousands of lines of code per year, software grows in size and complexity in a tremendous pace and often to an extent beyond understanding of the authors themselves. With that, any useful technique and tool that helps developers understand and maintain their programs is extremely valuable.

One of such techniques is *automatic static program analysis*, usually simply referred to as *static analysis*. Generally, it is a category of algorithms designed to automatically extract information from a program's source code, without actually executing the program (at least not under its original semantics)[117].

Static analysis is an opposed approach to dynamic analysis (which also includes, e.g., testing), which gathers facts about the program during its execution. Static analysis approaches have several advantages over the dynamic ones. Arguably, the most important one is the fact that static analysers can reason about all possible branches of the program while dynamic approaches can only access the program path that is executed. On the other hand, the very same fact is a major disadvantage—while dynamic analysers have precise information about the runtime of the program, such as the state and the layout of the memory, static analysis methods must often deduce those, which creates space for errors.

There are various categories of static analysis algorithms and tools, depending on the kind of information they strive to extract from the code. One of the most common use-cases of static analysis is for discovering bugs, where the code is scanned for commonly known errors and vulnerabilities. This category also features the widest portfolio of techniques, varying from simple tools for detecting bad programming patterns that often lead to bugs (so-called "code smells") to very complex systems capable not only to discover bugs but also to prove their absence. The latter are usually based on formal methods and consequently the group of techniques is denoted as *formal verification*.

Besides bug finding and formal verification focused on so-called safety properties (i.e., discovering that the program "does not do anything bad"), static analysis can be used to discover other properties of software. For instance, some formal verification tools are able to determine if the program terminates or not or there exist specialized approaches to deduce amortized program complexity. Another property often tackled with static analysis is semantic equivalence of programs. Tools from this category are able to determine if two programs, despite being written in a different manner, have the same semantics. Such

information can be valuable for multiple reasons—it can be used to prevent code duplication, to detect stability breakage, or even to discover introduction of hidden bugs.

Even though static analysis techniques have a great potential of use, they naturally come with a number of drawbacks. Perhaps one of the most important ones is their limited applicability within real-world software development, which has several causes. First, many techniques, especially those based on formal methods, are too complicated and have problems to scale on huge and complex contemporary systems. Second, due to their over-approximative nature, many static analysis algorithms produce false defect reports, so-called "false positives". If the number of such results is high, users tend to start ignoring all the issues reported by the tool, effectively mitigating the purpose of usage of such a tool. Last but not least, a number of static analysis tools is rather narrowly focused and therefore can be applied to a certain part or property of software only. This often creates a necessity to use a combination or a portfolio of different techniques, which poses a technical and maintenance burden on the users.

In this thesis, we introduce several novel techniques that approach the above issues from various perspectives. Our interest lies in static analysis of low-level software, usually written in C, therefore we mostly focus on program features that typically occur in such programs. In particular, we propose (1) a technique for formal verification of programs manipulating low-level data structures such as arrays and linked lists which we integrate into an existing verification framework and (2) a novel method for light-weight static analysis of semantic differences between versions of a project that is applicable to system code containing hundreds of thousands of lines of code. Even though these two areas share certain aspects, they are mostly distinct and therefore the thesis is split into two parts, each dedicated to one of the above areas.

In the rest of this chapter, we briefly introduce each of our contributions. For a more detailed general description, refer to the introduction of the first chapter of each part.

1.1 Verification of Programs with Data Structures

In the first part of this thesis, we focus on the area of formal verification of software. One of the most challenging verification tasks is verification of programs dealing with dynamically allocated data structures. There are multiple reasons for this: (1) the program heap is practically unbounded, (2) the structures are allocated during program execution, hence their shape is unknown in advance, and (3) the shape of the structures can be rather complex. To deal with these challenges, verifiers typically introduce some kind of abstraction to describe the program memory. A common problem of many of the existing approaches is that they often struggle to combine their data structure abstractions with abstractions of other parts of the programs such as values of numerical variables, contents of arrays, etc., which makes verification of some kinds of programs (e.g., programs using a linked list whose nodes contain arrays of pointers to integers) impossible.

In this thesis, we propose two new abstractions of data structures that allow to reason about (1) pointer-based structures on the heap such as various linked lists and (2) contents of arrays. To address the mentioned limitations of static analysis, we integrate our contributions into a verification framework called 2LS which features a rather specific verification approach allowing a seamless combination of various program abstractions. To be able to achieve this, our proposed abstractions need to comply with the form required by the framework.

The central idea of 2LS is that it combines several verification techniques together—namely abstract interpretation, k-induction, and bounded model checking—in an elegant algorithm. To allow for a seamless combination, 2LS translates the verified programs and properties into first-order formulae and uses an off-the-shelf SMT solver to reason about them. To exploit the true power of the algorithm, k-inductive invariants for program loops must be computed. The problem of finding invariants is a second-order logic problem (not efficiently solvable with the current solvers), hence 2LS reduces it to a first-order logic by requiring invariants to be instances of fixed, parametrized, first-order formulae, so-called *templates* (hence the approach is denoted as *template-based verification*).

The data structure abstractions that we propose must hence follow this requirement, which comes with several challenges, especially due to the fact that we need to represent potentially unbounded structures using a fixed first-order formula. On the other hand, our abstractions can be easily combined with each other and with other abstractions present in 2LS, which allows the verifier to reason about, e.g., numerical contents of linked lists or pointers stored inside arrays.

Our abstraction of the shape of the program heap is based on tracking the points-to relation between pointers and memory objects. Since the amount of memory objects is possibly unbounded, as a part of our work, we developed a new memory model in which we describe the entire memory using a finite number of so-called abstract dynamic objects. This model allows us to encode typical memory-manipulating operations (allocation and pointer (de)referencing) into first-order logic formulae. As for the abstraction of arrays, the central idea is to split each array into multiple contiguous, non-overlapping segments and to compute a different invariant for each of them.

Overall, we may summarize the contributions of this thesis part as follows:

1. We introduce a new memory model and an encoding of memory-manipulating operations into first-order logic which are suitable for the verification approach of the 2LS framework.

2. We propose a novel abstract template domain for reasoning over heap-allocated data structures such as singly and doubly linked lists using a template-based parameter synthesis engine.

3. We propose a novel abstract template domain for reasoning about the structure of arrays which is able to use an arbitrary abstract domain to describe values stored inside the arrays.

4. We show how we can build product and power domain combinations of our heap and array domains with structural domains (e.g., trace partitioning) and value domains such as template polyhedra that capture the content of data structures.

5. We implement our abstract domains in the 2LS verification tool for C programs. We demonstrate the power of the proposed domains on benchmarks which require combined reasoning about the shape and content of data structures.

The contributions in this part are based on our paper published at FMCAD 2018 [85], which introduced the shape domain, and several competition contribution papers from SV-COMP published at TACAS 2018, 2020, and 2023 [87, 90, 89]. The text is also heavily inspired by the chapter on 2LS in the *Automated Software Verification* book, which is to be published in 2023/24 (the final submitted version is available from arXiv [69]).

1.2 Static Analysis of Semantic Equivalence for C Projects

In the second part of the thesis, we move from formal verification to the area of automatic static analysis of semantic equivalence of programs. Similarly to verification of other properties, many tools in this area are based on formal methods, inherently sharing the same drawbacks, particularly poor scalability on large-scale software. On the other hand, there exist very lightweight tools for comparing programs based on simple text or abstract syntax tree similarity (let us name the Unix `diff` tool from all examples) which can analyse huge amount of code in a short time but they compare for syntactic similarity rather than the semantic one. To the best of our knowledge, until our work in this area there was no approach that would be able to statically analyse semantic equivalence of programs containing hundreds of thousands of lines of code in a reasonable amount of time. Such a tool would have a lot of potential applications, for example checking semantic stability and backwards compatibility of critical parts of industry-level software.

In order to fill the gap, we have proposed a novel approach for automatically analysing semantic equivalence of large-scale C programs. To allow for a high scalability, we limit ourselves to comparison of two versions of the same software where we expect that one version is a refactoring of the other. Our work is mainly targeted at the Linux kernel which is motivated by our cooperation with Red Hat, nevertheless, the proposed method is generic and applicable to other C projects, too (as we demonstrate in our later presented experiments).

Our method builds on several assumptions about the analysed programs. First, we expect large parts of the compared versions to be identical and hence no complicated method is necessary to compare those parts. Second, if the remaining parts contain a refactoring (i.e., a change in the program that preserves semantics), it is very likely that it will match a commonly known refactoring pattern that has already been described in the literature or that has occurred in the history of the analysed project.

With respect to these assumptions, our proposed method is based on translating the compared programs into an intermediate compiler representation (we use in particular LLVM IR), followed by comparing individual instructions which is usually sufficient for large parts of the compared versions. In addition, to make this more effective, we apply light-weight static transformations (such as dead code elimination) to bring the compared versions of the code closer to each other. Where instruction-by-instruction approach is not sufficient, we check if the observed change corresponds to a pattern from our list of predefined *semantics-preserving change patterns*. We obtain this list by taking the existing lists of common refactorings from the literature and by analysing the history of the Linux kernel. In addition, our method supports *user-defined patterns* to cover situations when our list misses some types of changes.

Naturally, strict semantic stability is usually not required for the entire software, hence we expect our method to be used for certain parts only. To this end, we support comparison of individual functions and of global variables. Semantic comparison of global variables is motivated by them often being used to represent adjustable system parameters (such as the Linux kernel runtime parameters configurable using `sysctl`). To compare semantic equivalence of a global variable, we compare all functions that use the given variable, however, we note that the functions do not have to be compared in their entirety. Instead, it is sufficient to compare parts of the functions that may be influenced by the value of the variable. Therefore, we propose a specialized *program slicing* algorithm which is able to remove parts of functions that do not depend on a global variable.

Overall, we may summarize contributions of this part of the thesis as follows:

1. We propose a light-weight approach for checking semantic equivalence of program versions obtained by refactoring that is—to the best of our knowledge—much more scalable than other existing approaches for checking semantic equivalence.

2. We propose a representation of custom patterns of code changes using parametrised control-flow graphs and a method to match such patterns inside our approach.

3. We propose a novel algorithm for program slicing which is able to efficiently remove all parts of a function which are not affected by the value of a certain global variable.

4. We have implemented the proposed methods in a new open-source tool DiffKemp that is capable of checking preservation of semantics of refactored code compiled into the LLVM intermediate representation.

5. We demonstrate the capabilities of the approach on several practical applications on large-scale real-life projects including the Linux kernel (which has the size in millions of LOC) and the `musl` C standard library.

The contributions in this part are based on a paper published at ICST 2021 [92], which introduced the main method, and a paper published at NETYS 2022 [91], which expanded the method with custom change patterns (the paper has received the *best student paper award* at the conference). In addition, we have submitted an extended version of [92] into the *ACM Transactions on Software Engineering* journal which features (among other extensions) the described slicing algorithm. The core algorithm of DiffKemp is also subject to U.S. Patent 11 449 317-B2 [93].

Part I

Template-Based Verification of Programs with Data Structures

Chapter 2

Template-Based Verification of Software

The first part of this thesis focuses on formal verification of low-level C programs. Despite the fact that the research area of formal verification of software experiences great progress every year, there still remain a lot of open challenges. One of the most common ones is a limited applicability of the methods to real-world software. A traditional problem of many techniques is that they are often focused on verification of a single kind of property. This, in addition to the fact that different techniques have different strengths and weaknesses, makes many tools suitable for a certain class of programs only. That is often confusing to users as it requires them to have at least a basic knowledge of the methods that the tool implements. To address this issue, approaches combining multiple techniques together emerge in the recent years.

In this work, we focus on one of such methods which is implemented in a framework called 2LS ("Tools"). In particular, 2LS tries to approach the above problem from two different points of view. The first one addresses the traditional "soundness versus completeness" problem—verification approaches are usually either sound-but-incomplete, i.e., they are able to prove program correctness but have problems to distinguish actual errors from the spurious ones, or they are precise-but-unsound, i.e., they do not report spurious errors but but fail to prove error absence. 2LS tackles this problem by using multiple verification techniques in a unique algorithm called *k*I*k*I (which stands for *k-invariants and k-induction*). The algorithm combines three well-known techniques, namely abstract interpretation, bounded model checking (BMC), and k-induction.

BMC falls into the precise-but-unsound category when, given a sufficient amount of resources, it is able to find any violation of a property but usually fails to prove programs correct. On the other hand, k-induction is able to provide a correctness proof, but it requires k-inductive invariants to do that. The invariants are often hard to compute, hence 2LS uses abstract interpretation to express the invariants using abstract program semantics. This comes with the traditional problem of abstract interpretation—it cannot distinguish real counter-examples from the spurious ones (caused by the introduced over-approximation). If such a situation happens, BMC can be used to validate the found potential counter-example.

The second part where 2LS employs combination of multiple techniques is on the level of abstract domains used in the abstract interpretation part of the *k*I*k*I algorithm. Typically, an abstract domain captures only selected parts of program semantics. This makes

individual domains suitable for certain programs only, but real-world programs are usually complex and use many different data and control structures.

To overcome this problem, 2LS comes with a unique solution—it requires all the abstract domains to have the same form of so-called *templates* (hence we denote the approach as *template-based verification*). Templates are fixed, parametrised, first-order logic formulae, which allows to use an SMT solver to reason about them. Additionally, the unified form of the templates makes it easy to combine multiple abstract domains together (in the simplest form by just taking a conjunction of their templates) since the heavy-lifting of abstract operation combinators can be left to the underlying solver.

In this chapter, we present the most important concepts of 2LS that our work builds upon. In order to use an SMT solver for verification, 2LS represents the program semantics using a first-order logical formula. This is done by translating the program into a *static single assignment (SSA)* form which, due to its nature, makes generation of the formula straightforward. The SSA form used in 2LS has several specific features that cannot be found in other verification approaches, hence we present it in detail in Section 2.1.

After that, we introduce the core algorithm of 2LS, *k*I*k*I, in Section 2.2. We mainly focus on the template-based invariant inference since that is the part the most related to our contributions.

An integral part of this thesis is an implementation of all the proposed concepts into the 2LS framework. Therefore, we present the overall architecture of 2LS in Section 2.3.

In the following chapters, we propose our original contributions for verification of programs manipulating data structures that are suitable for the verification approach of 2LS. Since some of the introduced concepts are fairly complex, we illustrate them on examples. To facilitate that, we introduce a running example at the end of this chapter (in Section 2.4) that we will refer to throughout the entire part of the thesis.

2.1 Internal Program Representation

The 2LS framework is built upon the CPROVER infrastructure [30] and therefore uses the same intermediate representation called *GOTO programs*. In this language, any non-linear control flow, such as if- or switch-statements, loops, or jumps, is translated to equivalent *guarded goto* statements. These statements are branch instructions that include (optional) conditions. CPROVER generates one GOTO program per C function found in the parse tree. Furthermore, it adds a new main function that first calls an initialisation function for global variables and then calls the original program entry function.

After obtaining a GOTO representation of the analysed program from CPROVER, 2LS performs a light-weight static analysis to resolve function pointers to a case split over all candidate functions, resulting in a static call graph. Furthermore, assertions guarding against invalid pointer operations or memory leaks are inserted. These are described in detail in Section 3.5. In addition, 2LS uses local constant propagation and expression simplification to increase efficiency.

After running the mentioned transformations, 2LS performs a static analysis to derive data flow equations for each function of the GOTO program. The result is a *static single assignment (SSA) form* in which loops have been cut at the back edges to the loop head. The effect of these cuts is havocking of the variables modified in the loop at the loop head. This SSA is hence an over-approximation of the GOTO program. Subsequently, 2LS refines this over-approximation by computing invariants.

2.1.1 The Static Single Assignment Form

Program verification in 2LS is based on generating program abstractions using a solver. In order to simplify generation of a formula representing the program semantics, 2LS uses the *static single assignment form* (SSA) to represent programs. SSA is a standard program representation used in bounded model checking or symbolic execution tools. We use common concepts of SSA—introducing a fresh copy (version) x_i of each variable x at program location i in case x is assigned to at i, using the last version of x whenever x is read, and introducing a *phi* variable x_i^{phi} at a program join point i in case different versions of x come from the joined program branches. For an acyclic program, SSA is a formula that represents exactly the post condition of running the code.

In 2LS, the traditional SSA is extended by two new concepts: (1) over-approximation of loops in order to make the SSA acyclic and (2) a special encoding of the control-flow [19]. These concepts allow a straighforward transformation of the SSA form into a formula that can be passed to an SMT solver. In addition, 2LS leverages so-called *incremental solving* which means that it tries to reuse as large parts of the generated formulae as possible for successive solver invocations. This is a great benefit to the verification perfomance, however, it comes with several drawbacks. The main one is related to encoding of pointer dereferencing operations since on-demand concretisation of heap objects is not possible (as the formula representing the program cannot change). To overcome this limitation, we propose a new memory model and a special representation of memory-manipulating operations as one of the contributions of this thesis. These are introduced in Sections 3.1 and 3.2, respectively.

Over-Approximation of Loops

In order to be able to use a solver for reasoning about program abstractions, we extend the SSA by *over-approximating* the effect of loops. As was said above, the value of a variable x is represented at the loop head by a phi variable x_i^{phi} joining the value of x from before the loop and from the end of the loop body (here, we assume that all paths in the loop join before its end, and the same holds for the paths before the loop). However, instead of using the version of x from the loop end, it is replaced by a *free "loop-back" variable* x^{lb}. This way, the SSA remains acyclic, and, since the value of x^{lb} is initially unconstrained, the effect of the loop is over-approximated. To improve precision, the value of x^{lb} can be later constrained using a *loop invariant* that will be inferred during the analysis. A loop invariant is a property that holds at the end of the loop body, after any iteration and can be therefore assumed to hold on the loop-back variable.

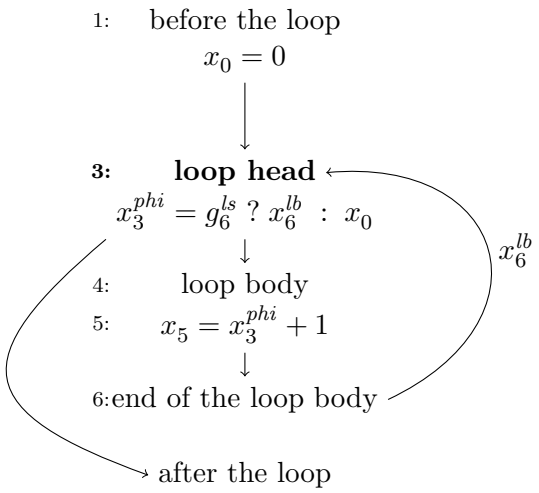
For a better illustration, we give an example of this SSA extension. Figure 2.1 shows a simple loop in C and its corresponding SSA. Instead of using x_5 in the phi variable, a fresh variable x_6^{lb} is introduced. Moreover, the join in the phi node is driven by a free Boolean variable g_6^{ls} (a so-called *loop-select guard*) modelling a non-deterministic choice between x_6^{lb} and x_0.

Encoding the Control-Flow

In 2LS, the program is represented by a single monolithic formula. It is therefore needed that the formula encodes control-flow information. This is achieved using so-called *guard* variables that track the reachability information for each program location. In particular, for each program location i, we introduce a Boolean variable g_i whose value encodes whether

```
1 unsigned x = 0;
2
3 while (x < 10)
4 {
5     ++x;
6 }
```

(a) A loop in C.



(b) Encoding of the loop in the SSA form.

Figure 2.1: Conversion of loops in the SSA form used in 2LS.

i is reachable. For example, in Figure 2.1, a guard g_5 encoding reachability of the loop body would have the value:

$$g_5 = x_3^{phi} < 10. \tag{2.1}$$

2.2 k-induction and k-invariants

The core part of 2LS is the kIkI algorithm (k-induction and k-invariants) which we introduce in this section. The goal of the algorithm is to connect widely-used verification techniques with well-understood interaction. There techniques are namely:

Bounded Model Checking (BMC) Given sufficient time and resource, BMC [15] will give counterexamples for all false safety properties, which are often of significant value for understanding the fault. However, only a small proportion of true properties can be proven by BMC.

k-Induction Generalising Hoare logic's ideas of loop invariants, k-induction [116] can prove true safety properties, and, in some cases, provide counterexamples to false ones. However, it requires inductive invariants, which can be expensive (in terms of user time, expertise, and maintenance).

Abstract Interpretation The use of over-approximation makes it easy to compute invariants which allow many true propositions to be proven [114, 110, 49]. However, false properties and true-but-not-provable properties may be indistinguishable. Tools implementing abstract interpretation may have limited support for a more complete analysis.

The kIkI algorithm draws together these techniques and combines them in a novel way so that they strengthen and reinforce each other. The k-induction technique uses syntactically restricted or simple invariants (such as those generated by abstract interpretation) to prove safety. Bounded model checking allows to test k-induction failures to see if they are real

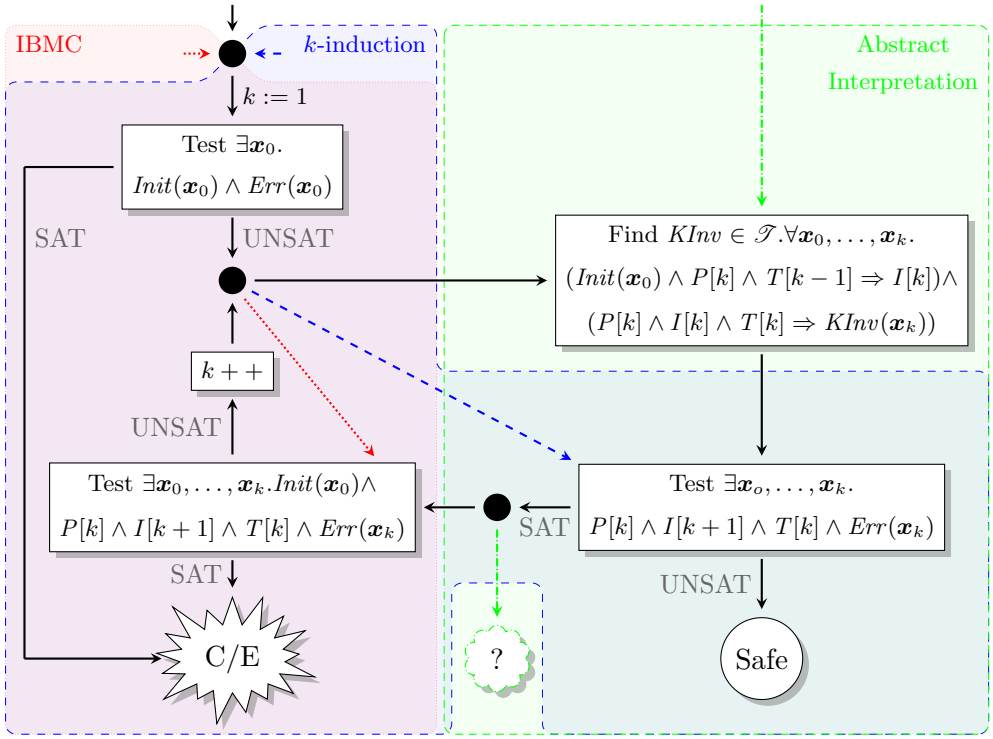


Figure 2.2: The kIkI algorithm [19].

counterexamples or, if not, to build up a set of assumptions about system behaviour. Template-based abstract interpretation is used for invariant inference with loop unrolling producing progressively stronger invariants, allowing the techniques to strengthen each other.

General flow of the kIkI algorithm is shown in Figure 2.2. The main loop of kIkI is based on incremental unwinding of the transition relation ($T[k]$ stands for the transition relation unwound up to the depth k). In the k-th iteration, the transition relation is unwound up to depth k and a k-inductive invariant is inferred ($\mathscr{T}$ is the set of predicates that can be used as invariant). The invariant is then used to strengthen the program's safety property in order to find a proof for the program's safety. In case that safety cannot be proved, the algorithm checks whether the current unwinding is sufficient to generate a counterexample. If this is not the case, the unwinding k is incremented and another iteration starts.

Incremental bounded model checking, k-induction, and classical over-approximating abstract interpretation are shown to be restrictions of kIkI. This is illustrated by the colored partitioning in Figure 2.2. kIkI can simulate k-induction by having $\mathscr{T} = \{\top\}$ and incremental BMC by over-approximating the first SAT check. Classical over-approximate abstract interpretation can be simulated by having $\mathscr{T} = \mathscr{A}$ for an abstract domain $\mathscr{A}$ and terminating with the result "unknown" if the first SAT check finds a model.

Internally, 2LS reduces program analysis problems expressed in second order logic such as invariant or ranking function inference to synthesis problems over templates. Therefore, it reduces (an existential fragment of) 2nd order Logic Solving (hence the name "2LS") to quantifier elimination in first order logic. This is one of the most important steps of the

algorithm and we explain it in detail in Section 2.2.1. In the remaining parts of this section, we present k*I*k*I*'s approach to bounded model checking (Section 2.2.3) and k-induction (Section 2.2.4).

2.2.1 Template-Based Predicate Inference

A key phase of kIkI is the generation of *KInv*, a k-inductive invariant. Perhaps the most obvious approach is to use an off-the-shelf abstract interpreter. This works but will fail to exploit the real power of kIkI. In each iteration, kIkI unrolls loops one more step (which can improve the invariant given by an abstract interpreter) and adds assumptions that previous unwindings do not give errors.

When directly using a solver, we would need to handle (the existential fragment of) second-order logic. As such solvers with reasonable efficiency are not currently available, we reduce to a problem that can be solved by iterative application of a first-order solver. We restrict ourselves to finding *k-inductive* invariants *KInv* of the form $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ where $\mathcal{T}$ is a fixed expression, a so-called *template*, over program variables $\boldsymbol{x}$ and template parameters $\boldsymbol{\delta}$. Fixing a template reduces the second-order search for an invariant to a first-order search for template *parameters*:

$$\begin{aligned} \exists \boldsymbol{\delta}. \forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k. \quad & (\mathit{Init}(\boldsymbol{x}_0) \wedge T[k-1] \Rightarrow \mathcal{T}[k](\boldsymbol{\delta})) \wedge \\ & (\mathcal{T}[k](\boldsymbol{\delta}) \wedge T[k] \Rightarrow \mathcal{T}(\boldsymbol{x}_k, \boldsymbol{\delta})) \end{aligned} \tag{2.2}$$

where $T[k]$ is the k-th unwinding of the transition relation and $\mathcal{T}[k]$ is a template for all states along the unwinding except for the last state $\boldsymbol{x}_k$:

$$T[k] = \bigwedge_{i \in [0,k-1]} \mathit{Trans}(\boldsymbol{x}_i, \boldsymbol{x}_{i+1}) \tag{2.3}$$

$$\mathcal{T}[k](\boldsymbol{\delta}) = \bigwedge_{i \in [0,k-1]} \mathcal{T}(\boldsymbol{x}_i, \boldsymbol{\delta}). \tag{2.4}$$

We resolve the $\exists\forall$ problem by an iterative solving of the negated formula, particularly of the second conjunct of (2.2), for different choices of constants $\boldsymbol{d}$ as the values of the parameter $\boldsymbol{\delta}$:

$$\exists \boldsymbol{x}_0 \ldots \boldsymbol{x}_k. \neg\big(\mathcal{T}[k](\boldsymbol{d}) \wedge T[k] \Rightarrow \mathcal{T}(\boldsymbol{x}_k, \boldsymbol{d})\big). \tag{2.5}$$

The resulting formula can be expressed in quantifier-free logics and efficiently solved by SMT solvers. Using this as a building block, one can solve the mentioned $\exists\forall$ problem.

From the abstract interpretation point of view, $\boldsymbol{d}$ is an abstract value, i.e., it represents (*concretises to*) the set of all program states $\boldsymbol{s}$—here, a state is a vector of values of variables from $\boldsymbol{x}$—that satisfy the formula $\mathcal{T}(\boldsymbol{s}, \boldsymbol{d})$. The abstract values representing the infimum $\bot$ and supremum $\top$ of the abstract domain denote the empty set and the whole state space, respectively: $\mathcal{T}(\boldsymbol{s}, \bot) \equiv \mathit{false}$ and $\mathcal{T}(\boldsymbol{s}, \top) \equiv \mathit{true}$ [19].

Formally, the concretisation function γ is:

$$\gamma(\boldsymbol{d}) = \{\boldsymbol{s} \mid \mathcal{T}(\boldsymbol{s}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{2.6}$$

In the abstraction function, to get the most precise abstract value representing the given concrete program state $\boldsymbol{s}$, we let

$$\alpha(\boldsymbol{s}) = \{\min(\boldsymbol{d}) \mid \mathcal{T}(\boldsymbol{s}, \boldsymbol{d}) \equiv \mathit{true}\}. \tag{2.7}$$

If the abstract domain forms a complete lattice, existence of such a minimal value $\boldsymbol{d}$ is guaranteed.

The algorithm for the invariant inference takes an initial value of $\boldsymbol{d} = \bot$ and iteratively solves (2.5) using an SMT solver. If the formula is unsatisfiable, then an invariant has been found, otherwise a model of satisfiability $\boldsymbol{d}'$ is returned by the solver. The model represents a counterexample to the current instantiation of the template being an invariant. The value of the template parameter $\boldsymbol{d}$ is then updated by combining the current value with the obtained model of satisfiability using a domain-specific join operator [19].

