BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

EQUIVALENCE-BASED SLICING OF PROGRAMS

PROŘEZÁVÁNÍ PROGRAMŮ ZALOŽENÉ NA JEJICH PODOBNOSTI

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **TATIANA MALECOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Bachelor's Thesis Specification

Student: **Malecová Tatiana**
Programme: Information Technology
Title: **Equivalence-Based Slicing of Programs**
Category: Software analysis and testing

Assignment:

1. Study existing algorithms for computing a maximum common induced subgraph and for static slicing of programs.
2. Get acquainted with DiffKemp, a tool for automatic analysis of semantic equivalence of functions in the GNU/Linux kernel.
3. Propose a method that, using results of analysis of semantic equivalence of two programs, is able to remove parts of the compared programs that are semantically equal.
4. Implement the proposed approach as a post-processing step of the DiffKemp framework. After showing two programs being non-equivalent, the tool should remove as many equivalent parts of the programs as possible while keeping the residual programs valid for further analysis.
5. Evaluate the implemented solution on at least 3 pairs of past versions of the Linux kernel. Discuss possible further usage of the sliced programs.

Recommended literature:

- Kann, Viggo. On the approximability of the maximum common subgraph problem. *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, Berlin, Heidelberg, 1992.
- Harman, Mark & Hierons, Robert. (2001). An Overview of Program Slicing. Software Focus.
- Official website of DiffKemp: https://github.com/viktormalik/diffkemp

Requirements for the first semester:

- The first two items of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Malík Viktor, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 12, 2021
Approval date: November 11, 2020

Abstract

The aim of this work is to design a method that simplifies two programs based on the results of analysis of their semantic difference. The goal is to remove as many semantically equivalent parts of the programs as possible. To find these equivalent parts, we apply our own solution to the problem of finding the maximum common induced subgraph. Subsequently, we are able to simplify the programs by using backward static slicing. By applying this simplification, we obtain sliced programs that consist of the differing parts and parts that can affect these differences. The method has been implemented as an extension of the DiffKemp tool, which is a static analyser of semantic differences between different versions of large scale programs. Our experiments on the Linux kernel show that the method is able to produce correct slices very efficiently (the analysis is prolonged only by 3.2%). Moreover, the created slices are much smaller than the original programs, which makes them suitable for further analysis.

Abstrakt

Cieľom tejto práce je navrhnúť metódu, ktorá zjednoduší dva porovnávané programy na základe výsledkov ich sémantickej analýzy. Cieľom je odstránenie čo najväčšieho množstva sémanticky ekvivalentných častí porovnávaných programov. Pre nájdenie týchto ekvivalentných častí aplikujeme vlastné riešenie problému nájdenia najväčšieho spoločného indukovaného podgrafu. Následne sme schopní zjednodušiť programy využitím spätného statického prerezávania. Aplikáciou tohto zjednodušenia získame prerezané programy, ktoré obsahujú rozdielne časti a časti programov, ktoré môžu tieto rozdiely ovplyvniť. Táto metóda je naimplementovaná ako rozšírenie nástroja DiffKemp, čo je statický analyzátor sémantických rozdielov medzi rôznymi verziami rozsiahlych programov. Experimenty vykonané na jadrách Linux-u ukazujú, že metóda je schopná veľmi efektívne vyprodukovať korektné prerezané programy (analýza sa predĺžila len o 3.2%). Navyše, vzniknuté prerezané programy sú omnoho menšie, ako originálne, čo ich činí vhodnými pre ďalšiu analýzu.

Keywords

DiffKemp, static analysis, backward static slicing of programs, maximum common induced subgraph.

Klíčová slova

DiffKemp, statická analýza, spätné statické prerezávanie programov, najväčší spoločný indukovaný podgraf.

Reference

MALECOVÁ, Tatiana. *Equivalence-Based Slicing of Programs.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

Pri práci vo veľkých projektoch môže čo i len malou zmenou implementácie ľahko dôjsť k narušeniu funkčnosti nejakej časti programu. K obmedzeniu týchto chýb sa využivajú aj statické analyzátory, medzi ktoré patrí aj nástroj DiffKemp. Nástroj DiffKemp je analyzátor sémantických rozdielov medzi rôznymi verziami programu. Používaním tohto nástroja sú vývojárovi poskytnuté informácie o syntaktických rozdieloch dvoch verzií jeho programu a sčasti aj elementárne sémantické odlišnosti. Sémantická analýza ekvivalencie program je všeobecne náročný problém, ktorý úspešne riešia nástroje využívajúce formálne metódy. Takéto metódy nezvládajú rozsiahle programy a sú pomalé, no presné.

DiffKemp si zakladá na vysokej škálovateľnosti, ktorú dosahuje využitím niekoľkých konceptov. Spočiatku sú preložené dve verzie programu do jazyka LLVM IR. Preklad do LLVM IR je veľmi efektívny len pre dva porovnávané programy, ktoré sú identické. Avšak, môže vyvolať mnoho falošných výsledkov. Aby DiffKemp nebral do úvahy tieto falošné výsledky, využíva koncept rôznych transformácií kódu. Následne aplikuje koncept porovnávania programov po jednotlivých inštrukciach. V prípade, že tento koncept nie je možné aplikovať, snaží sa na daný kus kódu aplikovať vopred definované vzory zmien zachovávajúce sémantiku.

Nástroj DiffKemp sa typicky používa pre dve verzie rovnakého programu, kde môžeme predpokladať, že veľké časti programov sú zhodné. Preto môžeme tieto zhodné časti odstrániť, čím sa rozsiahlosť programov zredukuje. Následne môžeme funkcie, ktoré DiffKemp porovnal ako neekvivalentné predať nástrojom založených na formálnych metódach, aby sme dosiahli presnejší verdikt sémantického porovnania. Odstránením ekvivalentných častí z funkcií by mohli byť nástroje založené na formálnych metódach schopné dané funkcie spracovať a vyhodnotiť v rozumnom čase. Pre nájdenie ekvivalentných častí určených k odstráneniu aplikujeme vlastné riešenie problému nájdenia najväčšieho spoločného indukovaného podgrafu. Grafový algoritmus je možné aplikovať vďaka jazyku LLVM IR, ktorý reprezentuje funkcie ako grafy riadenia toku programu. Po nájdení najväčšieho spoločného podgrafu sme schopní odstrániť všetky ekvivalentné časti a dané funkcie zjednodušiť. Pre zredukovanie funkcií využívame techniku prerezávania programov, konkrétne spätné statické prerezávanie. Prerezávanie je ukutočnené na základe prerezávacieho kritéria, vďaka ktorému sa odstránia určité časti programu. V našom prípade sú prerezávacími kritériami všetky nájdené rozdiely. Konkrétne spätné statické prerezávanie zachová okrem prerezávacieho kritéria aj všetky časti kódu, ktoré mohli ovplyvniť zadefinvané kritérium. Aplikáciou prerezávania sa získajú prerezané programy. Výsledné prerezané programy, ktoré vzniknú metódou navrhnutou v tejto práci, obsahujú rozdielne časti a závislé časti, ktoré môžu tieto rozdiely ovplyvniť.

Navrhnutá metóda je uspešne zakomponovaná v nástroji DiffKemp ako následné spracovanie dvoch funkcií, ktoré nástroj určil ako neekvivalentné. Na základe validácie je ukázané, že implementované riešenie nenarušilo funknčnosť nástroja DiffKemp a vyprodukované prerezané funkcie si zachovali sémantiku v rozdielnych častiach kódu. Ďalej je na základe experimentov ukázaná efektivita metódy. Efektivita spočíva vo vytvorení funkcií obsahujúcich o viac než polovicu menej inštrukcií ako pôvodné neprerezané funkcie. Okrem efektívneho zredukovania počtu inštrukcií je časová náročnosť vyššia o približne 3.2% pôvodného času porovnania.

Equivalence-Based Slicing of Programs

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Viktor Malík. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Tatiana Malecová
May 9, 2021

Acknowledgements

I would like to thank Ing. Viktor Malík, the supervisor of my work, for his willingness, great advice and help in elaborating the bachelor's thesis.

Contents

Chapter 1

Introduction

During development of large projects, even a tiny code change can easily disrupt functionality of some parts of the program. For finding these disruptions, one of the options to use is *static analysis of program semantics differences*. Generally, the goal of static program analysis is to collect some information about the behaviour of a program from its source code without executing it under its original semantics [19]. Static analysis has several forms, such as bug-pattern searching, dataflow analysis, or abstract interpretation. Nowadays, many static analysis tools exist, such as Clang static analyser, Cppcheck and others. In this thesis, we deal with DiffKemp, a tool that uses methods of static analysis to find out whether two versions of the same program have the same semantics.

The existing methods in the area of static analysis of semantic equivalence are either *fast and inaccurate* (they can cause many false positives) or *slow and accurate* (their comparison takes a long time). DiffKemp takes a middle ground – it is reasonably fast while being able to provide the user with information about differences between two versions of a program while ignoring changes that do not affect semantics. The main goal of the DiffKemp tool is high scalability to real source code. To achieve high scalability, it uses several different techniques. For analysis, DiffKemp translates the two versions of a program into the LLVM intermediate representation (LLVM IR) language. Then, it tries to compare the programs instruction by instruction. This instruction-by-instruction comparison works only on parts of the programs that are syntactically the same. Therefore, to make the comparison more likely to succeed, various semantics-preserving code transformations of the compared programs are performed. If the instruction-by-instruction comparison is not possible, DiffKemp tries to match the differing pieces of code to one of predefined semantics preserving change patterns.

Despite these features, DiffKemp still cannot handle some complicated refactorings. Such complex refactorings could be handled by more heavy-weight methods for analysis of semantic difference, however, the compared real world programs are usually large and such methods do not scale on them. On the other hand, when dealing with two versions of the same program, it can be assumed that large parts of the compared programs are very similar and can be handled by DiffKemp. One of the possible solutions would be to use DiffKemp to identify those parts and then to apply more advanced methods only to the remaining pieces of code. To facilitate such approach in this work, we propose a method which, based on the DiffKemp comparison, simplifies the compared programs so that only the different parts remain.

One of the most common analysis techniques for simplifying programs is program slicing. Program slicing is used for removing parts of a program that do not affect some point

of the program, usually denoted as the slicing criterion. In this thesis, we use backward static slicing to remove as many equivalent parts of the compared programs as possible. Hence, our slicing criteria are the differing parts of the programs. To find the equivalent parts of functions, we search for the solution to the maximum common subgraph problem (since LLVM represents program functions using control-flow graphs). After application of backward static slicing, the final program slice contains the different parts of the program and every instruction that could affect them. After the slicing, the sliced functions can be given to some slow and accurate method for getting more exact result.

The thesis is organised as follows. Chapter 2 is devoted to the description of the DiffKemp tool, its intermediate representation, and algorithm for checking semantic equivalence of functions. Chapter 3 is devoted to program slicing, forms of slicing, and the problem of finding the maximum common subgraph. Chapter 4 is devoted to the proposed method as a new extension of DiffKemp that removes as many equivalent parts of programs as possible. Chapter 5 is devoted to the architecture of DiffKemp and integration of the proposed method to DiffKemp. Chapter 6 is devoted to validation of the implementation, regression tests, and experimental evaluation.

Chapter 2

DiffKemp

DiffKemp[1] is a highly-scalable tool for analysing semantic differences between two versions of a program [15]. It uses methods of static analysis to automatically determine differences in functions between two different versions of a software, with a special focus on the GNU/Linux kernel. To achieve high scalability, DiffKemp performs the comparison using three concepts:

- It applies **instruction-by-instruction comparison** on the level of **LLVM intermediate representation** (LLVM IR). More details about LLVM IR are provided in Section 2.1. This technique is very efficient for unchanged parts of programs, however it may lead to many false results.
- To avoid a part of these false results, DiffKemp pre-processes the code using various **code transformations** and static analyses. After the pre-processing, DiffKemp tries to apply instruction-by-instruction comparison as often as possible.
- In case that this comparison is not sufficient, DiffKemp tries to match different parts of code with predefined **semantics preserving change patterns**.

Description of the algorithm for checking equivalence of functions is provided in Section 2.2 in more detail.

Thanks to the mentioned concepts, DiffKemp achieves higher scalability than tools based on heavy-weight formal methods, e.g., LLReve [13]. Such tools concentrate on sound equivalence checking using, e.g., automata or logics, and they are not able to process large programs. At the same time, DiffKemp is able to identify much more refactorings than light-weight methods while it maintains a performance similar to these approaches. The most known light-weight tool is the Unix diff tool [12] that is based on line-by-line lexical comparison. The experiments that show the contribution of DiffKemp tool into the field of program analysis can be found in article [15].

This chapter is organised into two sections. In Section 2.1, we describe LLVM IR as the intermediate representation of DiffKemp. We concentrate on specific constructions of LLVM IR that are important for our work. Then, we introduce several formalisms to represent analysed programs and at the end of the section, we provide an example of the intermediate representation of a function. In Section 2.2, we describe the algorithm for checking semantic equivalence that DiffKemp uses.

[1] **Diff**erence of **K**ernel functions, **m**odules, and **p**arameters [14]

2.1 Representation of the Program

Instead of analysing programs directly in the C language, DiffKemp uses a lower level language to make the analysis easier. Specifically, it uses the internal representation of the LLVM/Clang compiler, the LLVM *Intermediate Representation* [2]. This strongly typed intermediate language is often used for program analysis due to its independence from the source and the target code. It is an instruction-based program representation that represents functions by *control flow graphs* (CFG). A CFG is a tuple *(BB, E)* where *BB* is a finite set of basic blocks and *E* is a set of directed edges that connect the blocks and express the control flow of the program. *A basic block* is a sequence of instructions. In this sequence, there are no incoming or outgoing branches in the middle of the block. An incoming branch always leads to the first instruction of the basic block and an outgoing branch always leads from the last instruction of the basic block, a so-called *terminator instruction*. There are several basic types of terminator instructions:

1. A **branching instruction** defines the control flow within the function, and it can be:
   (a) A **conditional branching instruction** that contains a boolean condition and references to two basic blocks. Based on the evaluation of the condition, one of the blocks is chosen as the following block in the control flow. We denote the block that is followed if the condition is evaluated to true as the *true-case successor* and the block that is followed when the condition is evaluated to false as the *false-case successor*.
   (b) An **unconditional branching instruction** references a single basic block that will be performed next.
2. A **return instruction** returns the control flow back to the caller function and it optionally returns a value.