For example, assume we have a program with a loop that counts from 0 to 10 in variable x (such as the one from Figure 2.1), and we have a template $x \leq d$. Let us assume that the current value of the parameter d is 3, and we get a new model $d' = 4$. Then we update the parameter to 4 by computing $d \sqcup d' = \max(d, d')$, because max is the join operator for a domain that tracks numerical upper bounds.

In 2LS, we use a single template to compute all invariants of the analysed program. Therefore, typically, a template is composed of multiple parts, each part describing an invariant for a set of program variables. With respect to this, we expect a template $\mathcal{T}(\boldsymbol{x}, \boldsymbol{\delta})$ to be composed of so-called *template rows* $\mathcal{T}_r(\boldsymbol{x}_r, \delta_r)$, each row r describing an invariant for a subset $\boldsymbol{x}_r$ of variables $\boldsymbol{x}$ and having its own row parameter δ_r. The overall invariant is then a composition of individual template rows with computed values of the corresponding row parameters. The kind of the composition (it can be, e.g., a simple conjunction) is defined by each domain.

Guarded Templates

Since we use the SSA form rather than control flow graphs, we cannot use templates directly. Instead we use *guarded templates*. As described above, a template is composed of multiple template rows, each row describing an invariant for a subset of program variables. In a guarded template, each row r is of the form:

$$G_r(\boldsymbol{x}_r) \Rightarrow \widehat{\mathcal{T}}_r(\boldsymbol{x}_r, \delta_r) \tag{2.8}$$

for the r^{th} row $\widehat{\mathcal{T}}_r$ of the base template domain (e.g., template polyhedra). G_r is the conjunction of the SSA guards g_r associated with the definition of variables $\boldsymbol{x}_r$ occurring in $\widehat{\mathcal{T}}_r$. Since we intend to infer loop invariants, $G_r(\boldsymbol{x}_r)$ denotes the guard associated to variables $\boldsymbol{x}_r$ appearing at the loop head. Hence, template rows for different loops have different guards.

We illustrate the above on the example program in Figure 2.1 using a guarded interval template. The template has the form:

$$\mathcal{T}(x_6^{lb}, (\delta_1, \delta_2)) = \begin{array}{lcl} g_3 \wedge g_6^{ls} & \Rightarrow & x_6^{lb} \leq \delta_1 \wedge \\ g_3 \wedge g_6^{ls} & \Rightarrow & -x_6^{lb} \leq \delta_2. \end{array} \tag{2.9}$$

Here, g_3 and g_6^{ls} guard the definition of x_6^{lb}—g_3 expresses the fact that the loop head is reachable and g_6^{ls} expresses that x_6^{lb} is chosen as the value of x_3^{phi}.

In case x was a dynamically allocated object, the template row guard would also contain guards associated with the allocation of the given object. These are in more details described in Section 3.2

1: $\boldsymbol{d} \leftarrow \bot$
2: **while** $\mathcal{T}[k](\boldsymbol{d})$ is not an invariant **do**
3: solver $\leftarrow \mathcal{T}[k](\boldsymbol{d})$
4: solver $\leftarrow \neg\mathcal{T}(\boldsymbol{x}_k, \boldsymbol{d})$
5: **if** solver.solve() = SAT **then**
6: **for** $\mathcal{T}_r(\boldsymbol{x}_r, d_r) \in \mathcal{T}(\boldsymbol{x}, \boldsymbol{d})$ **do**
7: $m_r \leftarrow$ solver.model($\boldsymbol{x}_r$)
8: find d_m s.t. $\mathcal{T}_r(m_r, d_m)$ holds
9: $d_r \leftarrow d_r \sqcup d_m$
10: **end for**
11: **end if**
12: **end while**

Figure 2.3: Generic strategy iteration algorithm for solving the $\exists\forall$ problem.

Solving of the Exists-Forall Problem

As discussed above, it is necessary to solve an $\exists\forall$ problem to find values for template parameters $\boldsymbol{\delta}$ to infer invariants. The well-known method [110, 20] for solving this problem, expressed in (2.5), using an SMT solver is to repeatedly check satisfiability of the formula for different abstract values $\boldsymbol{d}$ (starting with the infimum $\boldsymbol{d} = \bot$):

$$\mathcal{T}[k](\boldsymbol{d}) \wedge T[k] \wedge \neg\mathcal{T}(\boldsymbol{x}_k, \boldsymbol{d}). \tag{2.10}$$

If it is unsatisfiable, then we have found an invariant; otherwise, we join the model returned by the solver with the previous abstract value $\boldsymbol{d}$ and repeat the process with the new value of $\boldsymbol{d}$ obtained from the join.

This method corresponds to performing a classical Kleene iteration on the abstract lattice up to convergence. Convergence is guaranteed because the abstract domains in 2LS are finite. However, while this method might be sufficient for some abstract domains (especially those with a low number of possible states), it is practically unusable for other ones. For example, when dealing with integer variables, the height of the lattice is enormous and even for a one-loop program incrementing an unconstrained 64-bit variable, the naïve algorithm will not terminate within human life time. Hence, some abstract domains (e.g., the template polyhedra domain) use an optimised method [19].

In general, we refer to these methods as to *domain strategy iteration methods*. We observe that even though some domains require specialised approaches to assure convergence, for most of the domains, the algorithms are to some extent similar. This is related to the fact that abstract domain templates are typically composed of multiple template rows, as described earlier in this section.

With respect to this, we developed a *generic strategy iteration* algorithm [95] parametrised by an abstract domain having the form of a template. The algorithm is shown in Figure 2.3.

The algorithm repeatedly solves (2.10) for the given abstract domain. If the formula is satisfiable, then for each template row r, the algorithm gets the model of satisfiability for the variables that r describes. The obtained model (i.e., the values of the concerned variables) is used to instantiate the template row formula (line 8) and the corresponding value of the template parameter is joined with the previous value of the row parameter.

Since 2LS uses incremental solving, we assume that the transition relation (expressed by the SSA form) is already a part of the solver clause set. Moreover, formulae added to the

solver clause set on lines 3 and 4 are removed after each iteration. This ensures efficiency of the method since only the new formulae need to be re-solved every time.

In addition, as we mentioned in the previous section, an optimisation may be required in order to assure scaling of the algorithm. In such a case, lines 7-9 are replaced by the optimised method for determining values of template row parameters.

2.2.2 Abstract Domains in 2LS

2LS supports analysis of various program features such as reachability of assertions, termination, or memory safety. For most of these analyses, 2LS introduces abstract domains [31] for invariant inference, which is one of the steps of the kIkI algorithm.

There are numerous existing abstract domains in 2LS. The most extensively used domain is the template polyhedra domain which is used for tracking numerical values of memory objects [19]. For analysing termination, 2LS uses domains for ranking functions and recurrent sets [27]. In addition to these, as a part of this thesis, we develop two new domains for analysing programs manipulating data structures: (1) a domain for the shape of dynamic data structures [85] and (2) a domain for the contents of arrays. We describe these two in detail in Chapters 3 and 4, respectively.

From the point of view of our contributions, the most important existing domain is the template polyhedra domain, hence we briefly introduce it below. The reason is that both heap structures and arrays often contain numerical values and we combine our new domains with the template polyhedra domain to analyse contents of such structures.

Template Polyhedra Abstract Domain

Template polyhedra [114] are a class of templates for numerical variables which have the form $\mathcal{T} = (\mathbf{A}\boldsymbol{x} \leq \boldsymbol{\delta})$ where $\mathbf{A}$ is a matrix with fixed coefficients. The r^{th} *row* of the template corresponds to the constraints generated by the r^{th} row of the matrix $\mathbf{A}$. Subclasses of such templates include:

- *Intervals* which require constraints of the following form for each variable x_i:

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix} x_i \leq \begin{pmatrix} \delta_{i1} \\ \delta_{i2} \end{pmatrix} \tag{2.11}$$

- *Zones* (differences) which extend the intervals by adding a constraint for each pair of variables x_i and y_i:

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix} (x_i \; y_i) \leq \begin{pmatrix} \delta_{i1} \\ \delta_{i2} \end{pmatrix} \tag{2.12}$$

- *Octagons* [97] which extend the zones by adding additive constraints for each pair of variables.

In this work, we use the interval and the zones templates. An example of a guarded interval template used in 2LS can be found in Equation (2.9).

In template expressions used in 2LS, variables $\boldsymbol{x}$ are *bit-vectors* representing signed or unsigned integers. All such variables can be mixed in template constraints. Type promotion rules are applied such that the bit-width of the types of the expressions are extended in order to avoid arithmetic under- and overflows in the template expressions. $\top$ corresponds to the respective maximum values in the promoted type, whereas $\bot$ must be encoded as a special symbol.

2.2.3 Incremental Bounded Model Checking

Bounded Model Checking (BMC) [15] focuses on refutation by picking an *unwinding limit* k and solving the equation

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k. \mathit{Init}(\boldsymbol{x}_0) \wedge T[k] \wedge \neg P[k+1] \tag{2.13}$$

where $T[k]$ is an unwound transition relation as defined by (2.3) and $P[k]$ is a predicate stating that k states are error-free:

$$P[k] = \bigwedge_{i \in [0,k-1]} \neg \mathit{Err}(\boldsymbol{x}_i). \tag{2.14}$$

Models of (2.13) correspond to concrete counterexamples of some length $n \leqslant k$. The unwinding limit gives an *under-approximation* of the set of reachable states and thus can fail to find counterexamples that take a large number of transition steps. In practice, BMC works well as the formula is existentially quantified and thus is in a fragment handled well by SAT and SMT solvers.

Incremental bounded model checking is one of the core components of the kIkI algorithm. It corresponds to the red part in Figure 2.2. Incremental BMC (IBMC) (e.g. [39]) uses repeated BMC checks (often optimised by using the solver incrementally) with increasing bounds to avoid the need for a fixed bound. If the bound starts at 0 (i.e., checking $\exists x_0. \mathit{Init}(\boldsymbol{x}_0) \wedge \mathit{Err}(\boldsymbol{x}_0)$) and is increased by one in each step (this is the common use-case), then it can be assumed that there are no errors at previous states, giving a simpler test:

$$\exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k. \mathit{Init}(\boldsymbol{x}_0) \wedge T[k] \wedge P[k] \wedge \mathit{Err}(\boldsymbol{x}_k). \tag{2.15}$$

2.2.4 Incremental k-Induction

Incremental k-induction [116] is the blue part of the kIkI algorithm in Figure 2.2. It can be viewed as an extension of IBMC that can show system safety as well as produce counterexamples. It makes use of *k-inductive invariants*, which are predicates that have the following property:

$$\forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k. I[k] \wedge T[k] \Rightarrow \mathit{KInv}(\boldsymbol{x}_k) \tag{2.16}$$

where

$$I[k] = \bigwedge_{i \in [0,k-1]} \mathit{KInv}(\boldsymbol{x}_i).$$

k-inductive invariants have the following useful properties:

- Any inductive invariant is a 1-inductive invariant and vice versa.
- Any k-inductive invariant is a $(k+1)$-inductive invariant.
- A (finite) system is safe if and only if there is a k-inductive invariant KInv which satisfies:

$$\begin{aligned} \forall \boldsymbol{x}_0 \ldots \boldsymbol{x}_k. \quad & (\mathit{Init}(\boldsymbol{x}_0) \wedge T[k] \Rightarrow I[k]) \wedge \\ & (I[k] \wedge T[k] \Rightarrow \mathit{KInv}(\boldsymbol{x}_k)) \wedge \\ & (\mathit{KInv}(\boldsymbol{x}_k) \Rightarrow \neg \mathit{Err}(\boldsymbol{x}_k)) \,. \end{aligned} \tag{2.17}$$

Showing that a k-inductive invariant exists is sufficient to show that an inductive invariant exists *but it does not imply that the k-inductive invariant is an inductive invariant.* Often the corresponding inductive invariant is significantly more complex. Thus k-induction can be seen as a trade-off between invariant *generation* and *checking* as it is a means to benefit as much as possible from simpler invariants by using a more complex property check.

However, finding a candidate k-inductive invariant is still hard, and so implementations often use $\neg Err(\boldsymbol{x})$ as the candidate. Similarly to IBMC, linearly increasing k can be used to simplify the expression by assuming there are no errors at previous states:

$$\begin{aligned} \exists \boldsymbol{x}_0, \ldots, \boldsymbol{x}_k. \quad & (Init(\boldsymbol{x}_0) \wedge T[k] \wedge P[k] \wedge Err(\boldsymbol{x}_k)) \vee \\ & (T[k] \wedge P[k] \wedge Err(\boldsymbol{x}_k)). \end{aligned} \tag{2.18}$$

A model of the first part of the disjunct is a concrete counterexample (k-induction subsumes IBMC) and if the whole formula has no models, then $\neg Err(\boldsymbol{x})$ is a k-inductive invariant and the system is safe.

2.3 Implementation and Architecture of the 2LS Framework

2LS is based on the CPROVER framework[1] which is the core part of another well-known verification tool, CBMC [30]. It is implemented in C++ and currently has around 25 KLOC (not counting CPROVER itself). The source code is available under the BSD license at `https://www.github.com/diffblue/2ls`.

Similarly to other verification tools, 2LS follows the pipeline architecture of compilers, consisting of a front-end for parsing and type checking the source code; middle-end passes for transforming and analysing the code based on an intermediate representation (which in 2LS is the single static assignment form, SSA); and a back-end, which, however, instead of code generation performs the analysis and verification. An overview of the 2LS architecture is shown in Figure 2.4. In the rest of this chapter, we describe the individual components in more detail.

Front-End

The verified program is first processed with an off-the-shelf C preprocessor (such as `gcc -E`) and subsequently translated into a GOTO program using the `goto-cc` compiler from the CPROVER framework. Afterwards, several analyses are performed to gather information important for building the SSA form (e.g., the *data-flow analysis* used to create valid SSA equations such as placement of phi nodes, etc.). As a part of this thesis, we introduce two new analyses that facilitate dealing with memory objects: (1) *dynamic object analysis* for determining the necessary numbers of abstract dynamic objects for each allocation site (cf. Section 3.1) and (2) *may-point-to analysis* used for pre-materialization of the results of memory-manipulating operations (cf. Section 3.2).

After all the analyses are done, the GOTO program is instrumented with new assertions guarding, e.g., against invalid memory operations or arithmetic overflows. Finally, a GOTO program is translated into the SSA form described in Section 2.1.1.

[1] `https://www.cprover.org`

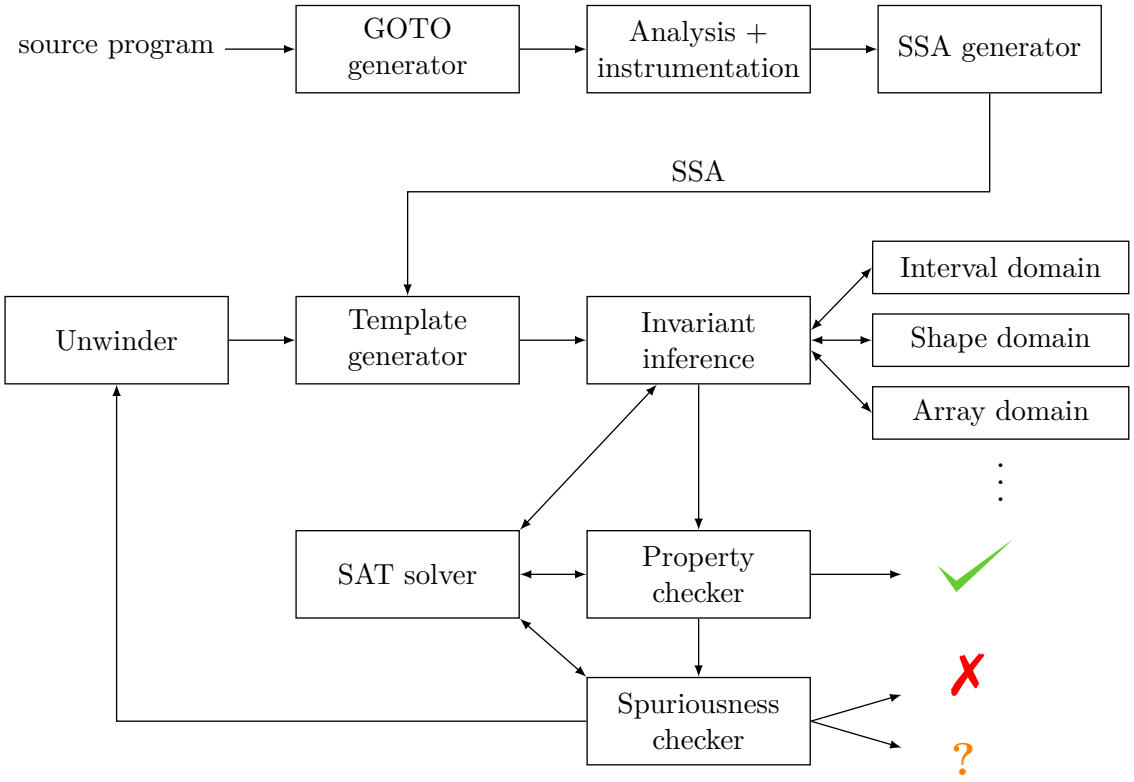


Figure 2.4: 2LS architecture overview.

Middle-End

The middle-end analyses the generated SSA and implements several steps of the *k*I*k*I algorithm introduced in Section 2.2. The most important one is generation of k-inductive invariants. Each iteration of *k*I*k*I starts with creating a template for the chosen abstract domain. Then, a solver is used to infer an invariant from the template using the algorithm described in Section 2.2.1. After the invariant is obtained, it is passed to the backend which uses it to check correctness of the program.

Another important transformation of the SSA form is loop unwinding. It is executed at the end of each *k*I*k*I iteration and is an essential step of the k-induction and BMC parts of the algorithm. However, the concepts proposed and implemented within this thesis do not deal with loop unwinding (they rather use it in the form that has been already present in 2LS), hence we do not describe it in detail. For more information on the way loop unwinding is implemented in 2LS, refer to [69].

Back-End

On its back-end, 2LS performs verification of the program with the help of the inferred invariant. First, the invariant is checked whether it is sufficiently strong to prove necessary properties and to claim the program safe. If that is not the case, i.e., the invariant includes an error state, BMC is used to check if the error is reachable within the current unwinding, which would efficiently prove the program incorrect. If the error is found to be spurious, and the unwinding level has not reached the maximum value, the SSA form is unwound and the algorithm continues.

```
1  struct node {
2    int data[1000];
3    int size;
4    struct node *next;
5  };
6
7  int main() {
8    struct node *list = NULL;
9    while (nondet()) {
10     struct node *n = malloc(sizeof(*n));
11     for (int i = 0; i < 1000; i++)
12       n->data[i] = 0;
13     n->size = 0;
14     n->next = list;
15     list = n;
16   }
17   int x = nondet();
18   assert(!list || list->data[x] == 0);
19 }
```

Figure 2.5: Running example for abstract domains

All steps of the backend require incremental solvers. Since support for incremental solving in SMT solvers is still lagging behind in comparison to SAT solvers, 2LS uses Glucose 4.0[2]. Consequently, CPROVER is used to translate the SSA equation into a CNF formula by bit-precise modelling of all expressions plus the Boolean guards. This formula is incrementally extended to perform invariant generation, to add further loop unwindings, and to add assertions for property checks. All this happens using a single solver instance so that information learned by the solver is never discarded. This approach has a great influence on the speed of verification in 2LS as we also show on our experiments.

2.4 Running Example

To facilitate understanding of concepts described in the following chapters, we introduce an example program in Figure 2.5 which we will use to illustrate the proposed mechanisms. The program features initialization of a data structure called *unrolled linked list* [115], which is a linked list whose nodes contain arrays of values. To check correctness of operations over such a data structure, the verification tool needs to be able to reason over linked heap structures as well as the integer contents of arrays at the same time. We achieve this in 2LS by introducing two new abstract domains and combining them with the existing template polyhedra domain (cf. Chapters 3 and 4 for details).

Even though we give the example in C, 2LS will transform the program into the SSA representation introduced in Section 2.1.1. Due to space reasons, we do not give the full SSA, we just mention its parts important for understanding the examples presented later.

[2] https://github.com/mi-ki/glucose-syrup

Each variable assignment creates a new SSA variable indexed by the line number of the definition. For instance, line 15 will be transformed into the equality $list_{15} = n_{10}$ since *list* is assigned on this line while the last assignment into n happened on line 10. Data-flow join points are represented using phi variables, e.g., the value of *list* on line 9 is given by the equation:

$$list_9^{phi} = g_{16}^{ls} \ ? \ list_8 : list_{16}^{lb} \tag{2.19}$$

where g_{16}^{ls} is a free boolean variable representing a non-deterministic choice, and $list_{16}^{lb}$ is an over-approximation of the value of *list* coming from the end of the loop body (line 16), which will be constrained by the computed invariant.

Operations over pointers and dynamic memory are encoded in a special way which we introduce as a part of this thesis in Section 3.2. Hence, we defer presentation of the concrete SSA equations for the relevant statements to that section.

Besides data-flow equations, the SSA form encodes the control flow using so-called *guard variables*. For each location (line) i, the variable g_i represents the boolean condition under which the line is reachable from the beginning of the program. For example, g_8 is *true* while g_{10} is non-deterministic (i.e., left unconstrained) as the `nondet` function returns a random integer value.

Prior to our work, 2LS would only be able to compute invariants over numerical variables stored on the stack. In particular, an invariant in the intervals domain would be computed for i_{12}^{lb}, which represents the value of the variable `i` returning from the end of the inner loop body to the loop head. Since 2LS uses guarded templates, the computed invariant is:

$$\begin{aligned} &g_{11} \wedge g_{12}^{ls} \Rightarrow i_{12}^{lb} <= 1000 \ \wedge \\ &g_{11} \wedge g_{12}^{ls} \Rightarrow -i_{12}^{lb} <= 1. \end{aligned} \tag{2.20}$$

The invariant states that if the loop head is reachable (g_{11}) and i_{12}^{lb} is chosen as the result of the corresponding phi expression (g_{12}^{ls}), then, at the end of the loop, the value of `i` will always be between 1 and 1000.

Chapter 3

Verification of Heap-Manipulating Programs

The first contribution of this part of the thesis is a novel approach for verification of programs manipulating data structures on the program heap. The method that we propose is suitable for the template-based verification approach of 2LS introduced in Chapter 2.

The most important contribution of this part is a new *abstract domain* for tracking the points-to relation between pointers and memory objects. We refer to it as to the *shape domain* and describe it in detail in Section 3.3.

Since 2LS comes with a rather specific program representation based on the SSA form, we discovered that solely the new domain is not sufficient for sound verification of heap-manipulating programs. The main problem lies in the fact that 2LS heavily relies on incremental solving and hence it attempts to reuse as large parts of the formulae as possible for successive solver invocations. This typically means that the formula representing the program semantics cannot change during the verification process, especially during the invariant inference phase. This creates problems when it comes to representation of dynamic memory operations. Since dynamic memory is possibly unbounded, an abstraction is necessary for sound verification. The traditional approach is to create an abstract representation of the dynamic memory and concretise heap objects on demand during the verification process. However, this is not possible in 2LS (since the formula representing the program cannot change), hence a different approach is necessary.

To overcome this limitation of 2LS and ensure soundness, we introduce a new *memory model* and a new *representation of memory-manipulating operations*. We present these in Sections 3.1 and 3.2, respectively.

In addition, since we deal with complex data structures, it is usually not sufficient to use a single domain to analyse a program. For instance, the shape domain may describe pointer links among heap objects, but we also need a numerical domain to reason about the contents of the objects to prove, e.g., correctness of an algorithm for finding a maximal node in a linked list. To facilitate such analyses, we introduce *domain combinations* that allow to compose abstract domains together. The domain combinations are presented in Section 3.4.

Besides verification of reachability assertions, heap-manipulating programs are usually verified for *memory safety*, too. To achieve this in 2LS, we introduce automatic instrumentation of the analysed program that allows to check for the most typical memory safety errors. Our approach is described in Section 3.5.

The rest of this chapter is heavily based on our FMCAD 2018 paper [85] and two SV-COMP papers [87, 90].

3.1 Memory Model

We now describe a memory model that we use to represent all program memory [85]. Our model is object-based, and we distinguish objects allocated *statically* (i.e., variables on the stack and global variables) and *dynamically* (i.e., on the heap).

3.1.1 Static Memory Objects

In our approach, we work with non-recursive programs with all functions inlined. Therefore, we do not need to consider the stack and the set Var of static memory objects corresponds to the set of all program variables. Each variable is uniquely identified by its name.

For convenience, we define subsets of Var that correspond to sets of variables of a chosen type:

- $NVar$ is the set of all variables of a numeric type (integer or floating point).
- $PVar$ is the set of all variables of a pointer type.
- $AVar$ is the set of all variables of an array type. Each array has its element type which we often refer to as the inner type.
- $SVar$ is the set of all variables of a structure type. Each structure type defines a set of named *fields*, each of them having its own type. We use Fld to denote the set of all fields used in the analysed program. Similarly to the variables, we let $NFld, PFld, AFld \subseteq Fld$ be the sets of all fields of numerical, pointer, and array types, respectively. In order to express access to individual fields of a structure-typed variable, we use the "dot" notation as is common in C.

We assume $NVar$, $PVar$, $AVar$, and $SVar$ are pairwise disjoint.

3.1.2 Dynamic Memory Objects

To represent dynamic memory objects (i.e., those allocated using `malloc` or some of its variants), we use *abstract dynamic objects*. An abstract dynamic object represents a set of concrete dynamic objects allocated by the same `malloc` call. We refer to a `malloc` call at a program location i as to an *allocation site* i.

Generally, a single abstract dynamic object is not sufficient to represent all concrete objects allocated by a single `malloc` call. This is due to the fact that the analysed program may use several concrete objects allocated at the same allocation site at the same time. If such objects are, e.g., compared, our memory model must allow us to distinguish them. This can be done either by concretisation on demand (as is common in many memory models) or by pre-materialisation of a sufficient number of objects at the beginning of the analysis. Since our approach uses a single formula to represent the analysed program and leverages on small incremental changes of the formula during the analysis, we use the latter approach.

Therefore, we use the set $AO_i = \{ao_i^k \mid 1 \leq k \leq n_i\}$ of abstract objects to represent all concrete objects allocated at the allocation site i. The number n_i of necessary objects

is determined for each allocation site using an approach described later in this section. The set of all dynamic objects of the analysed program is then defined as $AO = \cup_i AO_i$. Together with the set of all static objects, we define $Obj = Var \cup AO$ to be the set of all memory objects of our program abstraction. We require $Var \cap AO = \emptyset$ and $AO_i \cap AO_j = \emptyset$ for $i \neq j$.

Similarly to static objects, we denote NAO, PAO, AAO, and SAO the sets of dynamic objects of a numerical, pointer, array, and structure type, respectively. Fields of structure-typed dynamic objects, i.e., elements of the set $SAO \times Fld$, represent abstractions of the appropriate field of all represented concrete objects. Using the above sets, we may define the set Num of all *numerical objects* in the program as:

$$Num = NVar \cup NAO \cup ((SVar \cup SAO) \times NFld). \tag{3.1}$$

Analogically, we define the sets Ptr and Arr of all *pointers* and *arrays*, respectively, as follows:

$$Ptr = PVar \cup PAO \cup ((SVar \cup SAO) \times PFld) \tag{3.2}$$

$$Arr = AVar \cup AAO \cup ((SVar \cup SAO) \times AFld). \tag{3.3}$$

Pointers can be assigned addresses of objects. Since we do not support pointer arithmetic, only *symbolic addresses* and a special address `null` are considered. We use the operator & to get the address of both static and dynamic objects. For abstract dynamic objects, the symbolic address is an abstraction of symbolic addresses of all represented concrete objects. We define the set $Addr$ of all addresses in the program as:

$$Addr = \{\&o \mid o \in Obj\} \cup \{\texttt{null}\}. \tag{3.4}$$

Dynamic Object Pre-Materialisation

Above, we mentioned that, for each allocation site i, we represent (a potentially infinite number of) all objects allocated at i by a finite number n_i of abstract dynamic objects. In order for this abstraction to be sound, it is sufficient that this number is equal to the number of distinct concrete objects, allocated at i, that may be simultaneously pointed to at any location of the analysed program.

In order to compute this number, we first perform a standard static *may-alias analysis*. This analysis determines, for each program location j, the set P_i^j of all *pointer expressions* used in the program, that may point to some object allocated at i. Here, pointer expressions may be one of the following:

- Pointer variables.
- Dereferences of pointers to pointers. These correspond to pointer-typed dynamic objects.
- Pointer fields of structure-typed variables.
- Dereferences of pointers to structures followed by an access to a pointer-typed field. These correspond to pointer-typed fields of dynamic objects. Here, we use the C notation based on an arrow (e.g., $p{\rightarrow}n$ to express a dereference of p followed by an access to the field n of the pointed object).
- Pointer-typed elements of arrays.

For simplicity, we assume that all chained dereferences of forms $**p$ or $p{\rightarrow}f_1{\rightarrow}f_2$ are split into multiple dereferences using an intermediate pointer variable which is added to $PVar$.

Next, we compute the must alias relation $\sim_j$ over the set of all pointer expressions. For each pair of pointer expressions p and q and for each program location j, $p \sim_j q$ iff p and q must point to the same concrete object (i.e., they must alias) at j.

Finally, we partition each computed P_i^j into equivalence classes by $\sim_j$, and the number n_i is given by the maximal number of such classes for any j.