Each function has a single *entry block* and may have several *exit blocks*. *An instruction* performs an operation over a list of operands, which can be empty. *An operand* of the instruction may be a variable, a constant, or a function. LLVM IR distinguishes two kinds of variables:

1. **global variables** that correspond to the global variables of the original program and
2. **local variables** that have two categories [17]:
   (a) **stack-allocated local variables** that are created by allocation of variables on the stack frame of the currently executing function. The allocation is done by using the `alloca` instruction. These usually correspond to the local variables of the original program.
   (b) **register-allocated local variables** (also calles *registers*) that are used to hold intermediate values in the function, typically results of instructions. Each CFG satisfies the *static single assignment form* that requires assignment to each variable at most once, i.e., every time an instruction returns a value, a new local variable is created [5, 15]. In case that values from multiple basic blocks converge into one variable, a **phi** instruction is created. This instruction selects a value based on the fact which basic block was executed prior to the current basic block [2].

In LLVM IR, global and stack-allocated local variables are always referenced using pointers. Therefore, reading from and writing to these variables is always done using `load` and `store` instructions, respectively.

For the purpose of the following explanation, we introduce several formalisms to represent the analysed programs. Let F be the set of all functions of the program. For a function $f_i \in F$, let:

- I_i be the set of all instructions in f_i,
- L_i be the set of all local variables in f_i,
- G_i be the set of all global variables used in f_i,
- P_i be the list of all parameters of f_i,
- $V_i = L_i \cup G_i \cup P_i$ be the set of all variables used in f_i,
- C_i be the set of all constants used in f_i.

For each instruction (except the return instruction), we also define its *successors* - the set of instructions immediately following it. For branching instructions, these are the branch targets and for other instructions, it is only the following instruction inside the basic block. To formalise this, we introduce several functions:

- $succ : I_i \to I_i$ defines the successor of an unconditional branching and a non-branching instruction,
- $succT : I_i \to I_i$ defines the true-case successor of a conditional branching instruction,
- $succF : I_i \to I_i$ defines the false-case successor of a conditional branching instruction.

2.1.1 LLVM IR in Practice

In this subsection, we give an example of what the intermediate representation of a program may look like. At first, we need to define the forms of the LLVM IR. This language is designed to be used in three equivalent forms:

1. an on-disk bitcode representation,
2. an in-memory compiler IR,
3. a human-readable assembly language representation.

In this thesis, we work with the human-readable form of LLVM IR. To obtain this form, the source files are compiled using Clang with the following command:

```
clang -emit-llvm -S source_file.c
```

LLVM IR recognises two types of identifiers – local and global identifiers. *Global identifiers* include functions and global variables. *Local identifiers* are registers (local variables), basic block labels, and type definitions. Whereas global identifiers begin with the character '**@**', local variables are denoted by the character '**%**'. Moreover, LLVM IR may contain metadata marked with the character '**!**'.

At first, let us consider the following example of a simple function in C language to see how it is represented in the LLVM language:

```
1 // Get absolute value of the given number
2 int abs(int x) {
3   return (x < 0) ? -x : x;
4 }
```

Listing 2.1: Function in the C language that returns absolute value of given number.

The LLVM representation of the `abs` function shown in Listing 2.1 can be found in Listing 2.2. It represents a definition of the function named **abs** that returns an integer and has a single parameter of an integer type. The entry block is at lines 3-7, and it contains an allocation of a new stack variable (line 3), storing the value of the parameter to the new stack variable (line 4) and its loading to a new register (line 5). Then, the local variable is compared to 0 (line 6). Based on the result, the control flow of the function will be transferred to one of the following basic blocks. The transfer is done by the conditional branch instruction at line 7.

Besides the entry basic block, the function contains three more basic blocks. The one with label `'5'` is the true-case successor of the entry block, and it creates a negation of the given parameter. The one with label `'8'` is the false-case successor of the entry block, and it simply loads the given parameter to a local variable. The last block with label `'10'`, is the function exit block, and it contains a `phi` instruction which selects a value based on the basic block from which the control flow came. This value is then returned from the function.

```
1  ; Function Attrs: noinline nounwind optnone ssp uwtable
2  define i32 @abs(i32 %0) #0 {
3    %2 = alloca i32, align 4
4    store i32 %0, i32* %2, align 4
5    %3 = load i32, i32* %2, align 4
6    %4 = icmp slt i32 %3, 0
7    br i1 %4, label %5, label %8
8
9  5:                                                ; preds = %1
10   %6 = load i32, i32* %2, align 4
11   %7 = sub nsw i32 0, %6
12   br label %10
13
14 8:                                                ; preds = %1
15   %9 = load i32, i32* %2, align 4
16   br label %10
17
18 10:                                               ; preds = %8, %5
19   %11 = phi i32 [ %7, %5 ], [ %9, %8 ]
20   ret i32 %11
21 }
```

Listing 2.2: The LLVM IR representation of the `abs()` function.

2.2 Function Equality

The main goal of DiffKemp is to check whether two compared functions (typically being two versions of the same program) are semantically equivalent. Informally, two functions are semantically equivalent if, for the same input, they produce the same output. By input, we understand the values of parameters of the function and the initial state of the memory. By

output, we understand the return value of the function and the final state of the memory[2]. Checking of semantic equality is a difficult problem, and therefore DiffKemp divides the compared functions into smaller parts using so-called *synchronisation points*. A set of points is placed into each function, and a one-to-one mapping (synchronisation) is created between the two sets. Then, DiffKemp only compares semantics of every piece of code between two succeeding synchronisation points to the semantics of the piece of code between the two synchronised points in the other function. In other words, the synchronisation points denote places where the functions are in a semantically equivalent (synchronised) state. Typically, synchronisation points are placed at each instruction. When this is the case, DiffKemp performs the so-called *instruction-by-instruction* comparison of the programs. As this would normally be possible only for exactly the same programs, DiffKemp performs various semantics-preserving transformations of the compared programs. This can be, e.g., inlining, dead code elimination, constant propagation, or redundant code elimination such that instruction-by-instruction comparison can be made as often as possible. In addition, DiffKemp also supports a possibility that synchronisation points cannot be placed at each instruction. In such a case, it tries to match the corresponding pieces of code to one of predefined *semantics-preserving change patterns*. If the codes are matched, they are evaluated as semantically equal. To simplify the presentation in this thesis, we will only consider the instruction-by-instruction comparison (cf. [15] for the full algorithm).

More formally, when comparing functions f_1 and f_2, the goal is to find the following:

- $S_1 \subseteq I_1$, $S_2 \subseteq I_2$: the sets of synchronisation points in f_1 and f_2,
- $smap : S_1 \leftrightarrow S_2$: the function representing the mapping of synchronisation points,
- $varmap : V_1 \leftrightarrow V_2$: the function for mapping between local and global variables, as the functions may generally use different variables.

The process of comparing two functions f_i, for $i \in \{1, 2\}$, is shown in Algorithm 1. At first, the transformations are done (line 1) for easier instruction-by-instruction comparison. After that, if the numbers of function parameters are different, the functions are considered as non-equal (lines 2-3). Otherwise, parameters of functions are mapped by their order in *varmap* (lines 5-6). Also, the global variables used in functions are mapped by their name in *varmap* (lines 7-8). In the main loop, DiffKemp works with the queue Q that stores pairs of synchronisation points that need to be compared. In each iteration of the loop, the following is done:

1. A synchronisation pair is taken from Q (line 11).
2. Check, whether synchronisation points s_1 or s_2 are already in synchronisation maps and if they are mapped to each other (lines 12-13).
3. Function `cmpInst` compares instructions s_1 and s_2 (lines 14-15). The `cmpInst` function performs comparison of the instruction operations and operands. As we mentioned in Section 2.1, an operand can be of three types:
   - *Variable* - the function checks if they are mapped in *varmap*.
   - *Constant* - the function checks if the values are equal.

[2]This is a simplified definition that does not consider function termination or side effects. For a complete definition, see [15].

Algorithm 1: Checking semantic equivalence of functions.

Input : Functions f_1, f_2
Output: *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  for 1 ≤ i ≤ |P1| do
6  |  varmap(p1_i) = p2_i
7  for g1 ∈ G1 do
8  |  varmap(g1) = g2 ∈ G2 s.t. g1 has the same name as g2
   // Main loop
9  Q = (i1_in, i2_in)
10 while Q is not empty do
11 |  take any (s1, s2) from Q
12 |  if s1 ∈ S1 or s2 ∈ S2 then
13 |  |  check if s1 is mapped to s2
14 |  if ¬cmpInst(s1, s2) then
15 |  |  return false
   |  // Update synchronisation sets and maps
16 |  S1 = S1 ∪ {s1}, S2 = S2 ∪ {s2}
17 |  smap(s1) = s2
   |  // Results of instructions are stored in new local variables
18 |  varmap(v1) = v2
19 |  if s1 is conditional branch then
20 |  |  insert (succT(s1), succT(s2)) to Q
21 |  |  insert (succF(s1), succF(s2)) to Q
22 |  else if s1 is not return then
23 |  |  insert (succ(s1), succ(s2)) to Q
24 return true
```

- *Function* - the function recursively calls Algorithm 1 to check if the called functions have the same semantics.

4. When two instructions are compared as non-equal, functions are claimed to be non-equal and the comparison ends (lines 14-15).

5. Otherwise, synchronisation sets and maps are updated (line 16) by inserting the s_1 and s_2 instructions into synchronisation sets. A synchronisation between s_1 and s_2 is created (line 17). Newly created variables are defined and mapped (line 18).

6. Successors of s_1 and s_2 are found and inserted into Q for continuing in the synchronous traversal (lines 19-23). The successors of these instructions depend on the type of the compared instructions.

When the queue Q is empty, i.e., DiffKemp compared the entire functions and all instructions were synchronised, the functions are declared as semantically equal [15].

Chapter 3

Slicing of LLVM Programs

The thesis aims to simplify programs based on their semantic differences. The differences are found in the previous analysis, which is done at the level of LLVM IR, which represents the functions by control flow graphs. Program simplification (often referred to as *program slicing*) is a well-known and widely used technique. Our simplification is specific because we have two input functions and we simplify both of them simultaneously based on some relations (differences) between them. Therefore, the traditional slicing is not sufficient, as we are not dealing with a single program. Since the functions are represented using CFGs, we combine slicing techniques with *graph algorithms*. In particular, our simplification of functions represented by CFGs uses two concepts:

1. **static backwards program slicing** that is one of the traditional slicing techniques and
2. graph algorithms that find similarity of graphs, specifically the **maximum common subgraph**.

This chapter is organised into two sections, where we describe the above two concepts. In Section 3.1, we describe the main principles of slicing, its importance for this thesis, and we focus on static slicing and its two different forms. In Section 3.2, we describe the problem of finding a maximum common induced subgraph, and we provide a reason why this problem is significant for this thesis.

3.1 Program Slicing

Analysis of programs is typically a challenging problem, mainly in the sphere of static analysis. For many applications of static analysis, it has been found that it is not necessary to analyse the entire program. Often, it is enough to analyse only some of the program's parts related to the verified property [3]. The technique for simplifying programs that is used to speed up and streamline analysis of a program is a so-called **program slicing** [4, 9] introduced by Mark Weiser [20]. Informally, it removes parts of a program that are unnecessary due to the specified property. After the removal, a *slice* is constructed. A program slice, which can be executable, is an extraction of a program that influences or is influenced by the **slicing criteria**. The slicing criterion can be typically a *value of the variable*, which means that the parts of the program that are affected or affect the value of the variable are sliced. The slicing criterion can also be *reachability*, which means that the parts of the program from which it is not possible to get to a certain point of the program

are sliced. Formally, the slicing criterion is defined as a pair $< p, V >$, where p is a program point, and V is a subset of program variables [6]. The program slice is defined as a subset of program statements that do not change the behaviour of the original program according to program point p, whereas the variables included in V are the same in the original program and slice.

Simplification of programs for the purpose of acceleration and streamlining of program analysis is nowadays highly desired and beneficial. Therefore, applications of program slicing can be found in many branches of information technologies such as differencing, testing, debugging, re-engineering, program comprehension, and software measurement [18].

There exist several different types of slicing based on various criteria. The original program slice introduced by Mark Weiser is nowadays called *executable backward static slice*, where:

- *Executable* because the final slice must be an executable program. In some applications of slicing, such as debugging, executability is not required.
- *Backward* because the slicing is done from the slicing criteria whereas only the previous statements are considered (simply put, the traversal is done from target to source). More information about backward slicing is provided in Subsection 3.1.1. Besides backward slicing, there also exists *forward* slicing that is also described in Subsection 3.1.1 (simply put, the traversal is done from target to the end).
- *Static* because the slice is computed as the solution to a static analysis problem. This means that static slicing keeps every statement of the program that may affect the slicing criteria in every possible execution of the program, i.e., without considering the program inputs, unlike dynamic slicing.

Besides these common types, there are other special program slicing techniques, i.e., conditioned slicing, amorphous slicing, and quasi slicing.

Since this thesis aims to simplify two programs so that only semantic differences are kept, we can look at this problem as a problem of slicing. As we have the source codes available and the complete analysis of DiffKemp uses static analysis, we will review **static slicing**. Moreover, as we need to keep all parts of programs that may influence the parts with identified differences, we will use backward slicing. Finally, as the sliced programs are intended for further analysis, we will require executable slices.

3.1.1 Forward vs Backward Slicing

Now, we show the difference between forward and backward slicing, since these are the two most important types of static slicing. Both of them produce different program slices. To see what particular slices may look like, consider an example of function `fce` written in C in Listing 3.1. To produce the final slice, we need to determine the slicing criteria first. After that, we can choose one of the approaches of static slicing. Therefore, let us declare the statement `int res = sum(a,b)` at line 7 as the slicing criterion.

Forward slicing was introduced by Horwitz in [10]. Forward slicing keeps program statements that *are affected* by the slicing criterion. Program slice produced by application of forward slicing is shown in Listing 3.2. Only the statement at line 8 is affected by the slicing criterion, therefore everything else is removed. As we can see, the final slice *can not* be executable, since the variable `b` is not defined. There exist forms of forward slicing that

produce an executable slice. However, we will not analyse those in this thesis since forward slicing is not the goal of our work (cf. [8] for executable forward slicing).

By applying backward slicing, the program slice contains program statements that *can affect* the slicing criterion [9]. Informally, it works as follows:

1. For each statement in the slicing criterion, take all its operands and add to the slice those statements that change the value of that operand.
2. Then, repeat the same recursively for each statement added.

Program slice produced by backward slicing on the provided example is shown in Listing 3.3. Statement at line 2 is not relevant according to the selected slicing criterion, hence it is removed. Statement at line 8 does not affect the slicing criterion since it occurs after the criterion, hence it is removed. Other statements have to stay since they can affect the slicing criterion, hence they are relevant. In this thesis, we propose a method that uses backward static slicing.

```
1 int fce (int a) {
2   printf("fce\n");
3   int b;
4   scanf("%d", &b);
5   if (b < 0)
6     b = b * -1;
7   int res = sum(a,b);
8   return ++res;
9 }
```

Listing 3.1: Source code before the slicing.