3.2 Representation of Memory-Manipulating Operations

One of the strengths of 2LS is its efficient usage of incremental SMT solving that contributes to the great overall speed of the analysis. It is facilitated by representing the entire program as a single monolithic formula which is only minimally changed during verification. This approach, however, brings a number of obstacles, especially when it comes to representation of operations over dynamically allocated memory. Typically, a pointer dereference of an abstract object would be *concretised* on demand during analysis. In 2LS, that would require an update of the formula representing the program (which is not desirable), hence we *pre-materialise* results of all memory-manipulating operations at the beginning of the analysis. In this section, we describe how this is done for the two most typical operations—dynamic memory allocation and pointer dereferencing (where we use slightly different approaches for read and write accesses).

3.2.1 Dynamic Memory Allocation

As said in Section 3.1, all concrete objects allocated by a single `malloc` call at a program location i are abstracted by a set of abstract dynamic objects AO_i. In the SSA form, we represent such a call by a non-deterministic choice among objects from AO_i. A program assignment `p = malloc(...)` is therefore transformed into the formula

$$p_i = g_{i,1}^{os} \,?\, \&ao_i^1 : (g_{i,2}^{os} \,?\, \&ao_i^2 : (\ldots (g_{i,n_i-1}^{os} \,?\, \&ao_i^{n_i-1} : \&ao_i^{n_i}))) \tag{3.5}$$

where $g_{i,j}^{os}$, $1 \leq j < n_i$, are free Boolean variables, so-called *object-select guards*.

Running Example

Let us illustrate memory allocation equations on the running example from Figure 2.5. The program features one `malloc` call on line 10. At any point of the program, there may be at most two distinct concrete objects allocated at this location pointed by some live pointers (e.g., by pointers `n` and `list` on line 11). Therefore, the set of abstract dynamic objects for line 10 is $AO_{10} = \{ao_{10}^1, ao_{10}^2\}$ and the corresponding SSA equation is:

$$n_{10} = g_{10}^{os} \,?\, \&ao_{10}^1 : \&ao_{10}^2. \tag{3.6}$$

We omit the second part of the subscript in the object-select guard for better legibility.

3.2.2 Reading through Dereferenced Pointers

We now describe encoding of a pointer dereference appearing on the right-hand side of an assignment or in a condition (i.e., in an R-expression). Prior to generating the SSA, we perform a static *may-points-to analysis* which over-approximates—for each program

location i and for each pointer $p \in Ptr$—the set of all objects from Obj that p may point to at i. A dereference of p at i is then represented by a choice among the pointed objects.

Moreover, to simplify the representation and to improve precision (for reasons explained below), we also introduce so-called *dereference variables*. Let $*p$ be an R-expression that appears at a program location i and let us assume that p may point to a set of objects $O \subseteq Obj$. We replace $*p$ by a fresh variable $drf(p)_i$, and we define its value as follows:

$$\begin{aligned} &\bigwedge_{o \in O} p_j = \&o \implies drf(p)_i = o_k \wedge \\ &\left(\bigwedge_{o \in O} p_j \neq \&o \right) \implies drf(p)_i = o_\perp \end{aligned} \tag{3.7}$$

where p_j and o_k are the relevant versions of the variables p and o, respectively, at the program location i, and $o_\perp$ is a special "unknown object" representing the result of a dereference of an unknown or invalid (`null`) pointer.

Informally, Formula (3.7) expresses the fact that $drf(p)_i$ equals the value of o at i in case that p points to o at i, and it equals the value of the unknown object otherwise.

Running Example

We again illustrate the equations on the running example from Figure 2.5. The program features one read through a pointer in the assertion on line 18 (`list->data[x]`). The dereferenced pointer is `list`, with its last valid SSA version coming from the loop head, i.e., $list_9^{phi}$. At this program location, `list` may point to one of the abstract dynamic objects ao_{10}^1, ao_{10}^2 introduced in the previous section or to `null`. The variable `x` is defined on line 17, hence its SSA expression is x_{17}. With respect to the above definitions, the SSA equation corresponding to the asserted property on line 18 is:

$$list_9^{phi} = \texttt{null} \vee drf(list)_{18}.data[x_{17}] = 0 \tag{3.8}$$

where we define the value of $drf(list)_{18}.data$ as:

$$\begin{aligned} list_9^{phi} = \&ao_{10}^1 &\implies drf(list)_{18}.data = ao_{10}^1.data_9^{phi} \wedge \\ list_9^{phi} = \&ao_{10}^2 &\implies drf(list)_{18}.data = ao_{10}^2.data_9^{phi} \wedge \\ list_9^{phi} \neq \&ao_{10}^1 \wedge list_9^{phi} \neq \&ao_{10}^2 &\implies drf(list)_i.data = o_\perp.data. \end{aligned} \tag{3.9}$$

Note that we use $drf(list)_{18}.data$ rather than $drf(list)_{18}$. The reason for this is that we will compute different invariants for different members of the heap objects, hence we define distinct SSA symbols for them (i.e., we compute separate invariants for $ao_{10}^1.data$, $ao_{10}^1.size$, and $ao_{10}^1.next$ rather than a single invariant for the entire dynamic object ao_{10}^1). Here, the symbol $ao_{10}^1.data$ represents abstraction of the *data* field of all objects allocated on line 10 and for line 18, its valid SSA definition comes from the loop head (i.e., $ao_{10}^1.data_9^{phi}$). Same applies for $ao_{10}^2.data$. The expression $o_\perp.data$ represents an invalid object.

3.2.3 Writing through Dereferenced Pointers

Similarly to the operation of reading, we introduce an SSA encoding for the operation of writing into memory using a pointer dereference. Again, we leverage on the may-points-to analysis described above, and we again build on the special *dereference variables*. Let us

have an assignment $*p = v$ at program location i and let us assume that p may point to a set of objects $O \subseteq Obj$ at the entry to i. This assignment is replaced by the equality:

$$drf(p)_i = v_l. \tag{3.10}$$

where v_l is the valid version of v at i. The dereference variable $drf(p)_i$ is then used to update the value of the referenced object. This is done using the formula:

$$\bigwedge_{o \in O} o_i = (p_j = \&o)\,?\,drf(p)_i : o_k \tag{3.11}$$

where p_j, o_k are relevant versions of p and o, respectively, at the program location i.

In other words, this formula expresses the fact that an object o is assigned the value of v in the case when p points to o, and it keeps its original value otherwise. As mentioned above, usage of dereference variables may improve precision of the representation. This happens especially when we write into an abstract object through some pointer and afterwards read through the same pointer without changing its value nor the value of the pointed object in between. In such cases, we may reuse the same dereference variable which ensures that we get the same value that was written, which needs not happen otherwise since we read from an abstract object representing a number of concrete objects.

Running Example

Let us illustrate writing through pointers on the running example from Figure 2.5, in particular on the memory write on line 13 (`n->size = 0`). The dereferenced pointer is `n` which is previously assigned the return value of the `malloc` call, i.e., it may point to one of the abstract dynamic objects ao^1_{10}, ao^2_{10} introduced in Section 3.2.1. The last valid SSA expression for `n` is n_{10}.

With respect to the above definitions, the SSA equation corresponding to line 13 is:

$$drf(n)_{13}.size = 0 \tag{3.12}$$

and the dereference variable $drf(n)_{13}.size$ is used to update the apropriate fields of the two abstract dynamic objects:

$$\begin{aligned} ao^1_{10}.size_{13} &= (n_{10} = \&ao^1_{10})\ ?\ drf(n)_{13}.size : ao^1_{10}.size_{10} \wedge \\ ao^2_{10}.size_{13} &= (n_{10} = \&ao^2_{10})\ ?\ drf(n)_{13}.size : ao^2_{10}.size_{10}. \end{aligned} \tag{3.13}$$

When combined with Eq. (3.6), the `size` field of exactly one of the abstract dynamic objects is updated while the other one retains its original (unconstrained) value coming from the allocation site on line 10.

3.3 Abstract Domain for Heap Shape Analysis

We now present our first abstract domain for analysing programs working with data structures—the *heap shape domain*. The shape of the heap is defined by pointer links among memory objects. Therefore, our proposed shape domain is limited to the set Ptr of all pointers as defined by Equation (3.2). More particularly, since the shape domain is used to infer loop invariants, we limit it to the set Ptr^{lb} of all *loop-back* pointers which in our SSA representation abstract values of pointers returning from the ends of the loops that the pointers are updated in.

The shape domain over-approximates the *may-point-to* relation between the set Ptr^{lb} and the set of all symbolic addresses $Addr$. We define the form of the heap template to be the formula

$$\mathcal{T}^S \equiv \bigwedge_{p \in Ptr^{lb}} \mathcal{T}_p^S(d_p). \tag{3.14}$$

The template is a conjunction of *template rows* $\mathcal{T}_p^S$ where each row corresponds to a single loop-back pointer p and it describes the points-to relation of that pointer. The parameter $d_p \subseteq Addr$ of the row (i.e., the *abstract value of the row*) specifies the set of all addresses from the set $Addr$ that p may point to at the end of the corresponding loop. The template row can be therefore expressed as a disjunction of equalities between the loop-back pointer and all possible addresses:

$$\mathcal{T}_p^S(d_p) \equiv (\bigvee_{a \in d_p} p = a) \tag{3.15}$$

Computing an invariant in the given abstract domain allows 2LS to characterize the shape of the program heap. For example, abstract values of template rows corresponding to pointer fields of abstract dynamic objects describe linked paths in the heap, such as linked segments.

Running Example

Let us illustrate usage of the shape domain on the example from Figure 2.5. The program contains four pointers—two variables `list` and `n`, and the `next` fields of the two abstract dynamic objects $ao_{10}^1.next$ and $ao_{10}^2.next$. The variable `n` is local to the loop, hence we do not compute an invariant for it. The remaining three pointers are all updated in the outer loop, therefore we compute invariants for their loop-back variants $list_{16}^{lb}$, $ao_{10}^1.next_{16}^{lb}$, and $ao_{10}^2.next_{16}^{lb}$. For simplicity, we present only the final invariants for $list_{16}^{lb}$ and $ao_{10}^1.next_{16}^{lb}$:

$$\begin{aligned} g_9 \wedge g_{16}^{ls} &\Rightarrow list_{16}^{lb} = \&ao_{10}^1 \vee list_{16}^{lb} = \&ao_{10}^2 \ \wedge \\ g_9 \wedge g_{16}^{ls} \wedge g_{10}^{os} &\Rightarrow ao_{10}^1.next_{16}^{lb} = \&ao_{10}^1 \vee ao_{10}^1.next_{16}^{lb} = \&ao_{10}^2 \vee ao_{10}^1.next_{16}^{lb} = \texttt{null}. \end{aligned} \tag{3.16}$$

The first conjunct (row) of the invariant states that, at the end of the outer loop, the `list` variable points to one of the newly allocated dynamic objects. The second row states that if ao_{10}^1 is allocated (g_{10}^{os} holds), its *next* field points either to one of the dynamic objects or to `null`. The invariant for $ao_{10}^2.next_{16}^{lb}$ is analogous to the second row of Eq. (3.16) with $ao_{10}^2.next_{16}^{lb}$ used on the left-hand side of equalities and $\neg g_{10}^{os}$ used instead of g_{10}^{os}. The overall invariant effectively describes a linked list pointed by the variable `list`, composed of dynamic objects allocated on line 10, and terminated by `null`.

3.4 Abstract Domain Combinations

One of the main advantages of program verification implemented in 2LS is that all abstract domains are required to have a common form of *templates*—quantifier-free first order formulae. Thanks to this feature, it is quite straightforward to create various compositions of different domains while relying on the solver to do the heavy-lifting on the domain operators combination and on mutual reduction of the domain abstract values. In this section, we introduce two approaches to domain combinations: *product templates* and *power templates*,

particularly their form called *templates with symbolic paths* [85]. In addition, our array domain introduced in the following chapter has also the form of a combination domain since it contains an inner domain for describing properties of the array elements.

3.4.1 Product Templates

Product templates are one of the simplest forms of abstract domain combination in 2LS. They are based on using a *Cartesian product template* that combines domains of various kinds side-by-side. This can be achieved by simply taking a conjunction of their templates.

Product templates are particularly helpful for programs that require multiple different domains for different objects. An example of such a combination is the combination of the shape and polyhedra abstract domains. It allows 2LS to analyse values of variables of pointer and numerical type at the same time. This helps not only for analysing programs manipulating pointers and numbers at the same time but also opens a possibility to reason about the contents of data structures on the program heap.

Running Example

Our running example from Figure 2.5 contains memory objects of integer, pointer, and array types. Since 2LS requires a single template, we use the product domain composed from multiple domains to analyse the entire program:

$$\mathcal{T} = \mathcal{T}^I \times \mathcal{T}^S \times \mathcal{T}^A(\mathcal{T}^I). \tag{3.17}$$

$\mathcal{T}^I$ denotes the template for the interval domain and it is used for the variable `i` as well as for the `size` fields of the abstract dynamic objects. The template $\mathcal{T}^S$ for the shape domain is used for the variable `list` and for the `next` fields of the abstract dynamic objects. Last, the template $\mathcal{T}^A$ for the array domain is used for the `data` fields of the abstract dynamic objects. The array domain is designed as a composition domain and it features an inner domain which is used to describe values held by the array. In this case, since the values are numerical, we use the interval domain template $\mathcal{T}^I$. The array domain is described in more detail in Chapter 4, cf. Section 4.4 to see what the array invariant for the running example looks like.

Overall, using all these domains together will allow 2LS to reason about the unrolled linked list—the shape domain will take care of the list node linkage on the heap, the array domain will analyse the array contents of the nodes, and the interval domain will be used to reason about the sizes of nodes and actual values stored inside the arrays. Since all domains are represented using templates (i.e., first-order formulae), the underlying solver will take care of the necessary domain combination operations.

3.4.2 Templates with Symbolic Paths

Using simple templates of invariants, such as the described polyhedra templates or the heap shape template, may not be precise enough to analyse some programs, especially programs working with abstract dynamic objects. In such programs, it is often required that an invariant computed for a loop l distinguishes which loops were or were not executed before reaching l. When working with abstract dynamic objects allocated in loops, this allows one to distinguish situations when an abstract dynamic object does not represent any really allocated object, and therefore an invariant describing it is not valid.

$Inv^L \leftarrow true$
for $\pi \in \Pi$ **do**
  add $assert(\pi)$ to solver
  $Inv_\pi \leftarrow$ compute invariant from $\mathcal{T}$
  remove assert(π) from solver
  if $(\pi \wedge Inv_\pi)$ is satisfiable **then**
    $Inv^L \leftarrow Inv^L \wedge (\pi \implies Inv_\pi)$
  end if
end for

Figure 3.1: Algorithm for inferring an invariant from a symbolic paths template.

With respect to this, in order to improve precision, we have proposed in [85] the concept of *symbolic loop paths*. A symbolic loop path expresses which loops in the program were executed. Since 2LS uses loop-select guards to capture the control flow through the loops, a symbolic path is simply a conjunction of loop-select guard literals.

Formally, let G^{ls} be the set of all loop-select guards in the analysed program. A symbolic loop path π is defined as

$$\pi = \bigwedge_{g \in G^{ls}} l_g \tag{3.18}$$

where l_g is a literal of g, i.e., g or $\neg g$. We denote the set of all symbolic paths by Π. We also define a special path $\pi_\bot$ containing negative literals only. For this path, no loop invariant is computed since no loops were executed for this path.

Having some template $\mathcal{T}$ expressing an abstract domain (e.g., the shape domain or even a product domain), we define the corresponding template with symbolic paths as

$$\mathcal{T}^L \equiv \bigwedge_{\pi \in \Pi} \pi \implies \mathcal{T}. \tag{3.19}$$

This template can be viewed as a *power template*—in the sense of power domains—which assigns to each element of the base domain an element of the exponent domain.

The algorithm for inference of an invariant Inv^L from a symbolic path template is shown in Figure 3.1. The algorithm computes a separate invariant Inv_π in the inner domain (using the template $\mathcal{T}$) for each symbolic path π. To limit the invariant computation for π only, we assert that π (a formula expressing which loops are executed) holds during the computation. It may also happen that, after computing Inv_π, the symbolic path is in fact not reachable in the analysed program. Therefore, we check its reachability by solving $\pi \wedge Inv_\pi$ in the context of the formula generated from the SSA of the analysed program. If π is reachable, then $\pi \implies Inv_\pi$ is conjoined into the resulting invariant, otherwise Inv_π is discarded.

3.5 Memory Safety Verification

When dealing with complex data structures stored in the memory, one of the most necessary and challenging properties to verify is safety from memory errors. Such errors include dereferencing a `null` or a freed pointer, `free` errors, and memory leaks. In order to analyse memory safety, we introduce automatic instrumentation of the analysed program by assertions to check for typical memory safety errors [85, 86]. Verification of such assertions by using the abstract domain introduced in Section 3.3 allows 2LS to discover

occurrences of memory safety errors as well as to prove their absence. In the rest of this chapter, we describe the structure of the given assertions.

3.5.1 Safety from Dereferencing/Freeing a `null` Pointer

To check for this kind of errors, 2LS adds an assertion $p \neq \texttt{null}$ to each program location where $*p$ or `free(p)` occurs. Since the shape domain over-approximates the points-to relation, it is possible to soundly prove absence of such errors. If an error is found, BMC can be used to check whether it is spurious.

3.5.2 Safety from Dereferencing/Freeing a Freed Pointer

We introduce a special variable fr initialised to `null` that is used to track the possibly freed objects. Every call to `free(p)` in a program location i is replaced by a formula

$$fr_i = g_i^{fr} ? p_j : fr_k \tag{3.20}$$

where p_j and fr_k are relevant versions of p and fr, respectively, valid at i, and g_i^{fr} is a free Boolean variable. This formula represents a non-deterministic update of the value of fr by the freed address.

The shape domain (cf. Section 3.3) is then used to over-approximate the set of all addresses that fr may point to, which is essentially the set of all possibly freed memory objects. Proving `free` safety is then done by adding an assertion $p \neq fr$ at each program location where $*p$ or `free(p)` occurs.

The nature of the shape domain guarantees soundness of this approach, however, using it for abstract dynamic objects is often very imprecise. This is because freeing one of the concrete objects represented by the abstract one does not mean that the rest of the represented objects cannot be safely dereferenced or freed. This problem is resolved by modifying the representation of `malloc` calls described in Section 3.2.

In addition to the set AO_i of abstract dynamic objects used to represent all objects allocated at i, we add one object ao_i^{co} to AO_i. The object can be non-deterministically chosen as the malloc result (just like any other ao_i^k), however, it is guaranteed to represent a concrete object (i.e., it can be allocated only once). This is achieved by an additional condition asserting that ao_i^{co} cannot be allocated if there is a pointer pointing to it at the entry to the allocation site i. Therefore, the `malloc` representation has the form

$$p_i = (g_{i,co}^{os} \wedge \bigwedge_{p \in Ptr} p \neq \&ao_i^{co}) \,?\, \&ao_i^{co} : (g_{i,1}^{os} \,?\, \&ao_i^1 : (\dots)) \tag{3.21}$$

Afterwards, it is only allowed to assign the address of the concrete object ao_i^{co} to fr at each allocation site i. Checks for the free safety are also done on concrete objects only, which helps to avoid the described imprecision. This approach remains sound since ao_i^{co} represents an arbitrary object allocated at i, and if safety can be proven for it, it can be assumed to hold for all objects allocated at i.

3.5.3 Memory Leaks Safety

Similarly to the previous section, the variable fr is used to check for safety from memory leaks. At the end of the program, we check whether there is an object ao_i^{co} such that $fr \neq \&ao_i^{co}$. If such an object is found, a memory leak is present. However, proving absence

from memory leaks is only possible for loop-free programs (or for programs with all loops fully unwound). This is because we do not track sequencing of abstract objects representing concrete objects allocated at a single allocation site, and our analysis typically sees that ao_i^{co} can be skipped in deallocation loops, and hence remains inconclusive on the memory leaks.

3.6 Related Work

Shape analysis is a technique of static analysis aimed at discovering shapes of data structures dynamically allocated on the program heap. Such structures usually include various forms of linked lists (singly or doubly linked, circular, nested, etc.), trees, or more complicated structures such as skip-lists.

Unlike stack and static memory that can be abstracted by a finite set of named variables occurring in the analysed program, heap data is potentially unbounded and seemingly arbitrary. This poses a challenge in terms of used heap abstractions and makes shape analysis a widely explored research topic. In this section, we give an overview of some of the current approaches to shape analysis. For a more complete survey, we refer to [64].

We split the described methods into multiple groups based on the models they use to abstract the shape of the heap. The first two groups, using namely various kinds of logics, automata, and graphs are store-based, i.e., they describe the heap explicitely. On the contrary, the approaches in last group are inspired by storeless semantics.

3.6.1 Logic-based Methods

One of the first approaches to shape analysis is based on a so-called *three-value logic* [113]. This logic introduces a new value *unknown* to the traditional boolean values *true* and *false*. The approach is based on abstract interpretation and the value *unknown* is used to express the fact that some elements may or may not be in a relation after an abstraction is done. The method is rather generic but usually requires some manual intervention to be sufficiently scalable.

Another approach uses a so-called *Pointer Assertion Logic* [98] to verify data structures that can be described by graph types. The technique is highly modular, however, it is semi-automated only—it requires explicit loop and function call invariants.

A different group of shape analysis techniques uses *separation logic* [111]. It is an extension of *Hoare logic* [53] developed specifically for reasoning about programs manipulating heap. It builds on *Hoare triples*, which is a mechanism to describe how a program state changes after execution of a piece of code. A triple has a form $\{P\}C\{Q\}$, where P, Q are predicates (often called a *precondition* and a *postcondition*) and C is a command. The predicate P is assumed to hold before the execution of C, and Q is assumed to hold afterwards. Separation logic extends the predicate logic by several new operators and symbols: *emp* (a constant representing an empty heap), $e \mapsto e'$ (an operator expressing the fact that the heap contains a single cell at address e which maps to the value e'), $p * q$ (an operator expressing that the heap can be separated into two parts where p holds for one and q holds for the other), and $p \mathrel{-\!\!*} q$ (an operator expressing that if the heap is extended by a disjoint part in which p holds, then q will hold after the extension). There are many fully-automated tools based on separation logic such as Space Invader [121] and SLAyer [7].

Separation logic is also applied in practice, e.g., in combination with *bi-abduction* [22] in the Infer tool (https://fbinfer.com). However, the tool misses support for low-level

features such as block operations and advanced pointer arithmetic and handles simple data structures only (mainly lists). To expand bi-abduction to more complicated structures, *second-order bi-abduction* has been proposed [77, 33]. While it is able to learn recursive heap predicates, finding a solution of the generated equations is hard and the authors propose heuristics for certain shape of equations only. Bi-abduction has also been extended to handle low-level features of code [55], however, combination with other analyses (e.g., for analysing contents of linked lists) and scalability of the approach are so far quite limited.

Separation logic and bi-abduction are also among the main techniques used in the Gillian framework [43, 84], which allows to develop custom *compositional symbolic analysis* tools. Authors use the framework to create Gillian-C, a proof-of-concept tool for symbolic testing of C programs which, compared to our approach, is aimed at finding bugs rather than proving program correctness. The same is the case for some recent works which introduce and apply so-called *incorrectness separation logic* [106, 78] to find bugs in real-world programs.

Another verifier based on separation logic is $\texttt{S2}_{\texttt{td}}$ [79]. It implements a satisfiability procedure for separation logic extended by user-defined inductive predicates. Although it is able to handle very complex data structures (e.g., trees with linked leaves), it is rather fragile and can fail on quite simple structures if they are not handled by the program in a suitable way. Moreover, a support for combinations with other abstract domains is not very advanced and the approach has a problem with reliable diagnostics of discovered errors.

Also, more recently, automation of separation logic using SMT solvers by reduction to effectively propositional logic has been proposed by [104, 59, 60].

3.6.2 Methods Using Automata and Graphs

Another group of shape analysis tools describes the state of the heap using various forms of automata and graphs. One of such tools is Predator [38] which uses *symbolic memory graphs (SMGs)* [76]. These are designed as an abstract domain for the framework of abstract interpretation. SMGs model the heap with byte-precision and use summary nodes to represent abstractions of linked lists of unbounded length. They are designed to handle low-level manipulation of dynamic data structures. The usability of the approach is confirmed by mutliple wins in the heap-related categories of the International Competition on Software Verification (SV-COMP). However, the approach is missing a combination with other abstract data domains.

A different approach based on graphs uses *tree automata* and *regular tree model checking* [17] and is implemented, e.g., in the Forester tool [50]. The approach uses automata over words and trees to describe the shape of the heap and a tree-automata-based abstraction to over-approxmate the set of reachable heap configurations. The abstraction can be refined by counterexample-guided refinement. Combining these approaches with reasoning about value properties is not easy as shown in the works [1, 54] that extended Forester with reasoning about finite data and a specialised support for handling ordered list segments.

3.6.3 Methods Using Storeless Semantics

All of the above approaches are store-based, i.e. they explicitly describe the state of the heap using some logic or graphs. On the contrary, methods based on *storeless semantics* [63] use pointer access paths to describe reachable shapes on the heap [28, 112, 96, 18]. A *pointer access path* does not concretely express the heap state, it only describes which dynamic objects are reachable from a pointer. Using a set of access paths for each pointer, one can

efficiently describe the shape of (the reachable part of) the heap. These approaches usually use abstract interpretation over control-flow graphs and their support for dealing with the data content is limited [96]. However, pointer access paths proved the most suitable for our purposes and our work is heavily inspired by them.

Chapter 4

Verification of Array-Manipulating Programs

Arrays are one of the most widely used data structures in software in general. In this section, we introduce our abstract domain for analysing the contents of arrays—we refer to it as *the array domain*.

Similarly to all other domains in 2LS, the array domain has the form of a template. An important property of arrays is that they may have an arbitrary element type. Therefore, using a simple domain is not sufficient and our array domain is a so-called *combination domain* where the domain itself describes only the form (the memory layout) of the array and it delegates reasoning about the actual array element values to another abstract domain, which we denote as the *inner domain*. Note that the inner domain may be any domain present in 2LS, including the array domain itself, which can be used to analyse arrays with multiple dimensions.

The domain is limited to the set Arr of array-typed variables. In particular, since we deal with loop invariants, we concentrate on arrays that are updated inside loops. In our SSA representation described in Section 2.1.1, such arrays are abstracted using so-called loop-back array variables. We denote Arr^{lb} the set of such variables, and our array domain is then limited to this set.

The primary idea of our array domain is that each array $a \in Arr^{lb}$ is split into several *segments*, and an invariant is computed for each segment in the appropriate inner domain (based on the element type of a). The set of segments is for each a determined using so-called *segment borders* that we infer at the beginning of the analysis (using the set of index expressions that the analysed program uses to write into a—cf. Section 4.2 for details). A segment border can be any valid SSA expression.

In the rest of this section, we describe different aspects of the array abstract domain and invariant inference using it. First, we show in Section 4.1 how, given a set of borders, an array is split into segments and what the array domain template looks like. Next, we introduce the way we determine array segment borders in Section 4.2. Last, in Section 4.3, we present how invariants are computed in the array domain and how they can be used to verify program properties. To facilitate understanding of the presented concepts, we illustrate all of them on our running example in Section 4.4.

In the rest of this chapter, let us assume that we compute a loop invariant for an array $a \in arr^{lb}$ updated in a loop l. We use N_a to denote the size (number of elements) of a.

4.1 Array Domain Template

We now describe the form of the array domain template. As outlined before, each array is split into multiple segments and an invariant for each segment is computed in the array inner domain. Hence, the form of the array template is given by *array segmentation*, i.e., the way that each array is split into segments.

Let us denote B_a the set of segment borders for the array a. Prior to creating the segments, we perform two pre-processing steps: (1) making the borders unique and (2) ordering the borders.

Making the Borders Unique In order to decrease the number of segment borders and avoid empty segments, we first remove duplicate borders. This is done using an SMT solver, in particular, for each pair of segment borders $b_1, b_2 \in B_a$, we check if the formula:

$$b_1 \neq b_2 \wedge SSA \tag{4.1}$$

is satisfiable. In Eq. (4.1), SSA denotes the formula created from the SSA form of the analysed program and representing the (over-approximated) program semantics. If Eq. (4.1) is unsatisfiable, then the values of the borders are always equal and hence one of them can be removed from B_a. Repeating this process for each pair of borders, we obtain the set of unique borders.

Ordering Borders After making B_a contain unique indices only, we try to order them. Again, we query the SMT solver, this time using two formulae:

$$\neg(b_1 \leq b_2) \wedge SSA \tag{4.2}$$
$$\neg(b_2 \leq b_1) \wedge SSA. \tag{4.3}$$

If exactly one of Equations (4.2) and (4.3) is unsatisfiable for each pair of $b_1, b_2 \in B_a$, then a total ordering over B_a can be found. Otherwise, B_a is left unordered.

Array Segmentation Once the arrays are unique and possibly ordered, we create the array segmentation. We distinguish two situations:

1. B_a cannot be totally ordered. In such a case, we create multiple segmentations, one for each $b \in B_a$:
$$\{0\}\ S_1^b\ \{b\}\ S_2^b\ \{b+1\}\ S_3^b\ \{N_a\}. \tag{4.4}$$
The idea here is that if $a[b]$ is written to in a loop with gradually incrementing b, for any iteration of the loop, S_1^b will abstract all array elements that were already traversed, S_2^b will be the element accessed in the current iteration, and S_3^b will abstract elements to be traversed in the following iterations.