```
1 int fce (int a) {
2
3
4
5
6
7   int res = sum(a,b);
8   return ++res;
9 }
```

Listing 3.2: The forward slice.

```
1 int fce (int a) {
2
3   int b;
4   scanf("%d", &b);
5   if (b < 0)
6     b = b * -1;
7   int res = sum(a,b);
8
9 }
```

Listing 3.3: The backward slice.

3.2 Maximum Common Subgraph

In most cases, during comparing two functions, where one is refactoring the other, a significant part of functions is equal, and functions differ only in small parts. In the simplification process of such functions, we want to reduce functions as much as possible by applying the slicing technique to consist only of parts of code that differ plus their dependencies to preserve semantics. As we outlined in the introduction of this chapter, traditional slicing is not sufficient here, because the preceding difference analysis (described in Algorithm 1) only analyses the functions up to the point where the first difference is found. It is very likely, though, that large parts of the functions after (in the sense of control-flow) the first difference are equivalent. In order to find such parts, we analyse the graph problem (since functions are CFGs), whose task is to find as many common parts of two given graphs as possible. This problem is commonly known as the problem of finding a maximum common subgraph. After finding these equivalent parts, we can apply slicing to remove common parts. Before explaining what the maximum common subgraph is, we need to understand its background [16].

In Section 2.1, we explained that each function in LLVM IR is represented by a control flow graph (CFG). For a quick reminder, CFG consists of finite set of basic blocks (BB) and a set of directed edges (E) that connect the blocks and express the control flow of the program. Let $G = (BB, E)$ and $G' = (BB', E')$ be two graphs:

- G' is a **subgraph** of G if it is a graph where $BB' \subseteq BB$ and $E' \subset E$.

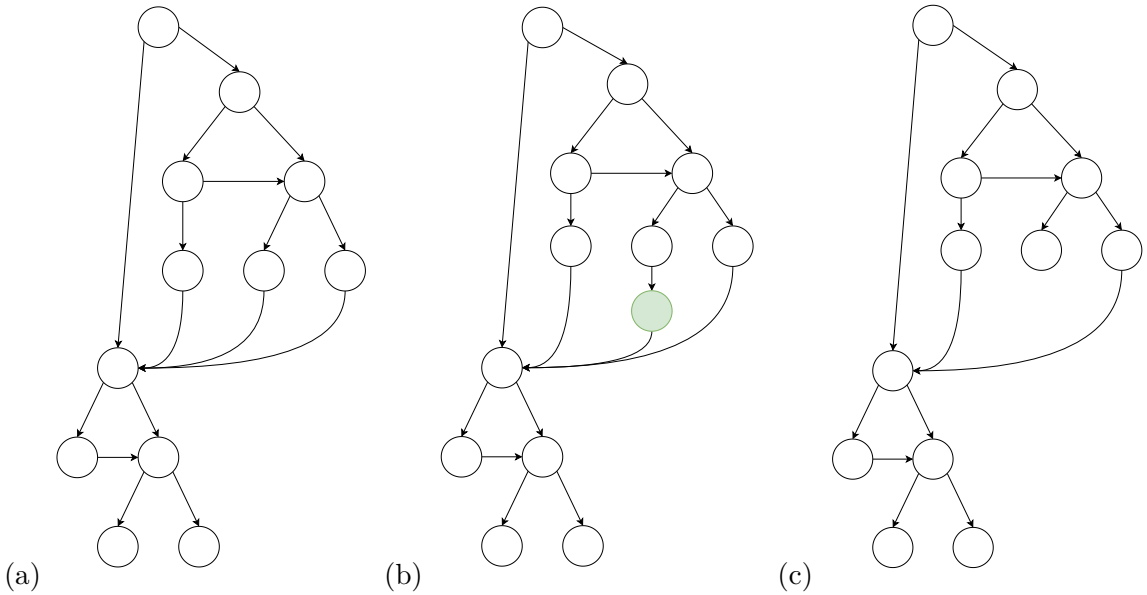


Figure 3.1: (a) an old version of function (b) a new version of function (c) a maximum common subgraph

- A **common subgraph** of G and G' is a graph that is isomorphic to some subgraphs of G and G'. A graph, G is *isomorphic* to G' if there exists a bijective function $f : BB \to BB'$ which preserves edges, i.e., $\forall(u, v) \in BB \times BB, (u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.
- G' is an **induced subgraph** of G if $BB' \subseteq BB$ and $E' = E \cap (BB' \times BB')$. Informally, we get G' by removing all blocks of G which are not in BB' and keeping all edges whose source and destination blocks are both in BB'.
- Finally, we can define a **maximum common subgraph** (MCS) that is the largest common subgraph of G and G'. We distinguish two types of MCS [7]:
  1. *Maximum common edge subgraph* (MCES) is the largest induced subgraph common to G and G' in the term of the number of edges.
  2. *Maximum common induced subgraph* (MCIS) is the largest induced subgraph common to G and G' in the term of the number of basic blocks. In this thesis, we focus only on MCIS.

Finding MCS is an optimisation NP problem (also called *NPO problem*) of the *subgraph isomorphism* problem, which is a famous NP-complete problem [11]. A problem is categorised as NP if it can be solved in polynomial time by a non-deterministic Turing machine (cf. [21] for details). If a solution to the NP problem is known, the demonstration of the solution correctness can always be reduced to P (polynomial bounds time of solution). If P is not equivalent to NP, the solution to the NP problem requires an exhaustive search in the worst case.

In the MCS problem, we have a pair of graphs, and the task is to find the largest induced subgraph common to the given graphs [1]. For a better understanding, consider two functions represented as CFGs shown in Figure 3.1. As we can see, CFG presented in (b) refactors CFG presented in (a) by adding some additional code (the green circle). Other parts are equivalent. Therefore, the MCS of these functions is equivalent to the subgraph

shown in (c). After this, when the common parts of two functions are found, we can apply static backward slicing, which preserves differing parts of the presented CFGs and every statement that can affect differences. In the case presented in Figure 3.1, the final program slice from the old version of the function would be empty. The final program slice from the new version of the function would contain the basic block represented by the green circle and all basic blocks with dependent instructions. We consider an instruction dependent if it can affect the execution of a differentiating instructions (any instruction in the green basic block).

Several algorithms solve the MCS problem with various worst-case run-time, such as brute-force and backtracking algorithms, usage of compatibility graphs, and usage of vertex cover [1]. Finding MCS is often associated with finding a maximum clique (usage of compatibility graphs). A clique is a set of such vertices in a graph that each vertex is connected to every other vertex [7]. The maximum clique is the largest set of pairwise compatible pairs [1]. Therefore, optimisation algorithms for finding maximum clique can be used to find MCS. In this thesis, we use our own algorithm for finding the MCS that is based on a simple exhaustive search. Thanks to some properties that CFGs have (e.g., they have a single entry node or a maximum of two outgoing edges for each node), our approach is much more efficient than the classic brute-force approach. The method proposed in this thesis that solves the maximum common subgraph of comparing function in LLVM IR and subsequently slices as many equivalent parts of code as possible is described in Chapter 4.

Chapter 4

Equivalence Slicing

This thesis aims to simplify two nonequivalent functions as much as possible based on the previous results of the analysis of semantic equivalence of two programs. The previous analysis is described in Algorithm 1, which ends at the moment when it finds two non-equal instructions. This means that parts of the functions located behind the first difference (in the sense of control flow) remain unanalysed. However, these may contain large pieces of code that are semantically equal between the compared versions. To find these semantically equivalent parts, we will search for the maximum common subgraph as described in Section 3.2. In particular, once we have found a difference, we try to find the first pair of instructions that are reachable from the difference and from which the sychronisation of the compared functions can be restored. We denote this pair of instructions as the *next synchronisation pair*. Moreover, since finding MCS is generally very hard, we split our MCS algorithm into two phases, as described below. Once we have found the MCS, we use backward static slicing to remove parts of the functions that are semantically equivalent. We described slicing in Section 3.1. Since we apply backward static slicing, we remove each instruction from equivalent parts of functions that *can not affect* differing instructions.

Formally, when analysing functions f_1 and f_2, the goal is to find so-called *keep sets* $K_1 \subseteq I_1$ and $K_2 \subseteq I_2$ that contain differing instructions.

Because of a more comprehensive solution, we propose a method that is logically divided into two phases:

1. The first phase finds **maximum common subgraph** of the compared functions. This phase consist of two subphases:

   (a) In the first subphase, we are finding the MCS on the level of entire basic blocks by following the control flow. In other words, we search for sychronisation of blocks that are successors of some previously sychronised blocks, starting from entry blocks of the compared functions. This step can be done in a quite efficient way and it is usually able to find a synchronisation in large parts of functions.

   (b) In the second subphase, we are finding the MCS on the level of individual instructions (i.e., finding the next sychronisation pair), but our search is limited for blocks that were not sychronised in (a). This is a more time-consuming step, but it is usually performed on small parts of the functions only.

2. The second phase produces a slice by **keeping instructions** compared as different and their dependencies. All other instructions are removed from the functions.

Algorithm 2: The full algorithm of slicing.

```
   Input  : Functions f_1, f_2
   Output: sliced f_1, f_2
1  K_1 = {}, K_2 = {}
2  Q = (bb^1_in, bb^2_in)
   // First phase
3  while Q is not empty do
       // First subphase
4      (DiffInst_1, DiffInst_2) = findMCSBasicBlocks(Q)
       // Second subphase
5      (K'_1, K'_2) = findMCSInsts(DiffInst_1, DiffInst_2)
6      K_1 = K_1 ∪ K'_1
7      K_2 = K_2 ∪ K'_2
8      insert all successors of newly synchronised basic blocks to Q
   // Second phase
9  f_1 = sliceFunction(f_1, K_1)
10 f_2 = sliceFunction(f_2, K_2)
```

Since the solution consists of more phases, the chapter is organised as follows. In Section 4.1, we introduce the top-level algorithm of the proposed solution. In Section 4.2, we propose a method that finds the maximum common induced subgraph of two given control flow graphs on the level of basic blocks and on the level of instructions. In this section, we also introduce a function that compares two basic blocks starting from given instructions. This function is used in both subphases of finding the MCS. In Section 4.3, we propose a method that removes any possible semantically equivalent parts of programs based on the found subgraph.

4.1 Top-level Algorithm

At the beginning of this chapter, we introduce the top-level algorithm that removes equivalent parts of two given programs. This algorithm is shown in Algorithm 2. As we mentioned in the introduction of this chapter, the solution consists of two phases. In the first phase, we are working with a queue Q that stores pairs of basic blocks of functions f_1 and f_2 that were not analysed yet. The algorithm works as follows:

1. We initialise the keep sets K_i for $i \in \{1, 2\}$ for collecting the instructions that are different (line 1).

2. We start the analysis from entry blocks of both functions that are inserted to the queue (line 2).

3. To remove every equivalent part of functions, we generalise the method of finding MCS for any number of differences and subsequent search of synchronisation. Hence, we put the process of finding the MCS into a loop that guarantees the analysis of each basic block of function (line 3). Inside the loop, the following is done:

   (a) We find the MCS of functions on the level of basic blocks by following the control flow. For each pair of basic blocks in Q, we compare their semantics and

if they are equal, we continue by synchronising their successors and subsequently comparing them. Basic blocks that contain differences or that are not successors of any synchronised basic block remain unanalysed. This step is performed in function `findMCSBasicBlocks`, which operates on the queue Q and it also returns the first pair of identified differing instructions denoted $DiffInst_1$ and $DiffInst_2$ for f_1 and f_2, respectively.

(b) We follow by finding MCS on the level of instructions inside the basic blocks that remained unanalysed after the previous step. This search aims to find the next synchronisation pair of instructions which denote the beginning of some semantically equivalent parts of functions. While searching for such instructions, we also collect all skipped (i.e., differing) instructions. This step is performed in function `findMCSInsts` which takes the first differing pair of instructions as the input and returns so-called keep sets (i.e., the sets of instructions that were compared as non-equal and must be thus kept in the final slice). The contents of the returned sets K_1' and K_2' are added to the main keep sets K_1 and K_2. For more details about finding MCS on the level of basic blocks and instructions, see Section 4.2.

(c) After a synchronisation is found, we repeat this process, starting from the successors of the newly synchronised blocks (we insert the successors into Q in line 8). The loop ends once all basic blocks were compared and Q is empty.

4. The second phase starts when the entire functions were analysed. Now, we can slice them to keep only differing instructions and their dependencies. The slicing is done in function `sliceFunction` (lines 9-10). This function gets a function for slicing f_i and a set K_i that contains instructions that must be kept. For more details about applying backward static slicing in our solution, see Section 4.3.

4.2 Finding Maximum Common Subgraph

As we already know, we are usually analysing two versions of the same program, i.e., most parts of functions are semantically equal, and only some parts differ. To find each semantically equivalent part, we need to analyse the entire functions. Therefore, as the first step, we analyse as many function parts as possible to narrow down the detection of the different parts and subsequent search for synchronisation. In this section, we propose a function that finds the maximum common subgraph of two given functions represented as two control flow graphs.

The solution of finding the maximum common subgraph is divided into two subphases, as we mentioned in the introduction. In both of them we use special function that compares given basic blocks from given instructions. Therefore, this section consists of three subsections. In Subsection 4.2.1, we review the comparison of two basic blocks from given instructions. In Subsection 4.2.2, we review the finding of MCS on the level of basic blocks. In Subsection 4.2.3, we review the finding of MCS on the level of instructions.

4.2.1 Comparison of basic blocks

In our method, we often compare instructions without previously comparing the instructions that are located before them (in the control flow). Therefore, it often happens that the

Algorithm 3: A function that compares basic blocks from given instructions until the end of basic blocks.

Input : synchronisation points s_1, s_2
Output: *true* if basic blocks are equal, else $false$
Function `cmpBasicBlocksFromInsts`(s_1, s_2):

```
    // Backup of maps
1   S'_1 = S_1, S'_2 = S_2
2   smap' = smap
3   varmap' = varmap
4   while s_1 is not terminator ∧ s_2 is not terminator do
        // Let (o^1_1, ..., o^1_{n_1}) and (o^2_1, ..., o^2_{n_2}) be list of operands
        // of s_1, s_2, respectively
5       for 1 ≤ i ≤ min(n_1, n_2) do
6           if o^1_i ∈ V_1 ∧ o^2_i ∈ V_2 ∧ o^1_i, o^2_i are not in varmap then
7               varmap(o^1_i) = o^2_i
8       if ¬cmpInst(s_1, s_2) then
            // Restore maps
9           S_1 = S'_1, S_2 = S'_2
10          smap = smap'
11          varmap = varmap'
12          return false
13      S_1 = S_1 ∪ {s_1}, S_2 = S_2 ∪ {s_2}
14      smap(s_1) = s_2
15      varmap(v_1) = v_2
16      s_1 = succ(s_1)
17      s_2 = succ(s_2)
18  return true
```

compared instructions have local variables on their input, which are the result of instructions that have not been compared yet. Therefore, those variables do not have mapping in *varmap*, which means that using the `cmpInst` function directly (listed in Algorithm 1) would compare such instructions as different. We do not want this to happen, because those unmapped variables can have the same semantics. Hence, for comparing the instructions in our method, we propose a special function that makes up the missing mapping. In addition, we will always apply our new comparison to all instructions of the basic blocks from some given (starting) instructions. Therefore, we propose the `cmpBasicBlocksFromInst` function which is described in Algorithm 3. It works as follows:

1. We back up the synchronisation maps and sets for a case when the comparison is not successful (lines 1-3).

2. We iterate through all instructions starting from the given s_1, s_2 until the end of basic blocks (line 4).

3. We check the operands of the current instructions. When the operands do not have a mapping, we create it (lines 5-7). The missing mapping can occur when the operands are results of instructions that are located in unanalysed basic blocks.

Algorithm 4: First subphase – finding the MCS on the level of basic blocks.