2. B_a can be totally ordered s.t. $b_1 \leq \cdots \leq b_n$. In such a case, we create a single segmentation for the entire array a:
$$\{0\}\ S_1\ \{b_1\}\ S_2\ \{b_1+1\}\ \cdots\ \{b_n\}\ S_{2n}\ \{b_n+1\}\ S_{2n+1}\ \{N_a\}. \tag{4.5}$$

A single array segment S denoted
$$\{b_l\}\ S\ \{b_u\} \tag{4.6}$$

is an abstraction of the elements of a between the indices b_l (inclusive) and b_u (exclusive). We refer to b_l and b_u as to the *lower* and *upper segment bounds*, respectively. In addition, for each S, we define two special variables: (1) the *segment element variable* $elem^S$ being an abstraction of the array elements contained in S and (2) the *segment index variable* idx^S being an abstraction of the indices of the array elements contained in S.

Template Form Having the set of loop-back arrays Arr^{lb} and a set of segments S^a for each $a \in Arr^{lb}$, we define the array domain template as:

$$\mathcal{T}^A \equiv \bigwedge_{a \in Arr^{lb}} \bigwedge_{S \in S^a} \left(G^S \Rightarrow \mathcal{T}^{in}(elem^S)\right) \tag{4.7}$$

where $\mathcal{T}^{in}$ is the inner domain template and G^S is the conjunction of guards associated with the segment S.

The inner domain is typically chosen based on the data type of $elem^S$ (e.g., we usually use the interval domain for numerical types and the shape domain for pointer types).

The purpose of G^S is to make sure that the inner invariant is limited to the elements of the given segment $\{b_l\}\ S\ \{b_u\}$. In particular, G^S is a conjunction of several guards:

$$b_l \leq idx^S < b_u \wedge \tag{4.8}$$

$$0 \leq idx^S < N_a \wedge \tag{4.9}$$

$$elem^S = a[idx^S] \tag{4.10}$$

where Eq. (4.8) makes sure that the segment index variable stays between the segment borders, Eq. (4.9) makes sure that the segment index variable stays between the array borders (since segment borders are generic expressions, they may lay outside of the array, hence Eq. (4.8) is not sufficient), and Eq. (4.10) binds the segment element variable with the segment index variable.

Using the above template, 2LS is able to compute a different invariant for each segment. For example, for a typical array iteration loop, this would allow to infer a different invariant for the part of the array that has already been traversed than for the part of the array that is still to be visited.

4.2 Computing Array Segment Borders

In the previous section, we assumed that we already have the set of segment borders for each array. In this section, we describe how this set is obtained. As we outlined earlier, the verification approach of 2LS requires the domain template to be a fixed, parametrized, first-order formula. To be able to fulfil the "fixed" property, we need to determine the set of segments at the beginning of the analysis so that we are able to create a finite set of array segments which will form the array domain template.

The main idea of our approach is that the segment borders should be closely related to the expressions that are used to access array elements in the analysed program (we denote these as *array index expressions*). Therefore, we perform a static *array index analysis* which collects the set of all expressions occurring as array access indices (i.e., expressions that appear inside the square bracket operators). In addition, we distinguish between *read* accesses (occurring on the right-hand side of assignments and in conditions) and *write* accesses (occurring on the left-hand side of assignments).

Once the array index analysis is complete, for each loop-back array a, we determine the set of its segment borders by taking the set of all index expressions used to write into a in the corresponding loop. In addition, if some of those expressions contain a variable whose value is updated inside the same loop, we also take the pre-loop value of the expression as a segment border.

To illustrate, let us have a simple loop initializing the second half of an array:

```
1 for (int i = N / 2; i < N; i++)
2     a[i] = 0;
```

The set of index expressions used to write into the array is $\{\texttt{i}\}$, but we would also use `N/2` (the initial value of `i`) as a segment border. Hence, the segmentation of `a` would be:

$$\{0\} \cdots \{\texttt{N/2}\} \cdots \{\texttt{N/2}+1\} \cdots \{\texttt{i}\} \cdots \{\texttt{i}+1\} \cdots \{\texttt{N}\} \tag{4.11}$$

Thanks to this segmentation, 2LS is able to differentiate three important parts of the array: (1) the first half of the array (which is untouched in the loop), (2) the part between `N/2` and `i` which in any iteration represents the already initialized part, and (3) the part from `i` to the array end which represents the part to be initialized in future iterations. In particular, 2LS would be able to infer an invariant stating that all elements in Part (2) are equal to 0, which would mean that the entire second half of the array is set to 0 after the loop ends.

4.3 Array Domain Invariant Inference

Once the array domain template is created, the invariant inference algorithm of 2LS is used to compute loop invariants for individual segments. As we already described, most of the work is delegated to the inner domain, and the array domain is mainly responsible for making sure that the segment element variables, for which the inner invariants are computed, are properly constrained.

Additionally, there is one more necessary step after the array invariants are computed. The problem is that the invariants describe properties of the segment element and index variables, however, these variables are not actually used inside the analysed program. Therefore, in order for the invariant to properly constrain the program semantics, we *bind* the computed invariants with all index expressions used to read from the arrays. We do not need to constrain the array elements that are written by the program since their value gets overridden, hence binding with read elements is sufficient. The set of expressions to bind the invariant with is obtained using the array index analysis introduced in Section 4.2.

In particular, for each segment S of each loop-back array a, we create a binding between the segment element and index variables $elem^S$ and idx^S and each index expression i_r used to read from a as follows. We take the computed invariant for S and replace all occurrences of idx^S by i_r and all occurrences of $elem^S$ by $a[i_r]$. Then, the obtained formula is passed to the solver which constrains the values of a for the given access through i_r. This process is done for the final invariant as well as for each candidate invariant found during the analysis to allow the invariant inference algorithm account with the already computed constraints.

4.4 Running Example

We now illustrate usage of the array domain on the running example from Figure 2.5. The program features two array-typed objects—the `data` elements of the abstract dynamic objects ao^1_{10} and ao^2_{10}. Similarly to the shape domain example, we only use ao^1_{10} throughout the example as the invariant for ao^2_{10} is analogous. We demonstrate inference of a loop invariant for the inner loop of the program, hence the SSA object that we work with is $ao^1_{10}.data^{lb}_{12}$.

First, 2LS runs the array index analysis to determine the set of indices used to access the array. In this case, there is a single index used for writing (`i` on line 12) and one index used for reading (`x` on line 18).

After the array index analysis is run, the analysed array must be segmented. There is a single written index, hence there will be three segments in total. In addition, the index is updated inside the same loop, hence we will use its loop-back variant (i^{lb}_{12}) inside the segmentation:

$$\{0\}\ S_1\ \{i^{lb}_{12}\}\ S_2\ \{i^{lb}_{12}+1\}\ S_3\ \{1000\}. \tag{4.12}$$

For each segment S_i, we introduce a segment element variable $elem^i$ and a segment index variable idx^i.

Since the array is of integer type, we will use the interval abstract domain as the inner domain. The interval domain has two template rows for each variable, hence our template will contain 6 rows in total (two for each segment element variable). For the sake of legibility, we only give the two rows for $elem^1$:

$$\begin{aligned} g_{11} \wedge g^{ls}_{12} \wedge g^{os}_{10}\ \wedge\ 0 \le idx^1 < i^{lb}_{12}\ \wedge\ 0 \le idx^1 < 1000\ \wedge\ elem^1 = ao^1_{10}.data^{lb}_{12}[idx^1] \\ \Rightarrow elem^1 \le d_1\ \wedge \\ g_{11} \wedge g^{ls}_{12} \wedge g^{os}_{10}\ \wedge\ 0 \le idx^1 < i^{lb}_{12}\ \wedge\ 0 \le idx^1 < 1000\ \wedge\ elem^1 = ao^1_{10}.data^{lb}_{12}[idx^1] \\ \Rightarrow -elem^1 \le d_2. \end{aligned} \tag{4.13}$$

Both rows have the same guard (the implication antecedent) consisting of multiple parts:

- The first three conjuncts ($g_{11} \wedge g^{ls}_{12} \wedge g^{os}_{10}$) are standard row guards used in other domains that guard the reachability of the loop, the definition of the loop-back variable, and the allocation of the corresponding abstract dynamic object.
- The second part ($0 \le idx^1 < i^{lb}_{12}$) guards that the segment index variable stays within the segment bounds.
- The third part ($0 \le idx^1 < 1000$) guards that the segment index variable stays within the array bounds.
- The last part ($elem^1 = ao^1_{10}.data^{lb}_{12}[idx^1]$) binds the segment element variable with the analysed array object and the segment index variable.

The actual properties to be computed (the implication consequences) are determined from the inner domain, in this case the interval abstract domain.

Using the above template in the invariant inference algorithm of 2LS, we will obtain values of $d_1 = d_2 = 0$ (i.e., values of all array elements in the segment S_1 are equal to 0). After the loop ends, the value of i^{lb}_{12} will be 1000 (thanks to the invariant computed for i^{lb}_{12} in Eq. (2.20)), which will effectively prove that all elements of the given array are equal to 0 at that moment.

The remaining part of the array domain usage is binding of the invariant onto indices used to read from it. Our example program features one array read at line 18 (`list->data[x]`). At this point of the program, `list` may point to the dynamic object ao^1_{10}, hence an access to $ao^1_{10}.data$ is possible in this expression. Therefore, we bind the invariant computed from the template from Eq. (4.13) with the read index x_{17} as follows:

$$\begin{aligned} g_{11} \wedge g^{ls}_{12} \wedge g^{os}_{10} \ \wedge \ 0 \leq x_{17} < i^{lb}_{12} \ \wedge \ 0 \leq x_{17} < 1000 \Rightarrow ao^1_{10}.data^{lb}_{12}[x_{17}] \leq 0 \ \wedge \\ g_{11} \wedge g^{ls}_{12} \wedge g^{os}_{10} \ \wedge \ 0 \leq x_{17} < i^{lb}_{12} \ \wedge \ 0 \leq x_{17} < 1000 \Rightarrow -ao^1_{10}.data^{lb}_{12}[x_{17}] \leq 0. \end{aligned} \tag{4.14}$$

The equation has been obtained from Eq. (4.13) by supplying the actual computed values of d_1 and d_2 and by replacing occurrences of idx^1 by x_{17} and the occurrences of $elem^1$ by $ao^1_{10}.data^{lb}_{12}[x_{17}]$. We removed the last part of each row guard as $elem^1$ is no longer used. Also, note that we bind the invariant to $ao^1_{10}.data^{lb}_{12}$ rather than to the SSA version of $ao^1_{10}.data$ valid on line 18 (which is $ao^1_{10}.data^{phi}_{9}$, cf. Eq. (3.8) and Eq. (3.9) for exact formula representing line 18) because we only want to constrain the value of the array coming from the loop that the invariant is computed for. In other words, Eq. (4.14) effectively allows to leverage the computed array invariant during further program analysis.

As we already mentioned, the last step would be done after each round of the invariant inference algorithm, however, we omit that here for the sake of simplicity and give the binding for the final invariant only.

4.5 Related Work

Similarly to shape analysis, there exists a vast body of works aimed at analysis of array contents and verification of programs manipulating arrays. We describe the most important works in this section. Many of the works are related and use a similar principle, hence we divide the overview into three categories.

4.5.1 Methods Based on Array Segmentation

One of the first works in the area [16] introduced two basic techniques for reasoning about the contents of arrays: (1) *array expansion* where each array element is represented using a single abstract value and (2) *array smashing* (also presented in [46]) where all elements of the array are abstracted using a single value. These approaches represent two extremes in approaching the arrays—while the first one often does not scale due to unbounded nature of the arrays, the second one abstracts away too much information that is often crucial for proving the required array properties.

An approach to overcome these problems, which we also take in our work, is to split arrays into multiple parts, usually called segments. This technique was first introduced in [47] where it was combined with simple numerical domains and was mainly able to reason about array initialization loops. This method was improved in [51] by extending it to handle relational abstract properties and consequently in [32] which proposed to use an arbitrary abstract domain for reasoning about array elements. Our approach is heavily based on the latter work, mainly due to the fact that it is compatible with the verification approach in 2LS. The proposed method uses automatic inference of segment bounds based on semantic pre-analysis of the array usage in the program.

Compared to all of these works, which are mainly aimed at analysis of numerical contents of arrays, we leverage other domains present in 2LS. In particular, the combination with

our newly introduced shape domain allows us to expand program verification to analysis of structures combining arrays and linked structures on the heap. In addition, it was necessary to formulate our approach in the very specific 2LS framework, hence the introduced domain has several unique features that cannot be found in other approaches.

The segmentation-based approaches were further extended to non-contiguous and overlapping segments [80, 23] but these are much more difficult to be described using first-order logic formulae, and therefore we did not consider them for our approach.

4.5.2 Methods Based on Analysis of Array-Manipulating Loops

A completely different approach to the typical array verification problems is taken by the VeriAbs verification tool [2]. Instead of trying to describe the contents of (potentially huge) arrays in an abstract way, the tool focuses on analysis of loops manipulating the arrays. In particular, VeriAbs features two imporant techniques related to verification of array contents: (1) *loop shrinking* [70] and (2) *full-program induction* [24].

The first technique automatically analyses loops that manipulate program arrays and for each loop it determines the so-called *shrink factor* k—the sufficient number of iterations that are necessary to prove the property being checked. After k is obtained, the processed array is reduced to the size k and filled with k non-deterministically chosen elements of the original array. The reduced program is then verified using state-of-the-art BMC tools.

In some cases, the shrink factor is not sufficiently low for BMC to scale and prove or refute program correctness. In such a case, VeriAbs transorms the arrays to be of symbolic size N and performs so-called full-program induction. This technique, given a program P_N parametrized by the array size N and pre- and post-conditions denoted $\varphi(N)$ and $\psi(M)$, respectively, is able to very efficiently check validity of the Hoare triple $\{\varphi(N)\ P_N\ \psi(N)\}$ for all values of $N > 0$.

The above methods have proven very effective since VeriAbs has been the most sucessful tool in the *ReachSafety-Arrays* sub-category of SV-COMP in the recent years [9, 10, 11]. On the other hand, VeriAbs does not compete in memory safety, so the effectiveness of these methods on programs combining arrays and linked structures remains questionable. Still, we may consider implementing modified versions of the proposed techniques in future to improve efficiency of verification of array-manipulating programs in 2LS.

4.5.3 Predicate Abstraction and Non-Automatic Methods

In the last group of works, we present those that use completely different verification approaches than 2LS does and hence were not considered for our case.

First, there is a large group of works [73, 72] based on predicate abstraction [41], possibly improved with counter-example guided refinement [14] and Craig interpolants [62]. These, however, make use of the property to be proved while our approach aims at discovering (previously unknown) existing properties.

Second, besides fully automated works, there exist approaches which require some user intervention. For instance, [48] also specifies abstract domains using templates, but their domains are universally quantified (as opposed to our quantifier-free templates). This naturally makes the domains much stronger, however, the verification approach requires all the abstract domains to be specified manually. In contrary, the verification approach of 2LS is fully automatic. Other techniques based on deductive methods [6, 25] suffer from a similar issue when they require users to provide loop invariants (which our method is able to infer automatically).

Chapter 5

Experimental Evaluations

We now present results of experimental evaluations of the 2LS framework with all extensions presented in this thesis implemented. Since 2LS regularly competes in the International Competition on Software Verification (SV-COMP)[1], our evaluations are heavily based on benchmarks taken from this competition.

SV-COMP is an annual competition in which verification tools compete against each other. The goal is to verify as many programs as possible while providing the minumum number of false results. In 2022, the C benchmark contained 15 648 tasks (programs) [11]. Since many tools are strong in some areas of verification only, the tasks are divided into several categories and sub-categories which focus on verification of particular kinds of programs and particular properties.

2LS competes in all categories, however, it does not support some kinds of programs (e.g., programs containing recursion), hence it returns the "unknown" result for all benchmarks in the unsupported categories. In order to highlight actual contributions of this thesis, we present results from some categories only. We describe the chosen categories along with the SV-COMP scoring scheme in Section 5.1.

For the selected categories, we present the evolution of scores across the recent years, which demonstrates the effect of our contributions. These are presented in Section 5.2. In addition to comparing 2LS with its previous versions, we highlight the areas in which it is stronger than other verification tools, including those that specialize in verification of programs manipulating data structures. We do this in Section 5.2.3.

While SV-COMP scores can demonstrate some qualities of a tool (e.g., soundness or precision), they certainly do not capture all properties which are necessary for a verification approach to be usable in practice. One of the most important features is the speed of verification as tools often take too much time to verify even simple programs. This is not the case for 2LS, which, thanks to its specific verification approach based on the kIkI algorithm and supported by incremental SAT solving, is generally quicker than its competitors. We demonstrate this on several alternative rankings in Section 5.3.

5.1 SV-COMP Organization and Rules

SV-COMP is held annually since 2012 as a part of the *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2LS has been competing since 2016, although during its first year, it has only participated in categories

[1] https://sv-comp.sosy-lab.org/

dealing with integer-only programs, hence we consider the results from 2017 onwards. In total, our contributions are related to the following categories:

Reachability Safety (ReachSafety) In this category, the goal is to verify reachability of assertions (i.e., whether a condition in a call to the `assert` function always holds, once the program execution reaches the call). This category is divided into many sub-categories, from which we are interested in *ReachSafety-Arrays* and *ReachSafety-Heap* which feature programs whose verification in 2LS requires usage of our new abstract array and shape domains, respectively.

Memory Safety (MemSafety) In this category, the goal is to verify memory safety properties, in particular *validity of dereference operations*, *validity of free operations*, and *absence from memory leaks*. In 2LS, this is done by instrumentation with assertions guarding against memory safety errors which we introduced in Section 3.5. The category is also divided into several sub-categories based on the kinds of data structures used. We are mostly interested in the *MemSafety-Heap* and the *MemSafety-LinkedLists* categories.

Each subcategory in SV-COMP features a number of verification tasks. Each task consists of a single C program, a property to verify, and an expected result. The result claims whether the property holds or not. For each task, the verification tool must provide a verdict and a so-called *verification witness*. The witness is a program trace in special format [13, 12] which contains either a path to the error (if the verifier claims that the property does not hold) or a proof (including an invariant) that the property holds. Witnesses are then checked by a special kind of tools called witness validators.

After the verifier analyses a task, it is awarded a score based on its answer. Since SV-COMP aims at sound and precise approaches, the penalties for incorrect results are much higher than awards for a successful verification. In particular, a tool is granted:

- 1 point if it founds the error in an incorrect program,
- 2 points if it proves that a program is correct,
- -16 points if it reports a spurious error in a correct program (a *false positive*), and
- -32 points if it fails to find an error (a *false negative*).

In the end, the scores are normalized across (sub)categories to obtain the final score. Since we deal with sub-categories only, the normalization procedure (cf. [8] for details) is not relevant for the upcoming presentation.

5.2 Scores of 2LS in SV-COMP

We now present evolution of scores of 2LS in SV-COMP between years 2017 and 2023. We limit ourselves to the categories relevant for extensions proposed and implemented within this thesis. Since we enhance 2LS with analysis of two kinds of data structures—arrays and linked lists—we present results for the relevant categories separately.

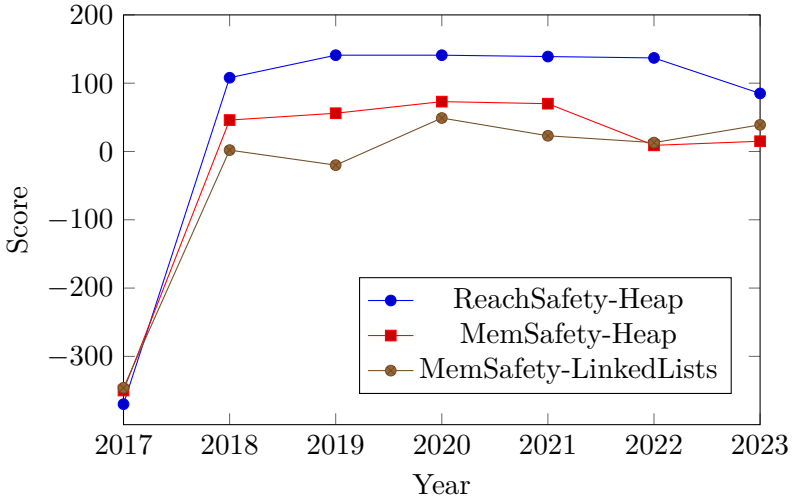


Figure 5.1: Results of 2LS in heap-related categories in SV-COMP 2017-2023

5.2.1 Heap-related Categories

Our proposed shape abstract domain, along with the new memory model and memory safety instrumentation, allows to reason about programs manipulating pointers and data allocated on the heap, especially in the form of linked lists. Programs of this kind occur in SV-COMP in three categories: *ReachSafety-Heap*, *MemSafety-Heap*, and *MemSafety-LinkedLists*. Figure 5.1 shows evolution of 2LS' scores in these categories between years 2017 and 2023 of SV-COMP.

Most of the mentioned contributions were introduced in 2018 [88], where we can observe a major improvement of the score. For 2019 and 2020, we improved our memory safety instrumentation [90], which again raised the scores in memory safety categories. Unfortunately, we can also observe some drops in the score in these categories (especially in 2022 and 2023) which were caused (1) by new false results that we were not able to fix in time before the competition and (2) by last-minute removal of some tasks in 2023 which makes comparison with this year less relevant. Nonetheless, the results show that our contributions heavily improve the capabilities of 2LS in terms of analysis of programs manipulating heap-allocated structures.

5.2.2 Array-related Categories

Our second contribution, the array abstract domain, is naturally useful for programs manipulating arrays. Units of such programs occur in many (sub-)categories of SV-COMP, however, the most relevant category for us is *ReachSafety-Arrays* which is entirely dedicated to programs with arrays.

2LS has traditionally received negative score in this category (-28 in 2020–2022) which has only changed in 2023 with the introduction of our proposed domain into 2LS [89]. Using the new array domain, 2LS was able to successfully verify 17 error-free tasks from this category (as opposed to 2 from the previous year)[2].While this is not a large number, it

[2]The given number is different from the official results of 2LS in SV-COMP 2023. The reason is that a number of tasks was last-minute disqualified due to past-deadline changes which were often related to the tasks being added to new categories (e.g., *NoOverflows*) rather than actual modifications of the tasks or

shows that our new domain is a good first step towards better analysis of array contents in 2LS. In addition, due to the nature of program analysis in 2LS, the domain may leverage from combination with other abstract domains (e.g., the shape domain), however, SV-COMP benchmarks do not yet feature tasks requiring such a combination.

5.2.3 Comparison to Other Tools

One of the greatest strengths of our approach is that the proposed domains are easy to be combined with other abstract domains in 2LS, allowing to verify multiple program properties at the same time. In SV-COMP results, this can be observed on a set of 10 tasks which require a combined reasoning about the shape of linked lists of an unbounded length and about the numerical values stored inside the list nodes. These tasks were contributed by our team to the *ReachSafety-Heap* category in 2019 [85] since no such tasks were present at the time.

As of 2023, 2LS remains one of the two tools (the other one being VeriAbs [2, 34]) capable of successfully verifying a majority of the tasks. All other tools, including the best tools in the category, verify at most 1–2 tasks and timeout or report "unknown" on the rest. This demonstrates that our contributions are truly unique and allow to reason about program properties that are beyond capabilities of state-of-the-art verification tools.

5.3 Alternative Rankings

Besides the ability to soundly and correctly verify programs, there are other properties that may support practical usefulness of a verification tool. In this section, we compare 2LS to other verifiers using some alternative metrics, in particular *verification speed*, *energy consumption*, and *correctness rate*.

5.3.1 Speed of Verification in 2LS

One property that can be observed from the SV-COMP results is that 2LS verifies most of the tasks in a very short time, compared to other tools. We support this claim by an experiment where we set a small time limit and then observe how 2LS would compete against other tools in SV-COMP 2022. Table 5.1 shows the position that 2LS would achieve in some of the main categories if the time limit was set to 5 seconds.

Table 5.1: Position of 2LS (with the total number of participating tools) in selected categories of SV-COMP 2022 with a 5 s time limit.

ReachSafety	MemSafety	NoOverflows	Termination	Overall
5. (of 21)	5. (of 14)	1. (of 15)	1. (of 10)	2. (of 13)

The table shows that 2LS would achieve a high position in all of the mentioned categories. Even though not all categories are related to analysis of data structures, many of the concepts proposed in this thesis (e.g., the memory model) are still used and contribute to these results.

their verdicts. Hence, we present results from the entire benchmark instead of the competition benchmark set as those results are more representative and can be better compared to the previous year results.

5.3.2 Power Consumption and Correctness Rate

In recent years, the competition report of SV-COMP [9] provides two *alternative rankings* of verifiers that honor different aspects of the verification process. These are in particular:

- *Correct Verifiers* which ranks the verifiers by a so-called *correctness rate*, which is a ratio of the number of incorrect results and the overall achieved score. 2LS finished third in this ranking in 2020 with only 0.0016 errors per score point.
- *Green Verifiers* which ranks the verifiers by the amount of energy used to achieve a single score point. In 2020 and 2021, 2LS finished second in this ranking by using only 180 J per score point.

The results presented in this section show that for the tasks that 2LS is able to prove, it does that very fast, mainly thanks to the specific approach using the incremental solving with a single SAT instance. All of our proposed extensions are designed in a way to follow this approach and hence contribute to the overall results.

Part II

Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects

Chapter 6

Static Analysis of Semantic Equivalence

The problem of automatically checking semantic equivalence of programs is nowadays a widely studied field of program analysis. Unfortunately, despite a lot of progress, existing approaches for sound equivalence checking often rely on heavy-weight formal methods and consequently have problems with scalability. This limits their usage in the industry despite the fact that there exists a lot of space for their potential applications.

One of such applications is for systems that require some sort of stability and backwards compatibility. These can be, e.g., various standard libraries or kernels of operating systems. One of the best known commercial operating systems is Red Hat Enterprise Linux (RHEL), whose kernel includes a list of functions, a so-called *Kernel Application Binary Interface* (KABI), whose semantics should be stable across the lifetime of a single major release (unless changes are dictated, e.g., by security issues). Ensuring this stability at presence of constant refactoring changes is rather difficult, and any (even partial) automation of the backwards compatibility checking has huge potential to save a lot of human effort and costs.

In this part of the thesis, we propose a novel automated method for verifying whether two versions of a program that should have the same semantics—as one of them is expected to be a refactoring of the other—do indeed have the same semantics (and hence that no error has been done during the refactoring). We aim at applicability to large-scale industrial code, where applicability of our approach to the Linux kernel is particularly important to us, which is motivated by a concrete interest of Red Hat. Our approach is, however, more general and applicable on other C projects too, which we demonstrate by experiments with one of the existing implementations of the C standard library.

General Approach In order to cover all possible behaviours and program paths, we build on *static code analysis*—more precisely, on analysing the LLVM intermediate code representation. To achieve the needed scalability to hundreds of thousands of lines of code, which—to the best of our knowledge—is beyond capabilities of current tools and approaches, we propose to compare the different versions primarily *per-instruction* on the level of their LLVM intermediate code representation. Of course, checking semantic equivalence on the level of single instructions would normally lead to many false non-equivalence results. In order to resolve this problem while retaining scalability, we *pre-process* the code to be compared using various static analyses and code transformations (such as inlining, constant

propagation, redundant code elimination, or dead code elimination) to bring the code to a form that can be compared instruction-by-instruction as often as possible.

Beyond checking per-instruction equivalence, we allow program versions under comparison to differ in ways described by so-called *semantics-preserving change patterns* (SPCPs). Our method is generic in the set of SPCPs to be applied provided they are described in a particular way that we propose. We call such SPCPs as *effective*. In our current implementation of the approach, we support several built-in SPCPs that are inspired by existing lists of refactorings (we particularly consider the list from [44]) and by our own extensive study of refactorings commonly appearing in the Linux kernel code.

Custom Change Patterns Besides scalability, methods of semantic equivalence detection suffer from other problems, too. One of the common ones is that software projects usually need to introduce semantic changes even to the parts which should remain stable. This happens, for example, when fixing bugs and security issues. Such changes are desirable, however, they change the semantics and hence pollute the output of semantic equality checks. Eliminating this problem is quite difficult as the changes are often project-specific: a change that is considered safe for one project needs not to occur or, even worse, may be considered unsafe in another one.

To address this issue, we generalize the notion of semantics-preserving patterns using so-called *custom change patterns (CCPs)* and we introduce a highly scalable method to detect such changes during analysis of semantic equivalence of real-world software. This allows users of our analyser to specify which kinds of differences they wish to ignore (i.e., to consider safe). Our approach to handling CCPs is based on describing code changes through parametrised control-flow graphs and on using a specific graph algorithm to match occurrences of change patterns between a pair of software versions.

Comparing Semantics of Global Variables In addition to checking semantic equality of functions that typically form the API of the project under analysis, our method is also designated for comparing the semantics of global variables that appear in the code (more precisely, of the way the global variables are used within the project). Our interest in the global variables is motivated by them often being used to represent adjustable system parameters whose meaning should be preserved during project updates. To check for semantic equivalence of global variables, we analyse all functions using the given variable, however, we notice that, rather than comparing the entire functions, it is sufficient to compare only those parts that are affected by the variable. To achieve this, we propose a custom algorithm for *program slicing* which removes parts of the functions unaffected by the value of such variables and, subsequently, we compare the remaining parts using our general comparison method.

Plan of Part II of the Thesis In the rest of this chapter, we describe the elementary concepts of our approach—the program representation that we use (based on LLVM IR) and the proposed generic algorithm for efficient checking of semantic equivalence that is prepared to handle built-in SPCPs as well as user-defined CCPs. We describe these two kinds of patterns in Chapters 7 and 8, respectively. After that, Chapter 9 introduces our specialized slicing algorithm for analysis of the semantics of global variables.

We have implemented all of the proposed concepts in a tool called DiffKemp. To demonstrate usefulness of our approach, we perform multiple experiments with DiffKemp and present the results in detail in Chapter 10.

6.1 Program Representation

We represent functions under comparison using *control flow graphs (CFGs)*. In particular, since our tool builds on the LLVM infrastructure, we translate the compared programs into the LLVM intermediate representation (LLVM IR), in which each function can be viewed as a single CFG. The following definition of CFGs is therefore heavily based on the particular notion of CFGs defined by LLVM IR.

A CFG is composed of basic blocks connected by edges representing program branches. A basic block is a list of instructions satisfying the property that all incoming edges are directed to the first instruction and all outgoing edges are directed from the last instruction.

An *instruction* performs an *operation* over a (possibly empty) list of operands and stores its result into a local variable (if it produces some result). An *operand* may be a variable (global or local), a constant, or a function (this is the case, e.g., for `call` instructions, in which the callee is represented as an operand, or for instructions that assign to variables holding pointers to functions). Each CFG satisfies the *static single assignment (SSA)* property requiring that each variable is assigned to at most once. Therefore, each instruction i assigns its result into a fresh local variable that we denote v_i. Assignments into global variables are done using the `store` instruction. In the rest of this work, we use the following notation to represent an instruction performing an operation *op* over operands $o_1, \ldots, o_n$ and creating a variable v:

$$v = op(o_1, \ldots, o_n). \tag{6.1}$$

The CFGs are typed—each variable and each constant has a type. The type system is defined by [75]. For convenience, we introduce the function *typeof* which returns the type of any value. In parts of this work, we deal with user-defined structure types which may be named, hence we introduce the function *typename* which for each named structured type returns its name.

Furthermore, to simplify the following presentation, we introduce the function *op* that assigns an operation to each instruction. The set of all operations consists of the different kinds of instructions in LLVM IR [75].

Each internal instruction of a basic block has exactly one successor: the instruction immediately following it. A block ends by a so-called *branch instruction* that expresses branching in the program or by a *terminator instruction* that terminates the function, possibly returning a value. A branch instruction may have one or two successors, which are always initial instructions of basic blocks. Branch instructions with one and two succeeding instructions are called *unconditional* and *conditional* branches, respectively. For conditional branches, the successor to be followed at runtime is chosen by evaluating a boolean condition that is an operand of the instruction. To simplify the presentation, we introduce three functions related to instruction successors:

- *succ* defines for each non-branching and each unconditional branching instruction its only successor,
- *succT* and *succF* define for each conditional branching instruction the successor instructions which will be followed if the branching condition is evaluated to true and false, respectively. We refer to these successors as to the *true-case* and the *false-case* successors.

6.2 Function Equality

Before describing the main idea of our method for proving semantic equality, we first define what we mean by two functions being semantically equal. The idea is to find so-called *synchronisation points* in both functions and to check that the code between pairs of corresponding synchronisation points is semantically equal. As we shall see, synchronisation points will *typically* but *not always* be at each instruction. Moreover, we will show that multiple syntactical transformations must often be done in order to enable usage of the typical per-instruction synchronisation points.

Intuitively, we consider two pieces of code to be *semantically equal* if they both terminate and their execution produces the same output for the same inputs, or they both do not terminate. Here, by output, we mean values of the output variables and the final state of the memory; and by input, values of the input variables and the initial state of the memory. The memory incorporates both the stack and the heap. The input and output variables are subsets of all variables used in the given function. To reflect possible concurrency, we check that both pieces of code being compared use the same synchronisation means (locks, shared memory, etc.) in the same way. We assume that the used synchronisation assures thread safety, and so we consider sequential executions only. Similarly, we check that both compared pieces of code use the same library and system calls in the same way.

Formally, let us have two functions f_1 and f_2, and, for $i \in \{1, 2\}$, let I_i and V_i denote the sets of instructions and variables used in f_i, respectively. We view the *problem of checking semantic equality* of f_1 and f_2 as the problem of finding two sets of synchronisation points $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$ and two synchronisation functions $smap : S_1 \leftrightarrow S_2$ and $varmap : V_1 \leftrightarrow V_2$ that are bijections and that represent mappings of synchronisation points and variables, respectively, between f_1 and f_2. We consider f_1 and f_2 to be semantically equal iff, for any $s_1, s_2 \in S_1$ and any $s'_1, s'_2 \in S_2$, all of the following hold:

1. For each variable $v \in V_1$ that is used but not defined between s_1 and s_2 (a so-called *input variable* of s_1), the variable $varmap(v)$ is used but not defined between $smap(s_1)$ and $smap(s_2)$, i.e., it is an input variable of $smap(s_1)$. An analogical requirement must hold for s'_1, s'_2 and every $v' \in V_2$, using $smap^{-1}$ and $varmap^{-1}$. This requirement may seem quite strict since the functions may have some input variables that do not influence the output (and that could thus be left out from the requirement), but we will use CFG transformations to eliminate such variables beforehand.

2. For each variable $v \in V_1$ defined between s_1 and s_2 and used after s_2 (a so-called *output variable* of s_2), the variable $varmap(v)$ is defined between $smap(s_1)$ and $smap(s_2)$ and used after $smap(s_2)$, i.e., it is an output variable of $smap(s_2)$. Same for s'_1, s'_2 and each $v' \in V_2$, using $smap^{-1}$ and $varmap^{-1}$.

3. If the value of each input variable v_{in} at s_1 equals that of $varmap(v_{in})$ at $smap(s_1)$ and the state of the memory at s_1 equals that at $smap(s_1)$, then, if the code between s_1 and s_2 is executed and terminates, an execution of the code between $smap(s_1)$ and $smap(s_2)$ terminates, too, and the value of each output variable v_{out} at s_2 equals that of $varmap(v_{out})$ at $smap(s_2)$ and the state of the memory at s_2 equals that at $smap(s_2)$. Likewise for s'_1, s'_2, using $smap^{-1}$ and $varmap^{-1}$.

6.3 Analysis of Function Equality

We now present the top level of our algorithm for checking semantic equality of two functions. Using the notions introduced above, our goal is to find the sets S_1 and S_2 of synchronisation points and the mapping functions *smap* and *varmap* such that blocks of code between corresponding pairs of synchronisation points are semantically equal. Proving such semantic equality is a rather difficult task, especially for large blocks of code. To cope with this problem, as already indicated, we use the following two main ideas:

1. We *transform* the compared functions so that synchronisation points can be defined as often as possible *per instruction*. Individual instructions are then quite simple to compare—intuitively, they should perform the same operations on operands that are the same or can be mapped to each other.

2. In case a per-instruction synchronisation cannot be achieved for a block of code, we check whether the compared blocks match one of the supported *semantics-preserving change patterns (SPCPs)*[1]. If so, we consider the blocks semantically equal too. The check is based on the features of effective SPCPs which are in detail described in Chapter 7 and whose usage is highlighted in the algorithm.

The main workflow of our semantic equivalence checking is shown in Algorithm 1. The algorithm takes two functions f_1 and f_2 as the input. For $i \in \{1, 2\}$, we let P_i denote the list of parameters of f_i, while G_i and C_i denote the sets of all global variables and constants used in f_i, respectively.

First, a number of *code transformations* is applied (Line 1) to the compared functions so that it is easier to define synchronisation points per instruction. These transformations are such that they do not change the semantics of the functions. The most important transformations that we use are constant propagation, redundant instructions elimination, and dead code and dead parameter elimination (since changes in unreachable code do not affect the semantics).

In addition, we run transformations of special calls that occur in LLVM IR, in particular indirect function calls (i.e., calls via function pointers) and calls to assembly code. These calls are replaced by calls to newly generated functions, so-called *abstractions*. *Indirect call abstractions* are function declarations that have the same parameters as the original indirect call and, in addition, a new parameter that represents the called pointer. *Assembly code abstractions* are functions that enclose the called assembly code and promote its parameters into the abstraction function parameters. The purpose of these generated abstractions will be explained later in this section. Finally, some transformations (in particular, function inlining and some related CFG simplifications) are run lazily during SPCP matching (see Section 7.2.2 for more details).

Thanks to the applied transformations, the only left parameters are those whose influence on the output of the functions could not be excluded. Therefore, we consider the functions semantically non-equal if they have a different number of parameters (Lines 2–3).

Afterwards, the algorithm starts building the sets of synchronisation points and the mapping functions. Since one of our main goals is high scalability, these are built lazily. Initially, for each function, the synchronisation set only contains the first instruction of the entry basic block (denoted i^1_{in} and i^2_{in} for f_1 and f_2, respectively), and these two instructions

[1] Note that we also support user-supplied custom change patterns (CCPs) but to simplify the presentation, we will only consider SPCPs in this chapter and extend them to CCPs later in Chapter 8.

Input: Functions f_1, f_2
Result: *true* if f_1 is semantically equal to f_2, *false* otherwise

```
1  run transformations of f1 and f2
2  if |P1| ≠ |P2| then
3      return false
   // Initialisation of synchronisation maps
4  S1 = {i1_in}, S2 = {i2_in}
5  smap(i1_in) = i2_in
6  for 1 ≤ i ≤ |P1| do
7      varmap(p1_i) = p2_i
8  for g1 ∈ G1 do
9      varmap(g1) = g2 where g2 ∈ G2 has the same name as g1
   // Main loop
10 Q = {(i1_in, i2_in)}
11 while Q is not empty do
12     take any (s1, s2) from Q
13     p = detectPattern(s1, s2)
14     for each pair (s1', s2') ∈ succPair_p(s1, s2) do
15         if (s1' ∈ S1 ∨ s2' ∈ S2) then
16             if smap(s1') ≠ s2' then return false
17             else continue
18         if p is none then
19             equal = cmpInst(s1, s2)
20         else
21             equal = compare_p((s1, s1'), (s2, s2'))
22         if ¬equal then return false
           // Update synchronisation sets and maps
23         S1 = S1 ∪ {s1'}, S2 = S2 ∪ {s2'}, smap(s1') = s2'
24         update varmap according to p
25         insert (s1', s2') to Q
26 return true
```

Algorithm 1: Checking semantic equivalence of functions

are synchronised. The variable mapping is created between pairs of parameters (based on their order—Lines 6–7) and pairs of global variables (based on their name—Lines 8–9).

The main loop of the algorithm works with a queue Q of pairs of synchronisation points. In each iteration, a single pair (s_1, s_2) is taken from the queue. The pair is analysed by the function *detectPattern* that checks whether some pattern p out of the supported SPCPs seems applicable. A special value *none* is returned if no pattern is applicable (forcing a per-instruction comparison). The algorithm can be easily generalised to iterate over multiple patterns possibly applicable at the same time—we have not included this possibility for brevity and also because the applicability of our current patterns is exclusive (i.e., no two patterns may be applicable at the same time).

Then, the function $succPair_p(s_1, s_2)$ checks where the next synchronisation points will be placed, i.e., it computes the successor synchronisation pairs of (s_1, s_2). We require each pattern to define the behaviour of $succPair_p$. Due to this, the top-level algorithm does not have to search from where to continue the analysis after a successful detection of an instance of a pattern. In our current implementation of the approach, for a majority of the SPCPs, the successor points will simply be at the instructions immediately following (s_1, s_2). The

```
1 succPair (s1, s2):
2   if op(s1) = op(s2) = cond.branch then
3     return (succT(s1), succT(s2)), (succF(s1), succF(s2))
4   else if op(s1) ≠ cond.branch ∧ op(s2) ≠ cond.branch then
5     return (succ(s1), succ(s2))
6   else yield error
```

Algorithm 2: Computing successor synchronisation points

procedure is, however, prepared to easily incorporate dealing with comparisons of other larger code blocks too—should that be needed.

The default implementation of *succPair* is shown in Algorithm 2. It uses the successor functions from Section 6.1 and returns either one or two pairs of synchronisation points depending on whether a conditional branching follows the current synchronisation points. If some conditional branching appears in one of the functions only, the function ends with an error, causing the comparison to fail as the control flow is different (this case is not included in Algorithm 1 for brevity).

In case the successor synchronisation points are at the instructions immediately following (s_1, s_2), we make use of the successor functions defined in the previous section. Namely, if there is no branching after the current synchronisation point in any of the functions, a single successor pair of synchronisation points is considered: namely, $(succ(s_1), succ(s_2))$. If some branching is encountered in both of the functions, two pairs of successor synchronisation points will be considered: $(succT(s_1), succT(s_2))$ and $(succF(s_1), succF(s_2))$. If some branching appears in one of the functions only, the comparison fails since the control flow is different (this possibility is not included in Algorithm 1 for brevity).

After choosing the successor pair of synchronisation points, we first check whether we have already visited one of the points. If so, we require that the synchronisation points are already mapped to each other, otherwise the synchronisation of the control flow is broken, and the functions are not semantically equal (Lines 15–17).

Subsequently, the blocks of code from the current to the next synchronisation point in each function are checked to be indeed semantically equal. If no pattern is used, each of the blocks contains a single instruction, and these are compared using the *cmpInst* function defined in Algorithm 3. The function checks if the instructions perform the same operation on the same or mapped operands.

Moreover, if the compared instructions use functions as operands, we run Algorithm 1 for the functions unless they were compared before. An exception to this are *indirect function calls* and *calls to assembly code*. As already mentioned, we replace such calls by calls to so-called abstraction functions. For an indirect call, only the arguments of the indirect abstraction function call are compared, leading to a comparison of both the original arguments and the function pointers through which the original call is done. A comparison of the target functions is started when they are assigned to the function pointers (since that is where the functions appear as operands). As for the assembly abstraction functions, the blocks of assembly code inside them are compared for literal equality instead of using Algorithm 1.

When a potentially applicable SPCP was detected, the comparison of blocks is done using the pattern-specific $compare_p$ function. If the blocks are not compared as equal, the algorithm ends, claiming the functions not to be semantically equal.

After the semantic comparison, the synchronisation sets and maps are updated in Lines 23–24. If an SPCP was used, the variable mapping is updated using the way that

```
1  cmpInst (i_1, i_2):
       // Assume (o^1_1, ..., o^1_{n_1}) and (o^2_1, ..., o^2_{n_2}) be the operand lists of i_1 and i_2, respectively
2      if op(i_1) ≠ op(i_2) then // Check equality of opcodes
3          return false
4      for 1 ≤ k ≤ n_1 do
           // Variables must be mapped
5          if o^1_k ∈ V_1 ∧ varmap(o^1_k) ≠ o^2_k then
6              return false
           // Constants must be equal
7          else if o^1_k ∈ C_1 ∧ o^1_k ≠ o^2_k then
8              return false
           // Functions must be recursively compared
9          else // o^1_k is a function
10             if Alg. 1 (o^1_k, o^2_k) = false then
11                 return false
12     return true
```

Algorithm 3: Comparing single instructions

is a part of the SPCP definition. If individual instructions were compared (which is often the case), the update is quite simple as instructions always have at most one output value represented by the fresh local variable created by the instruction. Hence, when two instructions i_1 and i_2 returning a value are compared, the mapping $varmap(v_{i_1}) = v_{i_2}$ is created where v_{i_1} and v_{i_2} are the fresh local variables introduced by the LLVM IR to hold the result of i_1 and i_2, respectively.

Finally, if all reachable synchronisation points were visited and no inequality has been found, the functions are considered semantically equal.

Chapter 7

Built-in Semantics-Preserving Change Patterns

As we have already mentioned in Chapter 6, our main goal is to develop an as-precise-as-possible but still highly-scalable method to automatically compare two versions of a function, typically obtained through refactoring, and determine whether the semantics of the function is preserved. The main complexity here lies in deciding whether a syntactic change in a function causes a change in its semantics. For high scalability, we concentrate on changes that can be handled on the level of particular instructions or that are instances of several generic types of changes, which we denote as *semantics-preserving change patterns (SPCPs)*.

The algorithm for comparing semantics of versions of large-scale software introduced in Chapter 6 supports so-called *effective* SPCPs, each of which must be specified through defining four functions (highlighted in Algorithm 1) that provide:

1. a test indicating potential applicability of the SPCP at a given pair of starting code locations of the compared program versions,
2. a way to compute code locations succeeding the given instance of the SPCP,
3. a condition under which the potential SPCP does indeed preserve semantics, and
4. a way to compute which program variables of the two program versions correspond to each other after the SPCP.

The idea is that the initial test (Point 1) should be done by a quick and efficient analysis of the compared program functions, which may be quite large. On the other hand, the method for determining the actual semantic equality of the code potentially matching the detected pattern (Point 3) may use a more complex algorithm since it is used on a substantially smaller code bounded by the detected potential starting location and the location succeeding the potential pattern instance (determined in Point 2).

In this chapter, we introduce the set of SPCPs that we support in our current implementation of the approach. First, in Section 7.1, we present the way we obtained this set by combining an existing list of refactoring patterns known in the literature with our own experimental study of change patterns commonly appearing in the history of the Linux kernel. After that, in Section 7.2, we present details on how each of the supported patterns is handled within our approach.

7.1 Supported Semantics-Preserving Changes

As we already outlined, the list of semantics-preserving change patterns that we concentrate on is inspired by two sources: (1) the list of refactoring patterns from [44] and (2) our own extensive study of frequent change patterns that we have performed on multiple past versions of the Linux kernel.

Concerning the list of [44], we observe that our proposed approach implicitly allows us to handle a number of patterns. This is mainly caused by three facts:

1. We use a CFG-based representation of programs, in which some constructions of high-level languages that look different in source code are represented the same way. This allows us to handle the *consolidate-conditional* (join adjacent cases in switch), *for-into-while*, *while-into-for*, *add-a-typedef*, and *replace-type* patterns.

2. Our method is insensitive to naming of program entities, which allows it to handle many renaming patterns, in particular, *rename variable/constant/user-defined type/-function*.

3. The CFG transformations we use effectively "neutralize" the effect of some change patterns. This is in particular the case for (i) dead-code elimination that allows us to handle the *remove-unused-variable/parameter/function* patterns, (ii) constant propagation that allows us to handle the *replace-value-with-constant* pattern, and (iii) memory-to-register promotion that allows us to handle the *add-variable* and *replace-expression-with-variable* patterns.

Hence, we observe that our algorithm in its basic form (without any explicit patterns) allows us to handle 15 out of the 29 refactoring patterns mentioned in [44]. From the rest of the patterns, we concentrate on those that occur the most often in real systems code. We support this claim by a study of a number of versions of the Linux kernel that we present below.

7.1.1 Change Patterns in the Linux Kernel

To derive SPCPs common in Linux, we started by analysing all versions of the RHEL 7 kernel from 2014–2018 (RHEL 7 was the major RHEL version until 2019). Newer RHEL versions are then used for an experimental evaluation of our approach in Chapter 10.

In particular, for pairs of succeeding releases, we compared the semantics of functions from the KABI list (cf. Chapter 6 for more information on KABI). We performed the comparison using our proposed algorithm without any custom patterns and looked for functions marked as non-equal. This way, the results contain patterns that our method is not able to handle yet (in other words, our algorithm already filters out changes caused by one of the 15 implicitly supported patterns described above). Note that the encountered changes need not appear directly in the function code—they may be caused, e.g., by a change in a type declaration too.

Subsequently, we manually analysed all the obtained differences and identified the most common remaining kinds of changes not affecting the semantics. This way, we obtained the following list of SPCPs. For each discovered SPCP, we enumerate all patterns from [44] that it covers. In addition, we note that many of the SPCPs identified in this section cover additional refactoring patterns that are not mentioned in [44] (e.g., because they are kernel-specific or because they are rather complex).

Table 7.1: Numbers of SPCPs in KABI functions

RHEL versions	KABI funs	Non-dominated changed functions	Data types	Function splitting	Code loc.	Enum values
7.5/7.6	739	112	10	2	2	0
7.4/7.5	734	218	33	13	1	1
7.3/7.4	678	142	6	3	4	0
7.2/7.3	644	223	9	13	2	0
7.1/7.2	551	111	6	4	3	0
7.0/7.1	395	82	2	5	0	2
	Sum	888	66	40	12	3

Changes in structure data types This pattern covers changes in user-defined structures and unions (typically additions, removals, or renamings of fields), which often result in a situation when, e.g., an access to the same structure field yields a different memory-access offset. The pattern covers two patterns from [44] – namely the *add/rename-structure-field*. We note that this pattern manifests in multiple different ways in the code that works with the updated structure. To this end, we define three variants of the pattern which cover changes in the field alignment (both in non-nested and nested fields) and in the structure size. We discuss these variants in detail in Section 7.2.1.

Splitting code into functions The code is refactored by moving parts of it into functions called from where the original code was. This covers multiple patterns from [44] – namely the *extract/inline-function*, and the *add/reorder-function-parameters* patterns.

Changes in a source code location In the Linux kernel, there are macros and built-in functions that allow one to report the file name and the line number of the current code location. In case such a function/macro is used and the location has changed, the semantics stays the same.

Changes of enumeration values This situation may happen, e.g., when a new value is added into the middle of an enumeration type. In such a case, the rest of the values are shifted and get different numerical values.

Table 7.1 shows numbers of appearances of the mentioned SPCPs in the compared RHEL kernel versions [1].

The first column states the versions of the RHEL kernel being compared. The second column contains the number of functions on the KABI list. The third column contains the number of functions, either from the KABI list or (directly or indirectly) called from them, that contain a difference and that are not called solely by some function already containing a difference (in other words, we do not descend into callees of functions that contain a difference). Here, note that changes in macros or data types show up in the code of functions in LLVM IR too. The remaining columns give numbers of those of the discovered differences that are caused by the SPCPs described earlier.

[1] The RHEL kernel is the same (up to a few minor modifications) as that of the open-source distribution CentOS, which can be retrieved from http://vault.centos.org/.

Changes in about 13 % of the changed functions are fully covered by the above described patterns, and the semantics of these functions did not change. (Usually, the change corresponds to a single pattern, but a few functions were changed via multiple patterns—hence the given percentage cannot be obtained directly from Table 7.1; we computed it separately.) We also analysed the other changes and discovered that 99 % of them affect the semantics. Changes in the remaining 1 % typically represent more complicated refactoring.

In addition to the above, we also inspected individual commits in the Git repository of the Linux kernel upstream (`https://github.com/torvalds/linux`) created between versions 5.10 and 5.17 and concentrated on the commits that are marked as "refactorings" in their commit message (i.e., those that are expected not to change the semantics). We analysed the changes introduced by these commits and identified two additional frequent SPCPs:

Inverse branching conditions A branching condition is replaced by an inverse condition with the branches swapped. This also applies to loop conditions, thus covering the *while-into-do-while* pattern of [44].

Relocated code A piece of code is relocated into a different part of a function (e.g., from the beginning of a loop iteration to before the loop). The relocated code is usually independent from the code skipped by the relocation. From the patterns in [44], it covers the *contract/extend-variable-scope* pattern.

In total, we have thus identified six SPCPs that we consider as important in the given context. These SPCPs cover 9 patterns from [44] but are more general, covering some semantics-preserving changes not covered by [44], yet showing up in the history of the Linux kernel. In addition, we note that we do not cover 5 patterns from [44] as we have never encountered them in our study of Linux. While it should be easy to handle some of them using effective SPCPs (this is applies to 3 patterns related to converting a variable into a pointer and vice-versa), some (in particular the patterns *convert-global-variable-into-parameter* and *group-set-of-variables-into-new-structure*) would require analysing the global state of the compared programs, which our algorithm currently does not do.

In the following chapter, we show that all the above identified six SPCPs can be formulated as effective SPCPs and hence handled by our algorithm.

7.2 Handling the Supported SPCPs

Our method for comparing the semantics of two functions is generic in handling effective SPCPs specified by providing the four functions listed in the beginning of this chapter. We now define these functions for the SPCPs that we identified as frequent in the Linux kernel through our empirical study presented in Section 7.1.1. Some of the patterns use default implementations of some of the functions, which were presented in Section 6.3. In such cases, we do not discuss the functions for the given pattern. We also propose a specific treatment for the code-relocation SPCP that goes beyond our notion of effective SPCPs.

7.2.1 Changes in Structure Data Types

The most common change that we saw in the Linux kernel and that results in different code produced by the compiler while maintaining the semantics is a change of the layout of a user-defined structure type. In C, a *structure type* (i.e., a structure or union) consists

of a list of *fields*, each field having its *name* and *data type*. A programmer can then use the "dot" operator to access individual fields of a variable of a structure data type. When accessing a particular named field f of a variable v, compilers translate the name of f into a numerical *offset*, which is a number that defines the relative offset of the address of f from the starting address at which v lies in the memory. In LLVM IR, this is done by the `getelementptr` (GEP) instruction, which takes a pointer and an index of the field (the first field has index 0, the second field has index 1, etc.) and returns a pointer to the required element. In general, a GEP instruction can take multiple indices as it is able to resolve multiple chained accesses from a single pointer at once.

If the layout of a structure type is changed, usage of the type may be affected in multiple ways: e.g., if a field is added to or removed from the middle of the structure type, the indices of all fields up to the end of the type change. As was outlined earlier, we consider three different variants of this pattern, one for a simple change of a field offset, one for a change of the structure size, and one for a more complicated change involving changes in nested structures but leading to accessing the same memory offsets in the end.

Changed Offset of a Structure Field

When a new field is added into the middle of a structure type, the fields from the point of addition to the end are shifted. Accessing such fields in C (by using the same field name) results in different indices generated by the GEP instruction. Therefore, a special semantic comparison of GEP instructions must be introduced into Algorithm 1 since two GEP instructions with a different constant value of a corresponding parameter may have the same semantics, which would result in a false non-equivalence result. In order to allow for an improved GEP comparison, we exploit LLVM *debugging information* that contains a mapping of field names to field indices. We then specify the pattern as follows (using the implicit versions of computing successor pairs and updating maps of variables).

Detection condition: $op(s_1) = op(s_2) =$ `gep` and both GEP instructions access a structure field.

Definition of $compare_p$: The comparison is done using a slightly modified version of the function *cmpInst* from Algorithm 3. When comparing operands that are GEP indices in Line 7, instead of comparing the numerical offsets o_k^1, o_k^2 for equality, we first retrieve the corresponding field names n_k^1, n_k^2 from debugging information. Then, we distinguish four possible situations:

- $o_k^1 = o_k^2, n_k^1 = n_k^2$—the operands are equal.
- $o_k^1 = o_k^2, n_k^1 \neq n_k^2$—check if n_k^2 occurs in the structure type that contains n_k^1. If it does not, then the operands are equal (the field has very probably been renamed), otherwise we treat the operands as not equal.
- $o_k^1 \neq o_k^2, n_k^1 = n_k^2$—the offset has been shifted, but the programmer still accessed the same name. We check whether there is some pointer arithmetic performed on the pointers computed as the results of the instructions s_1 and s_2. If so, the operands are not equal (as the absolute value of o_k^1 or o_k^2 matters), otherwise they are equal.
- $o_k^1 \neq o_k^2, n_k^1 \neq n_k^2$—the operands are not equal.

Using Absolute Sizes of Structure Types

Some program constructions rely on the absolute size of a type, typically using the `sizeof` operator. In case such a construction is used on a structure type and the layout of the type changes, the resulting size may be different. However, for some program constructions, this difference does not cause a difference in the semantics.

Within our experiments described in Section 7.1.1, we identified a set of such constructions commonly appearing in the Linux kernel and the considered implementation of the standard C library. These are calls to memory manipulating functions such as `malloc`, `memset`, or `memcpy`. We denote the set of these functions as *MemOps*. We are aware that the set needs not be complete, but it was sufficient in our experiments and it can be easily extended.

Detection condition: $op(s_1) = op(s_2) =$ `call` and the same function from *MemOps* is called at both s_1 and s_2.

Definition of $compare_p$: The calls are compared almost as in *cmpInst*, but when comparing operands corresponding to the size of a type where the type is a structure type, names of the structures (retrieved from the debugging information) are compared instead of the absolute sizes. The bodies of the called functions are not compared.

Different Ways to Access the Same Field

There may occur situations when the layout of a structure type is changed in a more complicated way, and the same fields are accessed in a different manner. An example that often happens in the Linux kernel is a replacement of a field f by a field u of a `union` type that contains the original field f and some other field g. When accessing the field f through u, the final generated offset is (usually) exactly the same (since f is stored at the beginning of u), but the access is done using one more field (and one more GEP instruction in LLVM IR).

In this case, the same semantics is achieved by a different number of instructions in each of the compared functions. Thus, this pattern must compare larger blocks of instructions.

Detection condition: $op(s_1) = op(s_2) =$ `gep` and there is a sequence of instructions $i_1, \ldots, i_n$ in the first version of the code where:

- $i_1 = s_1$, i.e., the sequence starts from s_1,
- for all $1 \leq k < n$, $succ(i_k) = i_{k+1} \wedge op(i_k) =$ `gep`,
- the sequence has a single input variable—the source pointer accessed by i_1,
- the sequence has a single output variable—the variable v_{i_n} that is the final pointer computed by the sequence, and
- for all $1 \leq k \leq n$, all index operands of i_k are constant.