```
   Input  : queue Q
   Output: differing instructions DiffInst_1, DiffInst_2
   Function findMCSBasicBlocks(Q):
1      while Q is not empty do
2          take any (bb_1, bb_2) from Q
3          if cmpBasicBlocksFromInsts(first inst of bb_1, first inst of bb_2) then
4              t_1 = terminator of bb_1
5              t_2 = terminator of bb_2
6              if t_1 is conditional branch then
7                  insert non-analysed (succT(t_1), succT(t_2)) to Q
8                  insert non-analysed (succF(t_1), succF(t_2)) to Q
9              else if t_1 is not return then
10                 insert non-analysed (succ(t_1), succ(t_2) to Q
11         else
12             DiffInst_1 = first differing instruction in bb_1
13             DiffInst_2 = first differing instruction in bb_2
14     return (DiffInst_1, DiffInst_2)
```

4. At the beginning, we do not know whether there is a difference and where it can be. Therefore we compare instruction by instruction (line 8).

5. When the compared instructions are equal, we insert them into sets and create a synchronisation (lines 13-15).

6. Once the synchronisation is broken by finding differing instructions, we need to remove the synchronisation added within this function, as it may be incorrect. Hence, we restore maps and sets from the backup (lines 9-11), and the function returns *false*.

7. We set s_1 and s_2 to the successors of the current instructions (lines 16-17).

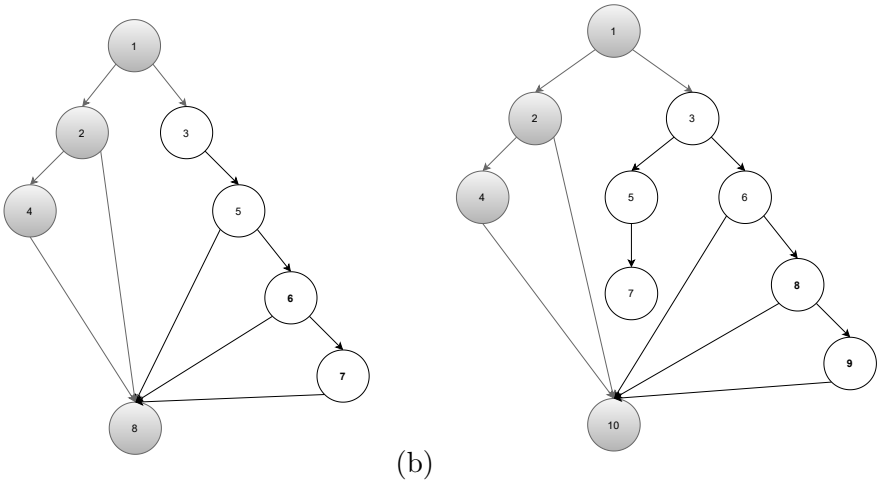
8. If all instructions of basic blocks were compared as equal, we declare the given parts of basic blocks equal, and we return *true*.

4.2.2 MCS on the Level of Basic Blocks

The search of MCS on the level of basic blocks is shown in Algorithm 4, where we work with the same queue Q as in the top-level Algorithm 2. As we synchronously go through the given functions by following the control flow, we work with pairs of basic blocks stored in Q. For the purpose of the following explanation, we introduce a new formalism BB_i to be the set of all basic blocks in function f_i. In each iteration of the algorithm, the following is done:

1. We take a pair of basic blocks bb_1 from f_1 and bb_2 from f_2 that are going to be compared (line 2).

2. We compare the taken basic blocks by using the function `cmpBasicBlocksFromInsts`. In principle, the function does the same as Algorithm 1, but it compares only one pair of basic blocks (b_1, b_2) from the given instructions s_1 and s_2. The function is described in Subsection 4.2.1.



(a) (b)

Figure 4.1: (a) the old version of function (b) the new version of function

3. If the compared basic blocks are semantically the same (synchronised), we insert their successors into Q (lines 4-10).

4. In case of unequal basic blocks, we store the found unequal instructions to $DiffInst_1$ from f_1 and $DiffInst_2$ from f_2. In this case, we take another pair from the queue for comparison without inserting the successors of these blocks.

Since we insert only successors of synchronised basic blocks, we ensure finding of a part of the maximum common subgraph on the level of basic blocks. For a better understanding, let us consider two control flow graphs in Figure 4.1. The entry blocks are basic blocks `1` and exit blocks are basic blocks with number `8` in the old version and `10` in the new version. In the previous analysis, DiffKemp found a difference in the first instructions of basic blocks marked as `3`. According to the proposed `findMCSBasicBlocks` function, we compared and synchronised basic blocks highlighted with grey, i.e., basic blocks marked with `1, 2, 4, 8` in the old version and `1, 2, 4, 10` in the new version. The remaining blocks are not analysed yet, because:

- *they contain differing instructions* (specifically basic blocks `3`) or
- *they are not successors of any synchronised basic block* (specifically basic blocks `5, 6, 7` in the old version and `5, 6, 7, 8, 9` in the new version).

4.2.3 MCS on the Level of Instructions

After the previous subphase, there remained unanalysed basic blocks, therefore we follow by finding MCS on the level of instructions inside them. We can found other semantically equal parts for removal in the unanalysed blocks. Therefore, we try to find the next synchronisation pair after the found differences. We propose a second subphase that tries to find the synchronisation pair in the unanalysed basic blocks. The finding of the next synchronisation pair is actually finding the MCS on the level of instructions. The algorithm of finding such instructions is shown in Algorithm 5. It works with two queues of instructions Q_1 for the function f_1 and Q_2 for the function f_2 that we use to traverse the functions on

Algorithm 5: Second subphase – the search of the synchronisation after difference.

Input : differing instructions $DiffInst_1$, $DiffInst_2$
Output: sets of instructions that must remain in slices K'_1, K'_2
Function `findMCSInsts(`$DiffInst_1, DiffInst_2$`):`