Moreover, an analogous sequence (possibly of a different length) of GEP instructions starts from s_2. We denote by s_1'/s_2' the last instructions of the sequences starting from s_1/s_2, respectively. At least one of the sequences has more than one instruction.

Definition of $succPair_p$**:** Returns a pair of synchronisation points $(succ(s'_1), succ(s'_2))$.

Definition of $compare_p$**:** As all indices are constant, we can compute the exact memory offset that each instruction would produce. Thus, $compare_p$ returns true iff (1) *varmap* maps the input variable of s_1 to the input variable of s_2 and (2) the sum of all offsets of all instructions is equal for both sequences.

Method for updating *varmap***:** $varmap(v_{s'_1}) = v_{s'_2}$.

Let us note that, currently, we do not combine this pattern variant with the first variant (*Changed Offset of a Structure Field*) presented earlier in this section. Thus, we allow for changes of field names and layouts only when comparing a pair of single GEPs. When comparing blocks of GEPs, the resulting offset must be exactly the same. This could be improved by comparing the difference between the computed offsets to the difference of offsets of changed fields, but we have never found a use-case for this construction, and so we have not implemented it.

7.2.2 Moving Code into Functions

Another frequent change that preserves semantics is splitting a block of code into pieces and moving (some of them) into some new (or existing) functions. Such functions are then called with appropriate parameters from the location where the original code was. This is a common refactoring process that usually improves readability and simplifies the code.

Handling such a situation in Algorithm 1 would require it to be able to compare a possibly large block of code with the full body of a function. However, as our experiments show, when some code is moved into a function, the function usually executes exactly the same operations as the original code. Thus, to handle this kind of changes, it suffices to find a correct synchronisation between instructions of the original code and those of the called function.

In order to achieve this, we make use of multiple CFG transformations with the most important being *function inlining*. It is a well-known process in which a function call is replaced by the function's body where the values of the call arguments are provided for the values of formal parameters. If the called function performs exactly the same operations as the original code did, this transformation subsequently allows us to perform a per-instruction synchronisation, which enables an efficient comparison of the semantics. Moreover, since function inlining is widely used during compilation, the LLVM infrastructure provides highly optimised inlining routines.

Detection condition: $op(s_1) = \texttt{call} \vee op(s_2) = \texttt{call}$.

Definition of $compare_p$**:** The implementation of $compare_p$ is shown in Algorithm 4. If the compared instructions are not equal and at least one of them is a call, the call is inlined and a new comparison of the current synchronisation pair is scheduled since the call instruction is replaced by new code.

The above approach is, however, not always sufficient. Sometimes, the code is moved into a function containing more behaviour than the original code, but that behaviour is not executed for the particular call (e.g., by setting some parameter to `false`). The semantics

```
1 compare_p (s1, s2):
2   if ¬cmpInst(s1, s2) then
3     if op(s1) = call then  inline s1 and simplify
4     if op(s2) = call then  inline s2 and simplify
5     insert (s1, s2) to Q // Yield a new comparison of s1, s2
6   return true // The comparison will always continue
```

Algorithm 4: Handling function refactoring in Algorithm 1

is preserved, but the code produced by inlining contains more instructions than the original code, and so a per-instruction synchronisation cannot be achieved.

Therefore, we perform additional semantics-preserving CFG transformations after the inlining, namely *constant propagation* and *dead code elimination*. Constant propagation may evaluate some conditions to false, and dead code elimination will then remove unreachable code, leaving only the code that can possibly be executed for the particular function call. If that code performs the same operations as the original code, our method is subsequently able to show this.

7.2.3 Changes in Enumeration Values

In C, the `enum` keyword allows one to create a list of (typically related) named constants. Usually, the numerical values themselves are not important and when they are changed, it is not considered a semantic change. Such changes often occur when a new value is inserted into the middle of an `enum`. All identifiers after the added one then get assigned a different value by the compiler. However, LLVM IR contains the resulting values only.

Detection condition: s_1 and s_2 are instructions containing a constant operand that corresponds to an `enum` identifier. To detect such a situation, the function analyses debugging information and collects possible mappings of values to `enum` identifiers.

Definition of $compare_p$**:** Instructions are compared via *cmpInst*, but if a constant operand corresponding to an *enum* identifier is checked, the identifier string is compared instead of the value[2].

7.2.4 Changes in Source Code Location

This semantics-preserving change is specific for the Linux kernel, in particular kernel warning functions. Calls to these functions contain two kinds of information that can be omitted without changing the semantics. First, from the semantics point of view, the warning message is not important. Second, calls to these functions often contain the line number and absolute path to the C source file where the call occurs. Nonetheless, a change of such information does not affect the semantics of the caller function. We handle this by the following SPCP-specific functions.

Detection condition: $op(s_1) = op(s_2) =$ `call`, calling same kernel warning function.

[2]Note that one can construct artificial programs that the described method would claim semantically equal although they are not. This might happen if the programs compare *enum* identifiers with actual constants that they represent. Such constructs, however, break the purpose of enumeration types, and we have not seen them in any of the real code that we considered in our experiments.

Definition of *compare*$_p$: Compare the calls using *cmpInst* but do not compare operands that represent a warning message, a line number, or a file name (we identified a list of such operands by manually analysing all kernel warning functions).

7.2.5 Inverse Branch Conditions

A common pattern that we identified among kernel refactoring commits covers situations when a branching condition (typically a condition in an `if` statement) is replaced by its inverse condition (i.e., a condition that holds if and only if the original condition does not hold). Such a change is semantics-preserving if the true- and false-case successors of the concerned branching instruction are swapped.

In LLVM IR, this situation may occur in multiple ways: by using comparison instructions with inverse conditions, by using a boolean negation instruction, or by a combination of both. Our detection condition handles all of these cases.

Detection condition: There exists a sequence $i_t, [i_n], i_b$ of instructions in the first version of code such that:

- i_t (a test instruction) returns a boolean value denoted v_t,
- i_n needs not be present but if it is, it returns a boolean negation of v_t, denoted v_n, and
- i_b is a conditional branching instruction where the condition is v_n if i_n is used. Otherwise, the condition is v_t.

An analogous sequence $i'_t, [i'_n], i'_b$ must exist in the second version of the program.

Definition of *compare*$_p$: The comparison starts by comparing i_t and i'_t in a way similar to *cmpInst*. Three cases may occur:

1. i_t and i'_t are semantically equal,
2. i_t and i'_t are comparison instructions with semantically equivalent operands and an inverse condition,
3. i_t and i'_t are not semantically equal.

For (3), we simply return *false* as the test instructions are incomparable. Otherwise, we check if one of the following conditions is true:

- case (1) occurred and exactly one of i_n and i'_n is present or
- case (2) occurred and either both or none of i_n and i'_n is present.

In these cases, the branching instructions have logically inverse conditions, hence we return *true* but swap the order of successors of i_n to compare its original true-case successor with the false-case successor of i'_n and vice-versa.

For the rest of the cases, the branching instructions have equal conditions (although one may have been negated twice), hence we return *true* and proceed with the default comparison algorithm.

Note that it may occur that there is a number of chained negation instructions between i_t and i_b, however, we assume that these are reduced to at most one instruction by our pre-processing transformations.

7.2.6 Code Relocations

The last semantics-preserving change whose support we consider as crucial w.r.t. our study from Section 7.1.1 is *relocation of a piece of code* into a different part of a function. For the concerned functions to be semantically equal, it is necessary that the relocated code is independent of the code that is skipped by the relocation. Currently, we require the relocated code to be sequential (without branching), but it may be relocated into any part of the same function. Based on our experiments with the Linux kernel presented in Section 7.1.1, this is the most common case for code relocation.

Code relocation cannot be covered by our main algorithm via the notion of effective SPCPs. The reason is that to handle it we need multiple interconnected phases as shown below, and, moreover, we want to apply it with the lowest priority (i.e., only when no other SPCP is applicable). The purpose of the latter point is that the detection of this pattern is the most complex one, and hence, for the sake of scalability, we want to rule out other patterns before we try to apply this one.

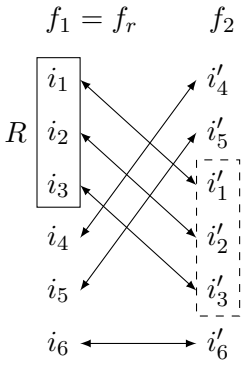


Figure 7.1: An illustration of a relocation detection

Therefore, we handle it as shown in Algorithm 5 that replaces Line 22 in Algorithm 1. The method works in three phases, namely relocation detection, relocation matching, and relocation checking. We introduce these phases in detail in the rest of this section. To make the presentation more understandable, we illustrate the algorithm on the example from Figure 7.1. It shows two functions f_1 and f_2, each containing a single basic block with six instructions ($i_1, \ldots, i_6$ for f_1 and $i'_1, \ldots, i'_6$ for f_2). For $1 \leq k \leq 6$, let i_k be semantically equivalent to i'_k (this is depicted with the arrows in the figure). The difference between the functions is that the instructions i_1, i_2, i_3 (highlighted with the box labeled R) have been relocated to a later part of the function (highlighted with the dashed box).

Relocation Detection

It is run if the current pair of synchronisation points is compared as non-equal and no potentially relocated block R is being processed (in Figure 7.1, this would happen when comparing the first pair of instructions i_1 and i'_4). Then, the function *detectRel*, described in Algorithm 6, is called. It tries to advance in the code (first in one of the functions and then in the other) with an aim of finding a synchronisation point in one of the functions under comparison that would match the current synchronisation point in the other function. If such a synchronisation point is found, the block of instructions that were skipped during the search is marked as a *potentially relocated block* R, and s_1 and s_2 are moved such

```
22 if ¬equal then
23   if R = [] then // Relocation detection
24     if (R = detectRel(s1, s2, f1)) ≠ [] then  fr = f1
25     else if (R = detectRel(s1, s2, f2)) ≠ [] then  fr = f2
26     else  return false
27     continue // Go to Line 11 of Algorithm 1
28   else if s1 ∉ R ∧ s2 ∉ R then // Start relocation matching
29     if fr = f1 then
30       sc = s1,  s1 = R[0]
31     else
32       sc = s2,  s2 = R[0]
33     insert (s1, s2) to Q
34     continue // Go to Line 11 of Algorithm 1
35   else  return false
36 else // Relocation checking
37   if fr = f1 ∧ s1 is the last instruction of R then
38     if (succ(s1), sc) depends on R then
39       return false
40     s'1 = sc, s'2 = succ(s2), R = []
41   else if fr = f2 ∧ s2 is the last instruction of R then
42     if the block of code between succ(s2) and sc depends on R then
43       return false
44     s'1 = succ(s1), s'2 = sc, R = []
   // Go to Line 19 of Algorithm 1
```

Algorithm 5: Handling code relocations in Algorithm 1

that they are synchronised again. The function in which the relocated block was found is remembered in f_r, and the computation proceeds to Line 11 of Algorithm 1. In the example from Figure 7.1, *detectRel* would first traverse f_1 by comparing i'_4 (the currently processed instruction of f_2) with i_1, i_2, i_3, and finally with i_4 where it would find a match. Hence, it would store $f_r = f_1$ and $R = [i_1, i_2, i_3]$. The synchronisation points s_1 and s_2 would be set to i_4 and i'_4, respectively. At this point, Algorithm 1 would normally continue (by successfully comparing i_4 to i'_4 and i_5 to i'_5) up to the point where the next phase begins.

Relocation Matching

The second phase is run if (1) the current pair of synchronisation points is compared as non-equal, (2) a potentially relocated block R has been previously identified, and (3) the comparison has not entered R, yet (i.e., the comparison of the two given functions has arrived to the location where R is assumed to be relocated). In this case, the current synchronisation point of f_r is remembered in s_c and then moved back to the first instruction of R, with the pair (s_1, s_2) re-inserted into Q and the computation proceeding to Line 11 of Algorithm 1. This way, Algorithm 1 will take care of comparing the relocated code for semantic equality. In Figure 7.1, this phase would start when comparing i_6 with i'_1. The synchronisation point $s_1 = i_6$ would be stored in s_c and then moved to i_1, while s_2 would stay at i'_1. The instructions inside R would be then handled by Algorithm 1.

```
1  detectRel (s1, s2, fr):
2      backup s1 and s2,  R = []
3      while op(s1) ≠ branch ∧ op(s2) ≠ branch do
4          if cmpInst(s1, s2) = equal then
5              return R
6          if fr = f1 then
7              append s1 to R,  s1 = succ(s1)
8          else
9              append s2 to R,  s2 = succ(s2)
10     restore s1 and s2
11     return []
```

Algorithm 6: Detection of code relocations

Relocation Checking

Finally, if the equality of the entire block is confirmed, this phase checks if the code that has been skipped by the relocation (which is the code between the instruction following the last instruction of R and the instruction s_c from which the comparison will continue) is not data-dependent on the relocated block. Two blocks of code are data dependent if one of the blocks reads a value that is written to by the other block. If the blocks are independent, the relocation is semantically equal, and the comparison continues normally from the remembered synchronisation point s_c (by setting s'_1 and s'_2 to the appropriate values and continuing with them on Line 23 of the original Algorithm 1). In Figure 7.1, this phase would run once the entire relocated block has been successfully matched, i.e., right after i_3 and i'_3 have been compared as equal. Then, we would check if the relocated code ($[i_1, i_2, i_3]$) depends on the skipped code ($[i_4, i_5]$). If it does not, then the relocation is confirmed as semantics-preserving, and the comparison would continue by setting $s'_1 = s_c = i_6$ and $s'_2 = succ(i'_3) = i'_6$ (which would be then compared as equal by Algorithm 1, effectively showing semantic equality of f_1 and f_2).

We note here that the proposed method is the first step towards a more robust handling of code relocations. It suffers from two issues: (1) it cannot handle relocations of blocks containing branching, and (2) it is limited to detecting a single relocated block at a time. Extending the method to blocks containing branching seems to be quite straightforward as long as the relocated block has a single entry and a single exit point. Extending it to simultaneously handling multiple relocated blocks is more difficult since the complexity would increase with each detected block (for each non-equal synchronisation pair, one would have to try to match all potentially relocated blocks), and the synchronisation sets and maps would have to be reset if a block is not matched (to remove a potentially incorrect synchronisation coming from an unsuccessful match). So far, we have not found enough use cases to support such non-trivial extensions, but they might be a part of our future work if a real need to support them appears in practice.

Chapter 8

Custom Change Patterns

Our method for checking semantic equivalence of versions of large-scale software that we introduced in Chapter 6 has originally been developed to handle generic code change patterns, with focus on semantics-preserving change patterns. In this chapter, we extend the set of supported patterns beyond SPCPs to user-supplied *custom change patterns (CCPs)*.

To achieve that, we first introduce a novel representation of CCPs suitable for our method in Section 8.1. After that, we use this representation as a building block for the algorithm for matching CCPs in the compared programs that we propose in Section 8.2.

8.1 Representation of Custom Change Patterns

In this section, we introduce a formal definition of CCPs. We require CCPs to be able to describe arbitrary changes that may occur in programs supported by our method (i.e., any programs compilable to LLVM IR). To this end, our pattern definition is inspired by and based on the program representation that we use.

With respect to the way patterns are currently handled in Algorithm 1, we require our CCP representation to allow for the following:

- describe a code change between two compared versions of a program,
- parametrise the pattern so that it can be matched to a larger set of changes, and
- express which memory locations should be synchronised after the pattern is successfully matched to an observed change.

8.1.1 Formal Definition of Custom Change Patterns

We represent our patterns with the help of *parametrised control-flow graphs*. A parametrised CFG c is a triple:

$$c = (in, cfg, out). \tag{8.1}$$

Here, *cfg* is a control-flow graph which can be parametrised using undefined local variables and undefined structure types, representing the input values and types of the CFG, respectively. The component in of c is the set of all input variables and types used in *cfg*. Last, out denotes the set of "outputs" of the parametrised CFG, i.e., the set of local variables which may be used outside of c.

With respect to this, we define a code change pattern as a tuple

$$p = (c_o, c_n, imap, omap) \tag{8.2}$$

where c_o and c_n are parametrised CFGs corresponding to the old and the new version of the code change that is represented by p, respectively. Let

$$c_o = (in_o, cfg_o, out_o) \tag{8.3}$$

$$c_n = (in_n, cfg_n, out_n). \tag{8.4}$$

Then, $imap : in_o \leftrightarrow in_n$ is a mapping between the inputs of the parametrised CFGs c_o, c_n expressing which values and types in the compared programs must have the same semantics in order to successfully match the pattern. Analogically, $omap : out_o \leftrightarrow out_n$ is a mapping between output variables of the parametrised CFGs expressing which variables of the compared programs will be mapped (i.e., will have the same semantics) after the pattern is successfully matched.

The above definition of CCPs allows us to seamlessly incorporate them into the current comparison algorithm. We do this by defining generic implementations of the four pattern-specific operations that were described in the beginning of Chapter 7.

8.1.2 Encoding Change Patterns with LLVM IR

To be able to use CCPs in practice, we need them to be encoded in a form that DiffKemp is able to use. As patterns are represented by parametrised CFGs, LLVM IR is the natural choice.

In particular, we encode each pattern using two LLVM IR functions, one for each parametrised CFG of the pattern (c_o and c_n). The sets of input values from in_o and in_n are encoded using LLVM function parameters and their mapping (for *imap*) is determined based on the parameters' order. For type parameters, the patterns contain a custom type prefixed with `diffkemp.type`. This custom type is then used in both c_o and c_n, hence no explicit encoding of the mapping is necessary.

The sets of output variables (out_o and out_n) and their mapping *omap* are represented by introducing a special function `diffkemp.mapping` which is called in each pattern function just before its exit. The call contains a list of variables representing out_o and out_n and the mapping is determined automatically based on their order.

The pattern matching algorithm that we introduce in the following section requires our patterns to encode some additional information. These are typically encoded in LLVM IR using LLVM metadata. More details on LLVM metadata can be found in [75].

8.2 Custom Change Pattern Matching

We now propose a method to detect occurrences of custom change patterns in the compared programs. Since our goal is to utilise this method in the comparison described in Section 6.3, we use the same approach as is already used for the semantics-preserving change patterns. In particular, we provide definitions for pattern-specific operations required by Algorithm 1. These definitions are generic, which means that they can be used for any custom change pattern having the form defined in the previous section.

To simplify the presentation in the rest of this section, we assume the following situation:

- two versions of a function f, denoted f_o and f_n (the old and the new version, respectively), are being compared using Algorithm 1,
- f_o and f_n are represented using CFGs as described in Section 6.1, and we refer to them as to *compared-function CFGs*,

- Algorithm 1 is at the point of processing a pair of synchronisation points s_o, s_n, and
- the goal is to check if a custom change pattern $p = (c_o, c_n, imap, omap)$ is applicable and, if so, to apply it. We also let $c_o = (in_o, cfg_o, out_o)$ and $c_n = (in_n, cfg_n, out_n)$ and we refer to cfg_o and cfg_n as to *pattern CFGs*.

In the following subsections, we propose definitions of functions required by the individual steps of Algorithm 1.

8.2.1 Pattern Detection

The purpose of this step is to check if a pattern can be applied from the current pair of synchronisation points. As the check may be executed for each pattern at each synchronisation points pair, it is necessary that it is done in a very quick and efficient way. On the other hand, custom change patterns may describe arbitrary changes, hence the largest part of the matching must be done in this step.

For a custom change pattern p, we need to check that cfg_o is a subgraph of f_o and cfg_n is a subgraph of f_n. Checking of subgraph isomorphism is generally expensive, however, we are dealing with CFGs in a specific situation which allows us to use a rather efficient approach. In particular, we build on assumptions that (1) each CFG has a single entry point and (2) we only need to match the pattern CFG starting from the current synchronisation point in the compared-function CFG. Hence, there is a unique point where the matching must start, and we can use a straightforward control-flow traversal to check whether all instructions of the pattern CFG match instructions in the compared-function CFG.

Even though the problem is now much reduced, two major issues remain:

1. We still need to perform the full CFG comparison from each pair of synchronisation points for each pattern. Even though the matching algorithm is efficient, running it so many times may cause problems with scalability, which is the main concern of our approach.

2. The occurrence of a pattern CFG in the corresponding compared function CFG may be interleaved with non-related instructions. This is a common situation as the LLVM compiler often reorders non-conflicting instructions.

To address the first issue, we always start the CFG matching from the first pair of instructions which differ between the pattern CFGs cfg_o and cfg_n. These are required to be marked explicitly in the pattern (we use LLVM metadata to do that) and we denote them as *the first differing instruction pair*. Thanks to that, the pattern can only be matched if f_o and f_n contain a synchronised pair of differing instructions and, in addition, that pair matches the first differing pair of the pattern. In practice, this heuristic quickly eliminates most of the non-matching pattern candidates for most of the synchronisation points.

Note that the pattern CFGs may start with sequences of instructions which are the same for both cfg_o and cfg_n and which will be not be initially compared. Such instructions denote a *context* in which the pattern must be applied—they define how some of the variables used inside the pattern must be created. We denote the sets of such variables ctx_o and ctx_n for the old and the new version of the pattern CFG, respectively. Naturally, it is necessary to check that the same context appears in the compared functions, i.e., that the variables of the compared functions matched with the context variables were created using equivalent instructions. This is done as the last step of our matching method.

Input: $c_x = (in_x, cfg_x, out_x)$: pattern CFG
f_x: compared function CFG
s_x: current synchronisation point in f_x
Result: $match_x$: mapping between values and types of cfg_x and f_x

```
1  e_p = first differing instruction of cfg_x
2  e_f = instruction immediately following s_x in f_x
3  Q = {(e_p, e_f)}
4  match_x = {}
5  while Q is not empty do
6      take any (i_p, i_f) from Q
7      if ¬matchInst(i_p, i_f, in_x) then // updates match_x
8          if succ(i_f) is defined then
9              add (i_p, succ(i_f)) to Q // instruction skipping
10             continue
11         else return ∅
12     if i_p is conditional branch then
13         add (succT(i_p), succT(i_f)) to Q
14         add (succF(i_p), succF(i_f)) to Q
15     else add (succ(i_p), succ(i_f)) to Q
16 if ¬checkContext(match_x, ctx_x, in_x) then
17     return ∅
18 return match_x
```

Algorithm 7: Matching pattern CFG to one of the compared functions

To address the second issue, we allow our matching algorithm to "skip" instructions on the side of the compared-function CFGs. These instructions will need to be compared using the default comparison, which is a problem that we address later in this section.

With respect to all the described mechanisms, our algorithm for detecting an occurrence of a pattern is shown in Algorithm 7. This algorithm must be run separately for both versions of the compared program and their corresponding pattern CFGs. Hence, we use the subscript $x \in \{o, n\}$ as a placeholder for either the old or the new version. If both comparisons succeed, the pattern is considered as applicable.

The algorithm simply traverses the control-flow of cfg_x and f_x, starting from the first differing instruction in cfg_x and from the instruction immediately following the current synchronisation point s_x in f_x. For each instruction pair, it uses the *matchInst* function to check that the instructions match. If they do not, the algorithm allows to skip instructions in f_x if they have a single successor which may be followed. We do not allow to skip conditional branching instructions. If the algorithm succeeds, it performs the context validation step and returns $match_x$, which is a map (a set of pairs) of semantically equivalent values and types between cfg_x and f_x.

The $match_x$ map is created by the *matchInst* function, whose definition is shown in Algorithm 8. At its entry, the function takes two instructions (using the notation from Eq. (6.1)) and the set of the corresponding pattern inputs. It checks whether the instructions perform the same operation over semantically equivalent operands. Operands are considered semantically equivalent if both their types and values match. This matching is checked in multiple steps:

1. If the type of the pattern instruction operand is a part of the pattern input, we create a new matching with the type of the corresponding operand from the compared function (Lines 3–4). In other words, this step marks which types of the compared funtion

Input: $i_p : v_p = op_p(o_p^1, \ldots, o_p^m)$ (pattern instruction),
$i_f : v_f = op_f(o_f^1, \ldots, o_f^n)$ (compared-function instruction),
in_x (pattern inputs)
ctx_x (pattern context)
Output: *true* if i_p matches i_f, *false* otherwise

```
1  if op_p ≠ op_f then return false // ensures m = n
2  for 1 ≤ i ≤ n do
3      if typeof(o_p^i) ∈ in_x then
4          add (typeof(o_p^i), typeof(o_f^i)) to match_x
5      else if typeof(o_p^i) ≠ typeof(o_f^i) then
6          return false
7      if o_p^i ∈ in_x ∨ o_p^i ∈ ctx_x then
8          add (o_p^i, o_f^i) to match_x
9      else if ¬(o_p^i = o_f^i ∨ name(o_p^i) ≈ name(o_f^i) ∨ match_x(o_p^i) = o_f^i) then
10         return false
11 match_x(v_p) = v_f
12 return true
```

Algorithm 8: Definition of the *matchInst* function

are mapped to which input types of the pattern CFG. Later, during the *semantic equality detection* step (Section 8.2.3), we check that the types of the old and the new compared functions which were mapped to semantically equivalent pattern inputs are also semantically equivalent.

2. If the types do not match (and one of them is not an input), operands are considered as semantically different (Lines 5–6).

3. A check similar to point 1 is done for the operand values, except that they may also be parts of the pattern context (Lines 7–8).

4. Last, a check similar to point 2 is done for the operand values (Lines 9–10). For values, the equality check is more complex than for types and it depends on the operand kind. For constants, we check for direct equality. For functions and global variables, we check for name match. Besides pure name equality, we allow patterns to specify *renaming rules* which describe how called function names may differ between the versions (this is expressed by $\approx$ in the algorithm). Last, for local variables, we check if the values have already been mapped via $match_x$. These checks are handled by the individual disjuncts at line 9 of the algorithm.

If the comparison succeeds, *matchInst* accordingly updates the mapping $match_x$. The mappings created by comparing the two given program versions against the different sides of a change pattern (i.e., $match_o$ and $match_n$) will be used later during the *semantic equality detection* step.

The last step of Algorithm 7 is *context validation*. As mentioned before, so-called pattern context instructions are initially not matched for optimisation purposes. In this step, we check that the values created by such instructions (the set of these values is denoted ctx_x) are created in the same way in the pattern and in the compared function. This check is performed by Algorithm 9.

The algorithm checks that variables created by context instructions which should have the same semantics (as they are in $match_x$) are created by semantically equivalent instruc-

```
Input: match_x: matching of values between cfg_x and f_x
       ctx_x: the set of pattern context variables
       in_x (pattern inputs)
1 do
2     for (v_p, v_f) ∈ match_x do
3         if v_p ∈ ctx_x then
              // Let i_p and i_f be instructions creating v_p and v_f, resp.
4             if ¬matchInst(i_p, i_f, in_x) then // may update match_x
5                 return false
6 while match_x is updated
7 return true
```

Algorithm 9: *checkContext*: matching context instructions between the pattern and the compared function CFGs

tions. Running *matchInst* may again update $match_x$, hence the check must be run while $match_x$ changes.

8.2.2 Determining Successor Synchronisation Points

The purpose of this step is to determine where the analysis continues from, after the comparison of the current code chunks succeeds. For custom change patterns, we continue from (i.e., place a successor synchronisation point to) each instruction i of the compared function which has not been matched to the pattern CFG but which is immediately following some instruction matched to the pattern CFG. However, there may be several such instructions, due to two reasons:

- the pattern CFG is not required to have a single exit point, hence matching may end in multiple basic blocks and
- as explained in the previous section, we allow to skip instructions in the compared-function CFG, and these must be compared after the pattern is successfully matched.

Due to these, there may be a large number of instructions to continue from, hence we introduce an additional limitation. We only place a synchronisation point at each instruction i if there is no other synchronisation point already placed at an instruction i' such that i is reachable from i'. This is safe to do as if there is such an instruction i', i will be eventually analysed using the default comparison method which follows the control flow.

During the subsequent comparison, we make the main algorithm ignore instructions that were already matched by the pattern.

8.2.3 Semantic Equality Detection

Once the pattern is determined as applicable (i.e., a matching sub-CFG can be found in the corresponding function), it is still necessary to check that it is applied on semantically equivalent values and types of the compared-function CFGs. In other words, we need to check that the values and the types that were matched to the inputs of the pattern CFGs have the same semantics in both compared function versions.

To do this, we make use of several mapping functions, in particular *imap* which maps semantically equivalent inputs of the pattern CFGs and $match_o$ and $match_n$ which map variables and types of pattern CFGs to variables and types of the compared-function CFGs

as determined during pattern detection. Using these mappings, we check if the following holds:

$$
\begin{aligned}
&\forall (i_o, i_n) \in imap : \\
&\quad match_o(i_o) = match_n(i_n) \vee && \text{(covers constants)} && (8.5)\\
&\quad varmap(match_o(i_o)) = match_n(i_n) \vee && \text{(covers variables)} && (8.6)\\
&\quad typename(match_o(i_o)) = typename(match_n(i_n)) && \text{(covers struct types)} && (8.7)
\end{aligned}
$$

That is, we check that pattern inputs which should be semantically equivalent (via *imap*) are matched to values and types in the compared functions which are also semantically equivalent. Semantic equivalence in the compared functions is done based on the kind of the compared value: constants are compared by value, variables are compared using *varmap*, and structure types are compared by name.

8.2.4 Updating the Variable Mapping

The last step of handling change patterns in Algorithm 1 is to determine which variables created by the pattern have the same semantics for the following comparison. This is done by updating the *varmap* function. For our custom change patterns, we again use the maps $match_o$ and $match_n$ created during the pattern detection step along with the pattern outputs mapping *omap*. In particular, we update *varmap* so that:

$$\forall (o_o, o_n) \in omap : varmap(match_o(o_o)) = match_n(o_n) \tag{8.8}$$

That is, for each pair of semantically equivalent pattern outputs (determined via *omap*), we let the values matched in the compared functions to be also semantically equivalent (via *varmap*).

Chapter 9

Comparing the Use of Global Variables

Until now, we always assumed that our method is used to compare two versions of a function. However, it may also be useful for analysing differences in the semantics of *global variables*. As we mentioned in the introduction of Chapter 6, global variables may represent, e.g., runtime or module parameters in the Linux kernel.

We view the semantics of a global variable as defined by the semantics of all the code that is in any way influenced by it. Therefore, to compare the semantics of a global variable, we compare the semantics of all functions using the variable with their counterparts in the other version of the project (the functions are matched according to their names). If the given variable is used in more functions in one version than in the other, we view its semantics as not equal.

Although comparing entire functions using a given global variable is sufficient to show the semantic equality of global variables, it may lead to some false non-equivalence verdicts. The reason is that not necessarily all parts of a function are influenced by the given variable, and if a change occurs in such an irrelevant part, it has no influence on the way the compared variable influences the code.

```
1  extern int var;
2  void f(){
3    A1;
4    if (var) { B; }
5    C1;
6  }
```

```
1  extern int var;
2  void f(){
3    A2;
4    if (var) { B; }
5    C2;
6  }
```

Figure 9.1: Functions semantically equivalent with respect to the value of `var`

The above situation is illustrated in the two pieces of code shown in Figure 9.1. Here, `A1`, `A2`, `B`, `C1`, `C2` specify arbitrary blocks of code that do not use `var`. Also, we assume that the code in `A1`, `A2`, `C1`, and `C2` has no dependence on the code in `B`. Under these assumptions, in both versions of the function, the only block of code affected by `var` is the block `B`. Therefore, we may say that the functions are semantically equal *with respect to the global variable* `var` because the code that the variable can influence has not changed. Below,

we use the notion of function equality w.r.t. a global variable to compare the semantics of global variables in different project versions.

9.1 Comparing Functions w.r.t. a Variable

We now propose an extension of the algorithm presented in Chapter 6 that allows one to compare functions w.r.t. a variable v. The extension is based on removing those parts of the code that are not dependent on the value of v. By dependence, we mean both data dependence (where the value of some expression depends on the value of v) and control dependence (where the fact whether some code is executed depends on the value of v).

For removing the irrelevant parts of the code, we use the approach of *program slicing* [119]. It is a well-known technique of reducing a program to the minimal form that satisfies some given criterion. We, in particular, propose a specialised combination of *forward* and *backward* slicing where the criterion is preservation of all statements influenced by the given variable v. The algorithm first proceeds forward through the code and preserves only those instructions that are (transitively) control- or data-dependent on the value of v. Afterwards, the algorithm takes all such identified instructions and proceeds backwards to preserve also parts of the program that may (in a control or data way) influence any of the preserved instructions. We describe the method we use in detail in Section 9.2.

We then extend Algorithm 1 to read the global variables whose semantics is to be compared as a part of its input and to slice both of the functions to be compared at the beginning of the procedure.

9.2 Slicing Algorithm

We now propose a slicing algorithm suitable for our needs. Although there is a number of existing slicing algorithms [67, 26], they are mostly aimed at high precision at the cost of speed. Typically, these algorithms are based on using a program dependence graph [40, 56], which can be quite expensive to compute. Also, many of the slicing algorithms are quite general, supporting various slicing criteria, backwards and forward slicing, inter-procedural slicing, etc. On the other hand, our method only needs to perform a specific intra-procedural slicing w.r.t. the value of a global variable and our top priority is very high scalability. Therefore, we propose a simple specialised slicing method that computes dependence relations on the fly and is able to slice a function w.r.t. the value of a global variable in a very efficient way. Compared to existing works, ours is the most similar to [3], but the algorithm of [3] computes a slice for all variables of the program simultaneously, which is inefficient for our case. Also, the work [3] does not feature the reconstruction of the control flow, contained in our algorithm and explained later on, which is crucial for a subsequent comparison of semantic equality.

Our slicing algorithm takes a function f and a global variable g as inputs. The output is the function f' that is the minimal slice of f containing all instructions that are dependent on the value of g in some way. The produced slice must be a valid CFG so that it can be compared to another CFG for semantic equality using the method proposed in the previous sections.

The method works in three main phases: (1) computing control- and data-dependent instructions that are to be preserved, (2) restoring the data flow among the dependent

instructions, and (3) restoring the control flow among the dependent instructions and those needed to preserve the data flow so that the produced CFG is valid.

Phase 1: computing dependent instructions In the first phase, the set of dependent instructions is computed in a rather standard way. We define an instruction j to be *data-dependent* on an instruction i if (1) it uses the local variable v_i created by i as an operand or if (2) it reads from a pointer p (or from any pointer aliased with it) that i writes into. For computing aliases, we use the LLVM's *Basic Alias Analysis* [1], which is a cheap and simple alias analysis. Optionally, a more precise alias analysis can be used.

Similarly, we define an instruction j to be *control-dependent* on an instruction i if the fact whether j is executed depends on the execution of i. In our LLVM-based program representation, such a dependence may occur only if i is a branching instruction. In such a case, we define the set of all instructions control-dependent on i as the set of instructions that are reachable from i via a single branch only (i.e., their execution is dependent on the value of the branching condition).

With the relations for data- and control-dependence defined as above, we compute the set of all dependent instructions as the transitive closure over the union of these relations, starting from the set of all instructions that read the value or the address of the global variable g. The computed instructions are to be preserved, the other instructions are candidates for removal (but they may be preserved too due to the further phases).

The further phases of our slicing restore the data and control flow among the instructions to be preserved, possibly adding further instructions, in order to produce a valid CFG that can be compared with other CFGs. In a valid CFG, all used local variables must be defined within the CFG. Also, the original control flow must be preserved in the sense that instructions included in the slice must be reachable for the same values of input parameters and the same state of the memory as they were in the original function. This is the main difference of our algorithm from other slicing algorithms (e.g. [3]) that do not necessarily produce a valid CFG.

Phase 2: restoring the data flow We extend the slice by all instructions that write into some variable used by some instruction in the slice. In particular, we repeat the following steps until a fixpoint is found:

- If an instruction in the slice uses a local variable v_j, the instruction j is included in the slice.
- If an instruction in the slice reads data from a pointer p, all instructions writing into p or into a pointer aliased with p are included in the slice. This step can be optimised by considering the last writing instruction(s) that reach the reading instruction only.

Phase 3: restoring the control flow Phase 3 defines, for each instruction i of the slice, its successor(s) within the slice. In other words, if the original successor of i is not included in the slice, a different successor must be chosen while preserving the control flow. This is simple if all instructions not present in the slice have a single successor. Then, the new successor of i can be found by following the single successor of each non-included instruction until an instruction of the slice is found. A problem arises with conditional branching instructions that are not included in the slice. For each such instruction, we first

[1] https://llvm.org/docs/AliasAnalysis.html

compute the sets RT and RF denoting the set of instructions of the slice reachable via the true-case and the false-case successor, respectively. Based on these sets, there are multiple scenarios that may occur:

- $RT \subset RF \vee RF \subset RT$: the sets are not equal and all instructions reachable by one of the successors are reachable by the other successor. Then, only the successor that reaches more instructions is followed.
- $RT \neq RF \wedge RT \not\subseteq RF \wedge RF \not\subseteq RT$: there are instructions in the slice reachable exclusively by each of the successors. In such a case, both successors must be preserved, i.e., the corresponding branching instruction must be included in the slice.
- $RT = RF$: there are no instructions reachable exclusively by either of the successors. In such a case, any successor could be followed; however, if one of them goes into a loop that returns back to the current branching instruction, we need to follow the other successor in order to reach the end of the function. Otherwise, any successor can be chosen.

This method ensures that each non-included branching instruction has a single followed successor, and therefore it is simple to restore the control-flow as explained above. Note, however, that Phase 3 may add new instructions to the slice, which happens if a conditional branch cannot be omitted from the slice. In that case, it is necessary to re-run Phases 2 and 3.

Chapter 10

Implementation and Experiments

We have implemented all of the proposed methods in a tool called DiffKemp. The tool is able to automatically compile the Linux kernel as well as any other `Makefile`-based project into LLVM IR and to compare the semantics of different versions of functions from the project. In addition, when analysing the Linux kernel, DiffKemp can compare runtime parameters (configurable via the `sysctl` command), too. Detailed usage information can be found in README (see the GitHub repository stated below).

Moreover, for projects written in C (such as the Linux kernel), which are the primary target of DiffKemp, it is able to precisely locate the C source symbol (a function, a macro, or a type) that causes the detected semantic difference—provided that such a difference is discovered. Currently, DiffKemp supports all versions of Clang/LLVM from 9 to 15. It is distributed as source code[1], an RPM package for Fedora[2], or a container image[3].

We performed several experiments with DiffKemp in order to demonstrate capabilities of the proposed methods. All experiments were run on an 8 core, 3 GHz Intel Core i7-1185G7 machine with 32 GB of RAM.

The experiments are presented in the following sections. In total, we performed 7 experiments related to various parts of our work. The first four experiments (Sections 10.1–10.4) evaluate the generic comparison method (Chapter 6) with built-in semantics-preserving change patterns (Chapter 7). The following experiment (Section 10.5) evaluates our slicing algorithm introduced in Chapter 9. Finally, the last two experiments (Sections 10.6 and 10.7) demonstrate usefulness of custom change patterns described in Chapter 8.

10.1 KABI Functions

As mentioned earlier, the ability of our tool to detect undesirable changes in a software project may be particularly useful for developers of those Linux distributions that aim at stability and compatibility. One of such distributions is the *Red Hat Enterprise Linux (RHEL)* whose KABI functions, cf. Chapter 6, should be stable across minor RHEL releases. A change of the semantics of a KABI function may lead into a compatibility breakage. To show that DiffKemp can indeed be helpful to ensure that this does not happen, we used it to check preservation of the semantics of KABI functions on versions of RHEL 8 up to the version 8.5. Table 10.1 shows the obtained results.

[1] Available from https://github.com/viktormalik/diffkemp under the Apache 2.0 license.
[2] https://copr.fedorainfracloud.org/coprs/viktormalik/diffkemp/
[3] https://hub.docker.com/r/viktormalik/diffkemp

Table 10.1: Checking semantic equivalence of KABI functions

RHEL versions	KABI functions	DiffKemp verdict: equal/not equal/ unknown	Total functions compared	Total LOC compared	Runtime (mm:ss)
8.0/8.1	471	362/84/25	3,460	36,066	05:58
8.1/8.2	521	334/161/26	3,601	36,474	05:06
8.2/8.3	628	420/180/28	4,103	44,017	12:15
8.3/8.4	631	447/156/28	3,625	38,130	11:02
8.4/8.5	640	443/169/28	4,025	43,136	10:45

For each pair of the considered successive kernel versions, Column 2 shows the number of KABI symbols that were compared. Column 3 displays the number of functions that are claimed by DiffKemp to be semantically equal, non-equal, as well as those whose equality could not be determined. The last kind of result occurred typically in cases in which DiffKemp was not able to find the function's definition in the kernel source (some functions are, e.g., generated during kernel compilation from macros). In all experiments, not a single function comparison ended with a tool crash nor the tool timed out (with a 30 s timeout).

Furthermore, we manually inspected the functions claimed by DiffKemp to be non-equal, and we have found at most units of (for some versions even zero) functions whose semantics seems unchanged. As far as we can say, the changes were mostly security fixes or bug fixes; they should not break anything in applications relying on KABI, but they indeed change the semantics to some degree as correctly announced by DiffKemp. Checking the potential impact of such semantic changes is left for the developers.

However, the results clearly demonstrate that DiffKemp heavily reduces the amount of the human effort needed—while we were able to check the changed functions in a reasonable amount of time (each pair of versions took a few hours, relying on the changes highlighted by DiffKemp), this would be impossible if one had to inspect all functions potentially reachable from some KABI symbol—their exact numbers are given in Column 4.

To further manifest the scope of the task performed in this experiment, Column 5 gives the total number of compared lines of C code. The last column gives the execution time (obtained as an average wall time of 5 runs) that DiffKemp spent on comparing the given pair of versions compiled to LLVM IR (the compilation time is not included). It shows that DiffKemp is able to check semantic equality on a large code base in the order of minutes, which is sufficient, e.g., to integrate it into the continuous integration process.

10.2 Refactoring Commits in the Linux Kernel

In our second experiment, we apply DiffKemp on various refactorings of the Linux kernel that produced code that is syntactically different but should have the same semantics. In particular, we took all the commits between versions 5.10 and 5.17 of the upstream kernel (https://github.com/torvalds/linux) containing the word "factor" or "refactor" in the commit message. In total, we discovered 45 such commits and, for each of them, we compared the semantics of the functions changed by the commit.

Prior to running the experiment, we manually investigated the commits and discovered that, despite the commits are marked as refactorings, many of them actually contain a

semantic difference. These are caused, e.g., by added assertions, safety checks (such as a check that a pointer is not `NULL` before dereferencing it), mutex locking, or by the fact that the new version of the function covers more behaviour than the original version did. Out of 24 such commits DiffKemp correctly identified all to be semantically not equal. The remaining 21 changes do indeed preserve semantics. From them, DiffKemp was able to confirm the semantic equality in 43 % of the cases. The rest of the commits contain more complicated refactorings that are beyond the capabilities of the light-weight approach of DiffKemp. Such changes seem to require a heavier-weight approach, perhaps relying on more complex formal methods, where, however, the scalability is a problem (as indicated also by an experiment presented in Section 10.4).

In addition, an interesting observation is that out of the 45 analysed commits, 3 commits were "fixup" commits which fixed a bug introduced by some previous commit marked as refactoring. We used DiffKemp on the bug-introducing commits, and it was capable to identify a semantic difference in all 3 cases.

All in all, this experiment demonstrates another valuable use case of DiffKemp—it can be used as a pre-processing tool when reviewing whether a change is truly semantics-preserving. In such an application, DiffKemp is able to handle a significant number of changes (in our case, 43 %) that do not need to be reviewed anymore (either manually or by other, more costly, approaches). Moreover, if the commit is not semantics-preserving (while it should be), DiffKemp is able to discover it, which may prevent introduction of hidden bugs into the codebase.

10.3 The *musl* Standard C Library

Even though the main desired application of DiffKemp is on the Linux kernel, the methods it implements are generic and applicable on any project compiled into LLVM IR. We demonstrate this in our third experiment where we check preservation of the semantics of library functions from the C standard library. For simplicity, we chose the *musl libc* implementation since there exists a project that allows this library to be compiled into LLVM IR (`https://github.com/SRI-CSL/musllvm`). We took the last 9 versions and compared the semantics of all exported functions. Then, we compared the differences identified by DiffKemp with the set of all syntactic differences obtained from the versioning system by using the `diff` command (since we built the project for x86, we excluded the differences for non-x86 architectures). Table 10.2 shows the results.

For each pair of versions, the table shows the total number of `diff` chunks obtained from the versioning system as described above (Column 2). We use chunks since they give a sufficient granularity (often, a single chunk represents a single change) while their number is quite simple to obtain. The following two columns show the numbers of chunks that do not change the semantics (as determined by a manual analysis) and that were marked by DiffKemp as equal (Column 3) or as not equal (Column 4). The results in Column 4 represent false positives (FP) where a chunk was determined to be semantically different although it is not. The number of such results is quite low for each pair of versions[4].

The last two columns of Table 10.2 show numbers of semantically different chunks (again, determined manually) that were marked by DiffKemp as equal (Column 5) and

[4]We admit that the last column of the table may include some (perhaps units of) complicated refactorings for which, during our manual analysis, we could have failed to see that they preserve the semantics and we classified them into non-equal results.

Table 10.2: Analysis of the *musl* C standard library functions

musl libc versions	`diff` chunks	Semantically equal DiffKemp equal	Semantically equal DiffKemp not equal (FP)	Semantically not equal DiffKemp equal (FN)	Semantically not equal DiffKemp not equal
1.1.19/.20	199	77	6	0	116
1.1.20/.21	598	470	12	0	116
1.1.21/.22	118	33	1	0	84
1.1.22/.23	126	93	3	0	30
1.1.23/.24	93	42	0	0	51
1.2.0/.1	77	20	1	0	56
1.2.1/.2	165	56	0	0	113

not equal (Column 6). Column 5 represents false negative (FN) results, where a differing chunk is not identified. We may observe that DiffKemp produced no such result.

Overall, this experiment shows that DiffKemp can successfully identify a large number of syntactic differences to be semantics-preserving (for some versions, there is more than 80 % of such differences) in a large real-world project while providing a relatively small number of false results.

10.4 A Comparison with Other Tools

To compare DiffKemp with some other existing tool for checking semantic equivalence, we considered the tools mentioned in Chapter 11. As our first choice to compare with, we took the LLReve tool [66] as it is rather recent and open-source. Moreover, it runs on Linux and uses LLVM IR, which allowed us to use our tooling for compiling the Linux kernel into LLVM IR. We chose 30 functions from our previous experiments, including both functions where DiffKemp succeeds as well as fails to provide a correct result, and compared their semantics using LLReve. Unfortunately, both the Linux kernel and the *musl* C library use program constructions that LLReve does not support (such as calls via function pointers, inline assembly code, or floating-point data types). Due to this, LLReve crashed for a large number of programs. Using various tweaks, we were able to run it on several examples; however, out of these, only a single comparison succeeded (here, LLReve confirmed the result of DiffKemp)—the rest timed out (on a 30-seconds time-out). This demonstrates that the complexity and the size of the comparison is too large for a tool based on heavy-weight formal methods.

As the second tool, we chose SymDiff [71] as it is also quite recent and open-source. However, SymDiff uses the so-called *Boogie Verification Language* as its internal representation and the available compilers are not able to compile the Linux kernel. We tried to use an LLVM-to-Boogie translation provided by the SMACK formal verification tool [108], but, despite all our efforts, we were unable to get the considered programs into a form that SymDiff could handle.

Table 10.3: Determining benefits of using slicing

RHEL versions	Compared functions	Differences without slicing	Differences with slicing	Eliminated differences	Runtime without slicing	Runtime with slicing
8.0/8.1	275	56	39	17 (30 %)	11:07	05:09
8.1/8.2	274	90	55	35 (39 %)	06:33	04:05
8.2/8.3	271	84	52	32 (38 %)	04:25	03:09
8.3/8.4	273	84	60	24 (29 %)	04:06	03:06
8.4/8.5	287	86	50	36 (42 %)	05:05	03:09

10.5 Effectiveness of Program Slicing

One of the important parts of the proposed approach is the program slicing algorithm introduced in Chapter 9. In particular, it is crucial for analysis of semantic differences related to usage of global variables (often representing system parameters). Slicing of functions using the global variables ensures that differences occurring in code not affected by the compared variable are not taken into consideration.

In this experiment, we assess the effectiveness of this approach by determining how many of such differences (which may be considered false positives) are removed by an application of our slicing algorithm. We do this by comparing the number of differences reported by DiffKemp with and without performing the slicing for a list of Linux kernel runtime parameters (also known as *sysctl* parameters). In particular, we chose all parameters from the `kernel` and the `vm` categories, making it 136–142 parameters in total, depending on the kernel version. For each parameter, we identified the global variable controlled by the parameter and compared the semantics of all functions that the variable is used in. We did this for each pair of successive versions of RHEL 8.

The obtained results are shown in Table 10.3. We observe that the application of our slicing algorithm has eliminated a rather large number of false positives, which form approximately 36 % of all the differences. This clearly shows that slicing is a very important component when analysing the semantics of global variables.

In addition, we may observe that the use of slicing reduced the overall runtime of DiffKemp (taken as an average wall time of 5 runs), presumably due to the fact that less code is compared. Slicing itself has very low overhead, taking less than 1 second in total for each pair of versions.

10.6 Application of Custom Change Patterns

In our first experiment related to custom change patterns (CCPs), we demonstrate their practical usability by applying several CCPs on the Linux kernel. We investigated changes done between pairs of the recent releases of RHEL and we identified five patterns of changes which repeat often across versions and although they alter the semantics, the changes are safe to be done. These include, e.g., modifications of the compiler behaviour concerning ordering or speculative execution of commands. A complete description of the used patterns can be found at the end of this section. Then, for each pair of succeeding versions of RHEL 8 (and the last pair of versions of RHEL 7), we performed a semantic comparison of all functions from KABI. We performed each comparison twice—once without the proposed patterns and once with them. A comparison of the obtained results is shown in Table 10.4.

Table 10.4: Results of KABI analysis with and without custom patterns

RHEL versions	Differing functions		Pattern occurrences				
	w/o patterns	with patterns	P1	P2	P3	P4	P5
7.8/7.9	20	18		✓			✓
8.1/8.2	137	132	✓	✓	✓		✓
8.2/8.3	150	145	✓	✓	✓		
8.3/8.4	173	172				✓	
8.4/8.5	150	144		✓		✓	

The table shows numbers of functions identified as semantically differing with and without usage of patterns. Note that these are not necessarily KABI functions but may be called by one of them. We may observe that using 5 patterns removed 19 detected differences, i.e., each pattern was able to eliminate almost 4 differences on average. While this may not seem a lot, note that every difference should be reviewed manually. Hence, removing even a small number of differences may substantially reduce the amount of human work needed. In this case, the removed functions were often called from multiple KABI symbols and the overall output was shortened by 40 diff chunks comprised of 816 lines.

Another important part of this experiment is in the second part of the table, which shows that each pattern was successfully applied in at least two different pairs of versions (some were applied even across major releases). This demonstrates that the patterns are generic enough to be defined just once and then reused within a project for its lifetime.

10.6.1 The List of Used Patterns

We now give the list of patterns used in the experiment. For each pattern, we give an example of a real usage of the pattern within the RHEL kernel. Even though our patterns are defined in LLVM IR, we give examples in C as it is much more readable. The LLVM IR representations of the patterns can be found in the DiffKemp repository[5]. In our experiment, we defined 5 patterns:

P1: Use `READ_ONCE` for a memory read

Usage of the `READ_ONCE` macro prevents the compiler from merging or refetching memory reads. This pattern describes a situation when a simple memory read is replaced by a memory read through the macro. For example:

$$\texttt{p}\rightarrow\texttt{cpu} \quad \rightarrow \quad \texttt{READ\_ONCE}(\texttt{p}\rightarrow\texttt{cpu})$$

The pattern is parametrised by 3 inputs: (1) the pointer to read from, (2) the field to read, and (3) the type of the pointer.

P2: Use `WRITE_ONCE` for a memory write

The `WRITE_ONCE` macro is analogical to `READ_ONCE`, except that it is suited for memory writes. This pattern describes a situation when a simple memory write is replaced by a write through the macro. For example:

$$\texttt{p}\rightarrow\texttt{cpu} = \texttt{cpu} \quad \rightarrow \quad \texttt{WRITE\_ONCE}(\texttt{p}\rightarrow\texttt{cpu}, \texttt{cpu})$$

[5] https://github.com/viktormalik/diffkemp/tree/master/tests/regression/patterns

Table 10.5: Comparison of runtime with and without pattern matching

RHEL versions	Run time		KABI functions	Compared functions
	w/o patterns	with 24 patterns		
8.0/8.1	2m 43s	2m 41s	471	3446
8.1/8.2	3m 30s	3m 29s	521	3643
8.2/8.3	4m 36s	4m 24s	628	3978
8.3/8.4	5m 36s	5m 35s	631	3607
8.4/8.5	5m 24s	5m 24s	640	4002

This pattern is parametrised by 4 inputs: (1) the pointer and (2) the field to write to, (3) the type of the pointer, and (4) the value to write.

P3: Use `unlikely` for a condition
Usage of the `unlikely` macro tells the compiler that certain condition will evaluate to true only in a very small number of cases. The compiler can use this information to, e.g., perform a more efficient ordering of instructions. This pattern reflects a situation when the `unlikely` macro is added to a condition. For example:

```
if(sched_info_on())   →   if(unlikely(sched_info_on()))
```

The boolean condition is the single input of the pattern.

P4: Replace `spin_(un)lock` by `raw_spin_(un)lock`
The Linux kernel provides multiple functions for locking. This pattern describes a situation when usage of `spin_lock` is replaced by `raw_spin_lock`. For example:

```
spin_lock(&last_pool->lock) → raw_spin_lock(&last_pool->lock)
```

The same situation may happen with unlocking, hence we would normally need 2 patterns. Thanks to the possibility to specify renaming rules (see Section 8.2.1), our approach allows to handle both locking and unlocking using a single pattern.

P5: Replace `RECLAIM_DISTANCE` by `node_reclaim_distance`
The `RECLAIM_DISTANCE` macro and the `node_reclaim_distance` global variable are two ways of setting the maximum distance between CPU nodes used for load balancing. This pattern describes a situation when the usage of the macro is replaced by the usage of the global variable. Since this is just a simple replacement of one identifier by another, we leave this pattern without an example.

10.7 Efficiency of CCP Matching

In our second experiment related to CCPs, we demonstrate high scalability of our pattern matching algorithm. Scalability is one of the main properties of DiffKemp, allowing it to be applied in practice. In this experiment, we again perform an analysis of KABI functions of the recent RHEL versions, however, this time, we apply as many as 24 patterns (taken from the first experiment and from our regression tests). Naturally, most of these will never be matched, however, the algorithm must try to match every pattern for every difference

found. Hence, this experiment shows that even this larger number of patterns does not affect scalability of the overall analysis. The results are displayed in Table 10.5.

We may observe that for all versions, the run time of the analysis is equal or even slightly shorter when using patterns. This shows that our matching algorithm is truly efficient. Each run time was obtained as an average wall time of 5 runs. The reason why times are shorter with patterns is that patterns cause more functions to be compared as equal, eliminating the necessity for further computations, such as precise difference localisation, which are a part of DiffKemp. To highlight the performance of the overall comparison, we give the number of KABI symbols and total unique functions compared for each pair of versions.

Chapter 11

Related Work

There is a number of existing works on static analysis of semantic equivalence—for an overview, see, e.g., [74]. Some of the approaches were implemented in tools applied to real-life code, such as RVT [45], SymDiff [71, 65], DiSE [103, 5], LLReve [66], or UC-KLEE [109]. Many of these tools use a similar general approach—namely, equivalence of functions under comparison is encoded using formulae and/or special program constructions, and a suitable decision procedure or program verifier is used to prove equality. In particular, SymDiff uses Z3 [35], RVT uses CBMC [30], UC-KLEE uses KLEE [21]. LLReve represents function equality by a set of Horn clauses and uses a Horn solver, such as Z3, to solve them. DiSE is slightly different in that it employs KLEE to generate function summaries and then compares the summaries. These approaches build on heavy-weight formal methods that, despite a lot of advances, do still not scale enough to allow equivalence checking on large code. This applies even to one of the latest works in the area [29], based on semantics-driven alignment of program traces, which scales on benchmarks from vectorizing compilers up to tens of lines.

On the other hand, there exist light-weight and extremly fast tools based on text similarity (such as the Unix `diff` tool) or on simple abstract-syntax-tree matching [99] that are able to compare huge code bases in the order of seconds. These are, typically, able to handle only the simplest semantics-preserving changes (such as, e.g., variable renaming).

Compared with the mentioned works, our approach lies in between the two areas. While our method is not as fast as the simple approaches, it can show equality of much more complicated refactorings in the order of minutes for a similarly large code. Also, since we build on light-weight back-end approaches, our method scales far better than those using formal methods, at the possible expense of not being able to show equality of some heavily refactored functions.

There still exist other tools—based, e.g., on comparing dependence relations [61], abstract semantic trees [107], or (similarly to our work) control-flow graphs [4]—that also aim at practical usability on large projects. However, these tools primarily aim at finding differences between programs and at describing the differences in the best way possible, typically not being able to ignore semantics-preserving changes.

Another field of works in this area aims at identification of refactorings in software: cf., e.g., [105]. These works often introduce or make use of a pre-defined list of refactoring patterns. The best known list is probably the Fowler's catalogue [42], which describes mostly structural refactorings occurring in object-oriented languages. Refactoring lists for low-level procedural languages (such as C) are less common—the most exhaustive that we are aware of is [44]. In our work, we concentrate on supporting a number of patterns

from [44], extended by several additional types of semantics-preserving changes occurring in the Linux kernel that are discovered by our own in-depth study of a number of Linux versions.

Principles of semantics-preserving code transformations and variable mapping were successfully used in other works too, e.g., in [36] for so-called on-stack replacement. We, however, combine these basic ideas with multiple further techniques (e.g., advanced pattern matching), allowing us to show equivalence of real-life programs with more complex refactorings.

Apart from comparing the semantics of programs, equivalence checking was successfully applied in hardware (see [58] for an overview), and there exist several industry-level works on translation validation of compilers [100, 118, 52] too. These are, however, far from the problems considered here.

Another important part of our work is the method for matching custom change patterns introduced in Chapter 8. The idea of using code pattern matching in static analysis is not new and there exists a handful of successful applications. One of the areas is bug finding represented by tools such as FindBugs and its successor SpotBugs [57] for Java or Cppcheck [94] for C/C++. These tools analyse programs on the level of abstract syntax trees or bytecode and try to identify pre-defined patterns which typically lead to incorrect or malicious behaviour. Our patterns differ from these in a way that they always work with a pair of programs (instead of a single one), since they have to describe a code change.

Pairs of program versions are also compared by works aimed at automatic extraction of bugfix patterns [81, 83]. These works extract patterns of changes from software patches using convolutional neural networks [81] or a patch generalization algorithm [83]. Especially the latter work uses patterns similar to ours—while the authors propose to use parametrized ASTs, we rely on parametrized control-flow graphs which better suit our use-case of checking functional equivalence. The main difference is that our goal is to take an existing pattern and detect its instance in a pair of versions, whereas these works take an existing set of patches and infer a set of patterns. The inferred patterns are then used to introduce new bug classes for FindBugs or for automatic program repair [82].

Similar to these are works aimed at summarizing differences between programs [4, 61, 107] which produce a description of differences done between two versions of a software. The produced output in some way corresponds to our patterns, however, such descriptions are typically not suited for further processing. This is also the case for a group of works aimed at identification of structural and API refactorings [37, 105, 120]. These try to analyse programs and find occurrences of refactorings from some pre-defined list, typically the Fowler's catalogue [42].

Perhaps the closest work to ours is the Coccinelle tool [102] whose original purpose is to detect and apply collateral evolutions in Linux drivers. The tool uses a custom specific language SmPL [101] to represent patterns of code changes. SmPL is based on semantic patches which are translated to the CTL logic. These are then matched against a CFG of a program (using model checking) to obtain a version of the code after the patch is applied. Coccinelle deals with a slightly different problem than we do—it applies a single patch to a single program version to obtain a new version, while we deal with two existing program versions and have to match the observed changes with one of the existing patterns. In addition, Coccinelle uses a more heavy-weight approach compared to our light-weight and fast graph matching algorithm capable of matching a large number of patterns on huge code-bases in the matter of minutes.

To the best of our knowledge, the work presented in this paper is the first one which attempts to deal with arbitrary code change patterns in the context of semantic equality analysis. Other tools for static analysis of semantic equivalence presented earlier in this section are able to deal with arbitrary code changes, however, the changes must be semantics-preserving (which is not a requirement for our patterns).

Chapter 12

Conclusion

In this thesis, we contribute to the area of static analysis of software with several new techniques targeting low-level system code written in C. The thesis is split into two parts, each dealing with a different area of static analysis. Our contributions strive to address typical problems of static analysers, especially their limited applicability to complex real-world programs.

In the first part, we introduce new methods for formal verification of programs manipulating dynamically-allocated data structures and arrays. In particular, we introduce two new abstract template domains capable of reasoning about the shape of linked structures on the heap and about the contents of arrays. Our domains are specialized for a template-based invariant synthesis, and we integrated them into the 2LS framework which implements this approach. The new domains are designed in a way that allows their seamless combination with other abstract domains present in 2LS (e.g., the template polyhedra domain), which enables verification of programs using various complex data structures such as unrolled linked lists. As a part of our efforts, we also developed a new memory model allowing to encode memory-manipulating operations using fixed first-order formulae.

2LS regularly competes in the International Competition on Software Verification (SV-COMP) and the results from the recent years show that our contribution heavily improved verification capabilities of 2LS. In addition, thanks to the abstract domain combinations, 2LS is able to reason about contents of unbounded dynamic data structures, which allows it to soundly verify programs that only one other verifier from the competition handles. Last but not least, we observe that our new domains preserve the great speed of verification, which is one of the assets of 2LS.

On the other hand, there still remains a lot of open challenges. Our heap shape domain is not able to track ordering of the linked nodes which prevents it, e.g., to verify absence of memory leaks in many cases. In addition, we do not properly support pointer arithmetic and more complex data structures such as trees or skip-lists. Our array domain is still rather simple and will require more sophisticated combination with other abstract domains (e.g., the *zones* domain) to allow it handle more complex programs.

In the second part of the thesis, we have proposed a light-weight and highly scalable approach to automatically checking semantic equivalence of functions and global variables in large-scale industrial programs. Our method is aimed at showing equality of programs containing semantics-preserving changes typically resulting from refactoring.

The proposed algorithm is based on a combination of light-weight techniques, namely instruction-by-instruction comparison on the level of LLVM IR, various program transfor-

mations, pattern matching, and a specific slicing algorithm that we propose. This combination allows to discover differences between two versions of a software while ignoring semantics-preserving changes which correspond to one of the built-in or user-defined change patterns. We also showed that our method handles 24 out of the 29 patterns from the list of common C refactoring patterns introduced in [44] and, in addition, several other semantics-preserving change patterns that the list does not state and that are common in the Linux kernel. Additionally, thanks to our support of user-defined patterns, our method is able to handle not only semantics-preserving changes, but also changes which break the semantics but are known to be safe (and therefore can be omitted from the results of the semantic comparison).

We have implemented the proposed approached in a tool called DiffKemp. Our experiments with DiffKemp show that it is able to successfully analyse large projects, such as the Linux kernel, reasonably fast and with not many false results. This is, to the best of our knowledge, beyond capabilities of any other of the existing tools in the area.

Approaches based on heavier-weight formal roots can, in theory, show equality of more functions, but their scalability is limited (as also our experiments showed). However, an interesting direction of future work is to use results from DiffKemp to simplify the programs under comparison to small fragments of code considered non-equal by DiffKemp and then compare these fragments by some heavier-weight approach.

Another area for future research is related to custom change patterns which at the moment must be specified manually. One of the possible future improvements could be automatic inference of patterns which, when combined with our solution, could substantially improve the area of scalable analysis of semantic equivalence of software.

Bibliography

[1] Abdulla, P. A., Holík, L., Jonsson, B., Lengál, O., Trinh, C. Q. et al. Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata. In: *Proceedings of the 11th Automated Technology for Verification and Analysis*. Springer, 2013, vol. 8172, p. 224–239. Lecture Notes in Computer Science.

[2] Afzal, M., Asia, A., Chauhan, A., Chimdyalwar, B., Darke, P. et al. VeriAbs: Verification by Abstraction and Test Generation. In: *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, p. 1138–1141. DOI: 10.1109/ASE.2019.00121.

[3] Alomari, H. W., Collard, M. L. and Maletic, J. I. A Very Efficient and Scalable Forward Static Slicing Approach. In: *Proc. of the 19th Working Conference on Reverse Engineering*. IEEE Computer Society, 2012, p. 425–434.

[4] Apiwattanapong, T., Orso, A. and Harrold, M. J. A Differencing Algorithm for Object-Oriented Programs. In: IEEE. *Proc. of the 19th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2004, p. 2–13.

[5] Backes, J., Person, S., Rungta, N. and Tkachuk, O. Regression Verification Using Impact Summaries. In: *International SPIN workshop on Model Checking Software*. Springer, 2013, vol. 7976, p. 99–116. LNCS.

[6] Barnett, M., Leino, K. R. M. and Schulte, W. The Spec# Programming System: An Overview. In: *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Berlin, Heidelberg: Springer-Verlag, 2004, p. 49–69. CASSIS'04. DOI: 10.1007/978-3-540-30569-9_3. ISBN 3540242872.

[7] Berdine, J., Cook, B. and Ishtiaq, S. SLAyer: Memory Safety for Systems-Level Code. In: *Proceedings of the 23rd International Conference on Computer-Aided Verification*. Springer, 2011, vol. 6806, p. 178–183. Lecture Notes in Computer Science.

[8] Beyer, D. Second competition on software verification. In: *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2013, p. 594–609.

[9] Beyer, D. Advances in Automatic Software Verification: SV-COMP 2020. In: Biere, A. and Parker, D., ed. *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, p. 347–367.

[10] Beyer, D. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In: *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, p. 401–422.

[11] Beyer, D. Progress on Software Verification: SV-COMP 2022. In: *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, p. 375–402.

[12] Beyer, D., Dangl, M., Dietsch, D. and Heizmann, M. Correctness Witnesses: Exchanging Verification Results between Verifiers. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: ACM, 2016, p. 326–337. FSE 2016. DOI: 10.1145/2950290.2950351.

[13] Beyer, D., Dangl, M., Dietsch, D., Heizmann, M. and Stahlbauer, A. Witness Validation and Stepwise Testification across Software Verifiers. In: *Proceedings of the 2015 10th Meeting on Foundations of Software Engineering*. New York, NY, USA: ACM, 2015, p. 721–733. ESEC/FSE 2015. DOI: 10.1145/2786805.2786867.

[14] Beyer, D., Henzinger, T. A., Majumdar, R. and Rybalchenko, A. Path Invariants. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: Association for Computing Machinery, 2007, p. 300–309. PLDI '07. DOI: 10.1145/1250734.1250769. ISBN 9781595936332.

[15] Biere, A., Cimatti, A., Clarke, E. M. and Zhu, Y. Symbolic Model Checking without BDDs. In: *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 1999, vol. 1579, p. 193–207. Lecture Notes in Computer Science.

[16] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L. et al. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In: Mogensen, T. Æ., Schmidt, D. A. and Sudborough, I. H., ed. *The Essence of Computation: Complexity, Analysis, Transformation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, p. 85–108. DOI: 10.1007/3-540-36377-7_5. ISBN 978-3-540-36377-4.

[17] Bouajjani, A., Habermehl, P., Rogalewicz, A. and Vojnar, T. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: *Proceedings of the 13th Static Analysis Symposium*. Springer, 2006, vol. 4134, p. 52–70. Lecture Notes in Computer Science.

[18] Brain, M., David, C., Kroening, D. and Schrammel, P. Model and Proof Generation for Heap-Manipulating Programs. In: *Proceedings of the 23rd European Symposium on Programming*. Springer, 2014, p. 432–452.

[19] Brain, M., Joshi, S., Kroening, D. and Schrammel, P. Safety Verification and Refutation by k-Invariants and k-Induction. In: *Proceedings of the 22nd Static Analysis Symposium*. Springer, 2015, vol. 9291, p. 145–161. Lecture Notes in Computer Science.

[20] Brauer, J., King, A. and Kriener, J. Existential Quantification as Incremental SAT. In: *Proceedings of the 23rd International Conference on Computer-Aided Verification*. Springer, 2011, vol. 6806, p. 191–207. Lecture Notes in Computer Science.

[21] Cadar, C., Dunbar, D. and Engler, D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 2008, p. 209–224.

[22] Calcagno, C., Distefano, D., O'Hearn, P. W. and Yang, H. Compositional shape analysis by means of bi-abduction. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2009, p. 289–300.

[23] Chakraborty, S., Gupta, A. and Unadkat, D. Verifying Array Manipulating Programs by Tiling. In: *Proceedings of the 24th Static Analysis Symposium*. August 2017, p. 428–449. DOI: 10.1007/978-3-319-66706-5_21. ISBN 978-3-319-66705-8.

[24] Chakraborty, S., Gupta, A. and Unadkat, D. Verifying array manipulating programs with full-program induction. In: Springer. *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2020, p. 22–39.

[25] Chalin, P., Kiniry, J. R., Leavens, G. T. and Poll, E. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. Berlin, Heidelberg: Springer-Verlag, 2005, p. 342–363. FMCO'05. DOI: 10.1007/11804192_16. ISBN 3540367497.

[26] Chalupa, M., Jonáš, M., Slaby, J., Strejček, J. and Vitovská, M. Symbiotic 3: New Slicer and Error-witness Generation. In: *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, vol. 9636. LNCS.

[27] Chen, H., David, C., Kroening, D., Schrammel, P. and Wachter, B. Synthesising Interprocedural Bit-Precise Termination Proofs. In: *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2015, p. 53–64.

[28] Chong, S., and Rugina, R. Static Analysis of Accessed Regions in Recursive Data Structures. In: *Proceedings of the 10th Static Analysis Symposium*. Springer, 2003, p. 463–482.

[29] Churchill, B., Padon, O., Sharma, R. and Aiken, A. Semantic Program Alignment for Equivalence Checking. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, p. 1027–1040.

[30] Clarke, E., Kroening, D. and Lerda, F. A Tool for Checking ANSI-C Programs. In: *Proceedings of the 10th International Conference on Tools and Algorithms for*

the Construction and Analysis of Systems. Springer, 2004, vol. 2988, p. 168–176. Lecture Notes in Computer Science.

[31] Cousot, P. and Cousot, R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1977, p. 238–252.

[32] Cousot, P., Cousot, R. and Logozzo, F. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2011, p. 105–118. POPL '11. DOI: 10.1145/1926385.1926399. ISBN 9781450304900.

[33] Curry, C., Le, Q. L. and Qin, S. Bi-abductive inference for shape and ordering properties. In: IEEE. *Proceedings of the 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2019, p. 220–225.

[34] Darke, P., Agrawal, S. and Venkatesh, R. VeriAbs: A tool for scalable verification by abstraction (competition contribution). In: *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, p. 458–462.

[35] De Moura, L. and Bjørner, N. Z3: An Efficient SMT Solver. In: *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, vol. 4963, p. 337–340. LNCS.

[36] D'Elia, D. C. and Demetrescu, C. On-stack Replacement, Distilled. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, p. 166–180.

[37] Dig, D., Comertoglu, C., Marinov, D. and Johnson, R. Automated detection of refactorings in evolving components. In: Springer. *Proceedings of the 20th European Conference on Object-Oriented Programming*. 2006, p. 404–428.

[38] Dudka, K., Peringer, P. and Vojnar, T. Byte-Precise Verification of Low-Level List Manipulation. In: *Proceedings of the 20th Static Analysis Symposium*. Springer, 2013, p. 215–237.

[39] Eén, N. and Sörensson, N. Temporal induction by incremental SAT solving. *Electronical Notes in Theoretical Computer Science*. 2003, 89:4, p. 543–560.

[40] Ferrante, J., Ottenstein, K. J. and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*. ACM. 1987, vol. 9, no. 3.

[41] Flanagan, C. and Qadeer, S. Predicate Abstraction for Software Verification. In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 2002, p. 191–202. POPL '02. DOI: 10.1145/503272.503291. ISBN 1581134509.

[42] Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, 2018.

[43] Fragoso Santos, J., Maksimović, P., Ayoun, S.-E. and Gardner, P. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2020, p. 927–942.

[44] Garrido, A. *Software Refactoring Applied to C Programming Language.* Urbana-Champaign, 2000. Dissertation. University of Illinois.

[45] Godlin, B. and Strichman, O. Regression Verification. In: *Proceedings of the 46th Design Automation Conference.* ACM, 2009, p. 466–471.

[46] Gopan, D., DiMaio, F., Dor, N., Reps, T. and Sagiv, M. Numeric Domains with Summarized Dimensions. In: Jensen, K. and Podelski, A., ed. *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 512–529. ISBN 978-3-540-24730-2.

[47] Gopan, D., Reps, T. and Sagiv, M. A Framework for Numeric Analysis of Array Operations. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: ACM, 2005, p. 338–350.

[48] Gulwani, S., McCloskey, B. and Tiwari, A. Lifting Abstract Interpreters to Quantified Logical Domains. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* New York, NY, USA: Association for Computing Machinery, 2008, p. 235–246. POPL '08. DOI: 10.1145/1328438.1328468. ISBN 9781595936899.

[49] Gulwani, S., Srivastava, S. and Venkatesan, R. Program analysis as constraint solving. In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 2008, p. 281–292.

[50] Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J. and Vojnar, T. Forest Automata for Verification of Heap Manipulation. In: *Proceedings of the 23rd International Conference on Computer-Aided Verification.* Springer, 2011, p. 424–440.

[51] Halbwachs, N. and Péron, M. Discovering Properties about Arrays in Simple Programs. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* New York, NY, USA: Association for Computing Machinery, 2008, p. 339–348. PLDI '08. DOI: 10.1145/1375581.1375623. ISBN 9781595938602.

[52] Hawblitzel, C., Lahiri, S. K., Pawar, K., Hashmi, H., Gokbulut, S. et al. Will You Still Compile Me Tomorrow? Static Cross-Version Compiler Validation. In: *Proceedings of the 9th.* ACM, 2013, p. 191–201.

[53] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Communications of the ACM.* New York, NY, USA: ACM. oct 1969, vol. 12, no. 10, p. 576–580. DOI: 10.1145/363235.363259. ISSN 0001-0782.

[54] Holík, L., Hruška, M., Lengál, O., Rogalewicz, A. and Vojnar, T. Counterexample Validation and Interpolation-Based Refinement for Forest Automata. In: *Proceedings of the 18th International Conference on Verification, Model Checking, and Abstract Interpretation.* Springer, 2017, vol. 10145, p. 288–309. Lecture Notes in Computer Science.

[55] Holík, L., Peringer, P., Rogalewicz, A., Šoková, V., Vojnar, T. et al. Low-Level Bi-Abduction. In: Springer. *Proceedings of the 36th European Conference on Object-Oriented Programming.* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, p. 19:1–19:30.

[56] Horwitz, S., Reps, T. and Binkley, D. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems.* ACM. 1990, vol. 12, no. 1.

[57] Hovemeyer, D. and Pugh, W. Finding bugs is easy. *ACM SIGPLAN Notices.* ACM New York, NY, USA. 2004, vol. 39, no. 12, p. 92–106.

[58] Huang, S.-Y. and Cheng, K.-T. *Formal Equivalence Checking and Design DeBugging.* Kluwer Academic Publishers, 1998.

[59] Itzhaky, S., Banerjee, A., Immerman, N., Lahav, O., Nanevski, A. et al. Modular reasoning about heap paths via effectively propositional formulas. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* ACM, 2014, p. 385–396.

[60] Itzhaky, S., Bjørner, N., Reps, T. W., Sagiv, M. and Thakur, A. V. Property-Directed Shape Analysis. In: *Proceedings of the 26th International Conference on Computer-Aided Verification.* Springer, 2014, vol. 8559, p. 35–51. Lecture Notes in Computer Science.

[61] Jackson, D. and Ladd, D. A. Semantic Diff: A Tool for Summarizing the Effects of Modifications. In: *Proceedings of the 10th IEEE International Conference on Software Maintenance.* IEEE Computer Society, 1994, p. 234–252.

[62] Jhala, R. and McMillan, K. L. Array Abstractions from Proofs. In: *Proceedings of the 19th International Conference on Computer-Aided Verification.* Berlin, Heidelberg: Springer-Verlag, 2007, p. 193–206. CAV'07. ISBN 9783540733676.

[63] Jonkers, H. B. M. Abstract storage structures. In: *Algorithmic Languages.* IFIP, 1981, p. 321–343.

[64] Kanvar, V. and Khedker, U. P. Heap Abstractions for Static Analysis. *ACM Computing Surveys.* 2016, vol. 49, no. 2, p. 29:1–29:47.

[65] Kawaguchi, M., Lahiri, S. and Rebelo, H. *Conditional Equivalence.* MSR-TR-2010-119. 2010.

[66] Kiefer, M., Klebanov, V. and Ulbrich, M. Relational Program Reasoning Using Compiler IR. In: *Proceedings of the 8th Working Conference on Verified Software: Theories, Tools, and Experiments.* Springer, 2016, vol. 9971, p. 149–165. LNCS.

[67] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J. and Yakobowski, B. Frama-C: A software analysis perspective. *Formal Aspects of Computing*. Springer. 2015, vol. 27, no. 3, p. 573–609.

[68] Korakitis, K., Dodd, L., Muir, R., Solodkov, N. and Coates, S. *State of the Developer Nation, 23rd Edition*. London, United Kingdom: SlashData, October 2022.

[69] Kroening, D., Malík, V., Schrammel, P. and Vojnar, T. *2LS for Program Analysis*. 2023. Available at: https://arxiv.org/abs/2302.02380.

[70] Kumar, S., Sanyal, A., Venkatesh, R. and Shah, P. Property checking array programs using loop shrinking. In: Springer. *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2018, p. 213–231.

[71] Lahiri, S., Hawblitzel, C., Kawaguchi, M. and Rebelo, H. SymDiff: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: *Proceedings of 24th International Conference on Computer-Aided Verification*. Springer, 2012, vol. 7358, p. 712–717. LNCS.

[72] Lahiri, S. K. and Bryant, R. E. Indexed Predicate Discovery for Unbounded System Verification. In: Alur, R. and Peled, D. A., ed. *Proceedings of the 16th International Conference on Computer-Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, p. 135–147. ISBN 978-3-540-27813-9.

[73] Lahiri, S. K., Bryant, R. E. and Cook, B. A Symbolic Approach to Predicate Abstraction. In: Hunt, W. A. and Somenzi, F., ed. *Proceedings of the 15th International Conference on Computer-Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, p. 141–153. ISBN 978-3-540-45069-6.

[74] Lahiri, S. K., Vaswani, K. and Hoare, C. A. R. Differential Static Analysis: Opportunities, Applications, and Challenges. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 2010, p. 201–204.

[75] Lattner, C. and Adve, V. *LLVM Language Reference Manual*. 2020. Available at: https://llvm.org/docs/LangRef.html.

[76] Laviron, V., Chang, B. E. and Rival, X. Separating Shape Graphs. In: *Proceedings of the 10th European Symposium on Programming*. Springer, 2010, vol. 6012, p. 387–406. Lecture Notes in Computer Science.

[77] Le, Q. L., Gherghina, C., Qin, S. and Chin, W.-N. Shape analysis via second-order bi-abduction. In: Springer. *Proceedings of the 26th International Conference on Computer-Aided Verification*. 2014, p. 52–68.

[78] Le, Q. L., Raad, A., Villard, J., Berdine, J., Dreyer, D. et al. Finding real bugs in big programs with incorrectness logic. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2022, vol. 6, OOPSLA1, p. 1–27.

[79] Le, Q. L., Sun, J. and Chin, W.-N. Satisfiability Modulo Heap-Based Programs. In: *Proceedings of the 28th International Conference on Computer-Aided Verification*. 2016, p. 382–404.

[80] Liu, J. and Rival, X. Abstraction of Arrays Based on Non Contiguous Partitions. In: D'Souza, D., Lal, A. and Larsen, K. G., ed. *Proceedings of the 16th International Conference on Verification, Model Checking, and Abstract Interpretation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, p. 282–299. ISBN 978-3-662-46081-8.

[81] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S. and Le Traon, Y. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*. IEEE. 2018, vol. 47, no. 1, p. 165–188.

[82] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T. F. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In: IEEE. *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. 2019, p. 1–12.

[83] Long, F., Amidon, P. and Rinard, M. Automatic inference of code transforms for patch generation. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, p. 727–739.

[84] Maksimović, P., Ayoun, S.-E., Santos, J. F. and Gardner, P. Gillian, Part II: Real-World Verification for JavaScript and C. In: *Proceedings of the 33rd International Conference on Computer-Aided Verification*. Springer-Verlag, 2021, p. 827–850.

[85] Malík, V., Hruška, M., Schrammel, P. and Vojnar, T. Template-based verification of heap-manipulating programs. In: *Proceedings of the 2018 Formal Methods in Computer-Aided Design*. 2018, p. 103–111.

[86] Malík, V., Hruška, M., Schrammel, P. and Vojnar, T. 2LS: Heap Analysis and Memory Safety (Competition Contribution). 2019. Available at: http://arxiv.org/abs/1903.00712.

[87] Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T. et al. 2LS: memory safety and non-termination. In: Springer. *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2018, p. 417–421.

[88] Malík, V., Martiček, Š., Schrammel, P., Srivas, M., Vojnar, T. et al. 2LS: Memory Safety and Non-termination (Competition Contribution). In: *Proceedings of the 24th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018, p. 417–421.

[89] Malík, V., Nečas, F., Schrammel, P. and Vojnar, T. 2LS: Arrays and Loop Unwinding. In: Springer. *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 2023, p. to be published.

[90] Malík, V., Schrammel, P. and Vojnar, T. 2LS: Heap Analysis and Memory Safety. In: *Proceedings of the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2020, p. 368–372.

[91] Malík, V., Šilling, P. and Vojnar, T. Applying Custom Patterns in Semantic Equality Analysis. In: Springer. *Proceedings of the 10th edition of the International Conference on Networked Systems*. 2022, p. 265–282.

[92] Malík, V. and Vojnar, T. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In: IEEE. *Proceedings of the 14th International Conference on Software Testing, Verification and Validation*. 2021, p. 329–339.

[93] Malík, V. and Glozar, T. *Detection of Semantic Equivalence of Program Source Codes*. U.S. Patent 11 449 317-B2, Sep. 20, 2022.

[94] Marjamäki, D. *Cppcheck: a tool for static C/C++ code analysis*. 2022. Available at: https://cppcheck.sourceforge.io/.

[95] Marušák, M. *Generic Template-Based Synthesis of Program Abstractions*. Brno, 2019. Master's thesis. Brno University of Technology.

[96] Matosevic, I. and Abdelrahman, T. S. Efficient Bottom-up Heap Analysis for Symbolic Path-based Data Access Summaries. In: *Proceedings of the 10th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2012, p. 252–263.

[97] Miné, A. The Octagon Abstract Domain. In: *Proceedings of the 8th Working Conference on Reverse Engineering*. IEEE Computer Society, 2001, p. 310–319.

[98] Møller, A. and Schwartzbach, M. I. The Pointer Assertion Logic Engine. In: *Proceedings of the 22th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2001, p. 221–231.

[99] Neamtiu, I., Foster, J. S. and Hicks, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. ACM, 2005, p. 1–5.

[100] Necula, G. C. Translation Validation for an Optimizing Compiler. In: *Proceedings of the 21th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2000, p. 83–94.

[101] Padioleau, Y., Hansen, R. R., Lawall, J. L. and Muller, G. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In: *Proceedings of the 3rd workshop on Programming languages and Operating Systems: Linguistic Support for Modern Operating Systems*. 2006, p. 10–es.

[102] Padioleau, Y., Lawall, J. L. and Muller, G. Understanding collateral evolution in Linux device drivers. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*. 2006, p. 59–71.

[103] Person, S., Dwyer, M. B., Elbaum, S. and Pãsãreanu, C. S. Differential Symbolic Execution. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2008, p. 226–237.

[104] Piskac, R., Wies, T. and Zufferey, D. Automating Separation Logic Using SMT. In: *Proceedings of the 22th International Conference on Computer-Aided Verification*. Springer, 2013, vol. 8044, p. 773–789. Lecture Notes in Computer Science.

[105] Prete, K., Rachatasumrit, N., Sudan, N. and Kim, M. Template-based reconstruction of complex refactorings. In: *Proceedings of the 26th IEEE International Conference on Software Maintenance*. 2010, p. 1–10.

[106] Raad, A., Berdine, J., Dang, H.-H., Dreyer, D., O'Hearn, P. et al. Local reasoning about the presence of bugs: Incorrectness separation logic. In: Springer. *Proceedings of the 32nd International Conference on Computer-Aided Verification*. 2020, p. 225–252.

[107] Raghavan, S., Rohana, R., Leon, D., Podgurski, A. and Augustine, V. Dex: A Semantic-graph Differencing Tool for Studying Changes in Large Code Bases. In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 2004, p. 188–197.

[108] Rakamarić, Z. and Emmi, M. SMACK: Decoupling Source Language Details from Verifier Implementations. In: *Proceedings of the 26th International Conference on Computer-Aided Verification*. Springer, 2014, vol. 8559, p. 106–113. LNCS.

[109] Ramos, D. A. and Engler, D. R. Practical, Low-Effort Equivalence Verification of Real Code. In: *Proc. of the 23rd International Conference on Computer-Aided Verification*. Springer, 2011, vol. 6806, p. 669–685. LNCS.

[110] Reps, T. W., Sagiv, S. and Yorsh, G. Symbolic Implementation of the Best Transformer. In: *Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2004, vol. 2937, p. 252–266. Lecture Notes in Computer Science.

[111] Reynolds, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2002, p. 55–74.

[112] Rinetzky, N., Bauer, J., Reps, T., Sagiv, M. and Wilhelm, R. A semantics for procedure local heaps and its abstractions. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2005, p. 296–309.

[113] Sagiv, M., Reps, T. and Wilhelm, R. Parametric Shape Analysis via 3-valued Logic. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 1999, p. 105–118.

[114] Sankaranarayanan, S., Sipma, H. B. and Manna, Z. Scalable Analysis of Linear Systems Using Mathematical Programming. In: *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, vol. 3385, p. 25–41. Lecture Notes in Computer Science.

[115] Shao, Z., Reppy, J. H. and Appel, A. W. Unrolling Lists. In: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. New York, NY, USA:

Association for Computing Machinery, 1994, p. 185–195. DOI: 10.1145/182409.182453. ISBN 0897916433.

[116] Sheeran, M., Singh, S. and Stålmarck, G. Checking Safety Properties Using Induction and a SAT-Solver. In: *Proceedings of the 2000 Formal Methods in Computer-Aided Design*. Springer, 2000, vol. 1954, p. 108–125. Lecture Notes in Computer Science.

[117] Thomson, P. Static Analysis. *Communications of ACM*. New York, NY, USA: ACM. December 2021, vol. 65, no. 1, p. 50–54. DOI: 10.1145/3486592. ISSN 0001-0782.

[118] Tristan, J.-B., Govereau, P. and Morrisett, G. Evaluating Value-Graph Translation Validation for LLVM. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2011, p. 195–205.

[119] Weiser, M. Program Slicing. *IEEE Transactions on Software Engineering*. IEEE Computer Society. 1984, no. 4.

[120] Weissgerber, P. and Diehl, S. Identifying refactorings from source-code changes. In: IEEE. *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2006, p. 231–240.

[121] Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B. et al. Scalable Shape Analysis for Systems Code. In: *Proceedings of the 20th International Conference on Computer-Aided Verification*. Springer, 2008, vol. 5123, p. 385–398. Lecture Notes in Computer Science.