```
1   synchronisationFound = false
2   K'_1 = {}
3   Q_1 = {DiffInst_1}
4   while Q_1 is not empty do
5       take front s_1 from Q_1
        // Empty the set because previous s_1 instruction did not have
        // synchronisation
6       K'_2 = {}
7       Q_2 = {DiffInst_2}
8       while Q_2 is not empty do
9           take front s_2 from Q_2
10          if cmpBasicBlocksFromInsts(s_1, s_2) then
11              synchronisationFound = true
12              break
13          insert s_2 into K'_2
14          insert all non-analysed successors of s_2 to Q_2
15      if synchronisationFound then
16          break
17      insert s_1 into K'_1
18      insert all non-analysed successors of s_1 to Q_1
19  if synchronisationFound then
20      K'_1 = K'_1 ∪ Q_1
21      K'_2 = K'_2 ∪ Q_2
22  return (K'_1, K'_2)
```

the principle of *breadth first search*. These queues store possible synchronisation points, i.e., instructions that have no synchronisation yet. At the input, the algorithm gets the pair of differing instructions computed in the previous subphase, from which the search for synchronisation begins. At the end of the algorithm, we get sets containing instructions without synchronisation (i.e., different parts of the functions). The algorithm works as follows:

1. We initialise the sets K'_1 and K'_2 that will contain differing instruction from functions f_1, f_2 respectively (line 2 and line 6). We start finding the synchronisation from the differing instructions $DiffInst_1$ found in f_1 and $DiffInst_1$ found in f_2 by putting them into Q_1 and Q_2 (line 3 an line 7).

2. We *freeze* the instruction s_1 taken from Q_1 while we iterate through all instructions without synchronisation from f_2 until we find a synchronisation. In other words, we compare each unanalysed instruction from f_1 with each unanalysed instruction from f_2. During this analysis we use the breadth first search of control flow graphs.

3. In order to avoid getting a „false synchronisation“, (i.e., a situation when we would compare two seemingly equivalent instructions as equal, but the following parts would be different), we require that the synchronisation is found from the given instructions

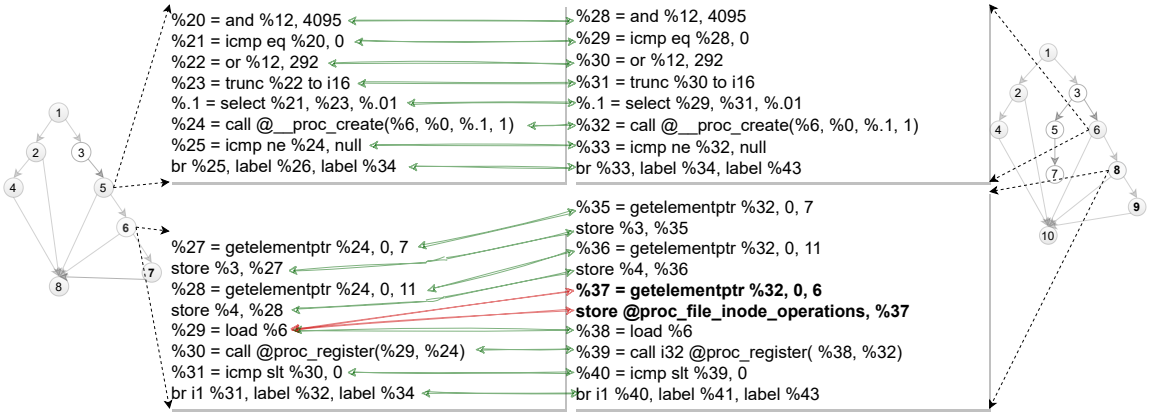


Figure 4.2: (a) old version of function (b) new version of function

for all subsequent instructions to the end of basic blocks. To do this, we use the function `cmpBasicBlocksFromInsts` which compares the basic blocks starting from s_1 and s_2 (line 10).

4. Sets K_1' and K_2' contain instructions for which no synchronisation was found. If we do not find synchronisation for an instruction, we will add it to the appropriate keep set. For instructions from f_1, we do this only when we have compared it with all (not yet analysed) instructions in f_2. In contrast, we go through the instructions from f_2 repeatedly (we go through all of them for each instruction from f_1). Therefore, every time we start the traversal from the beginning, we have to reset the keep set K_2'.

5. If we do not find synchronisation for an instruction, we put all its successors in to the appropriate queue, thus guaranteeing the traversal in the breadth first search order. We do not insert instructions that were analysed in the previous phases or instructions that are in the keep sets.

6. When the finding of synchronisation ended, the sets K_1' and K_2' contain the instructions that must be kept (such instructions have no synchronisation). When there is no synchronisation found, these sets contain each instruction from f_1 and f_2 that we did not analyse in the first subphase. Otherwise, when a synchronisation was found, they contain instructions from f_1 and f_2 starting from differing instructions until the new synchronisation point. In such a case, there may remain some unanalysed basic blocks (those after the found synchronisation), hence we repeat the entire MCS algorithm for these blocks.

7. We always add the basic blocks that remained in Q_1 and Q_2 to keep sets, because that are some instructions that were not analysed yet, but they are located after non-equal instructions. Since we cannot reach them, we keep them in slices. Note that we could further improve this step by trying to find another synchronisation between instructions left in Q_1 and Q_2. An example of this case is shown in the figure we illustrate.

For illustrating what happened in this subphase, see Figure 4.2 which display contents of some basic blocks from Figure 4.1. The differing instructions are the first instructions in

basic blocks `3` in both versions. Therefore, we freeze the first instruction in the old version, and we iterate through the entire basic block `3` in the new version without successfully finding the synchronisation. Hence, we analyse other remaining basic blocks, i.e., `5, 6, 7, 8, 9`. The synchronisation was not found in any of mentioned basic blocks. Then we get the successor of frozen instruction and do the full process of finding again. The synchronisation was not found for any instruction from block `3` in f_1. Therefore, we take the basic block `5` (the only successor of block `3`) in the old version, and we again try to find synchronisation with one of the non-synchronised basic blocks from the new version. This process continues until we get the first instruction (`%20`) from basic block `5` in the old version. For this instruction we find a synchronisation with the first instruction (`%28`) of the basic block `6` in the new version. The entire basic blocks `5` in f_1 and `6` in f_2 are equal, i.e the function `cmpBasicBlocksFromInsts` succeeds. Before ending this subphase, we need to check the sizes of queues. Since we traverse the functions by breadth first search, at the moment when the synchronisation is found, the block `7` from the new version is still in Q_2 (i.e., unanalysed). This means that this block is an extra one without any synchronisation, therefore it has to be kept in the slice.

Now, when the synchronisation is found, we have still unanalysed basic blocks `6, 7` in f_1 and `8, 9` in f_2, hence we repeat the entire MCS search, starting from the newly synchronised blocks `5` and `6`. The first subphase founds another difference right in the first blocks `6` and `8`. This difference is shown in Figure 4.2 with a red arrow. The new differing instructions are `%29` in the old version and `%37` in the new version. Therefore, we again move to the second subphase and try to find the next synchronisation.

In the second subphase, we freeze instruction `%29` in f_1 and try to find a synchronisation for it. This time, we succeed rather quickly and find a synchronisation starting from `%38` in the new version. We can see this next synchronisation in Figure 4.2. The second subphase ends with two unsynchronised instructions in K_2' while K_1' is empty in this case.

The remaining basic blocks, specifically `7` and `9`, are analysed in the first subphase. This subphase compares blocks as equal, and the analysis of functions ends. Hence, we analysed the entire functions and collected all the differing instructions in K_1 and K_2. At this moment, we can create the program slices. About the production of program slices in this thesis, see the next Section 4.3.

4.3 Equivalence Slicer

In the first phase, we analysed the entire functions and collected differing parts of functions that are semantically unequal. Now, we need to produce the final program slices. The differing instructions are stored in sets K_1 for f_1 and K_2 for f_2. To produce backward static slices, we also need to keep every instruction that can affect the differences and remove other instructions. Then, the slices are produced, and we have only semantically differing parts of non-equal functions. The algorithm of producing final program slices is shown in function **sliceFunctions** in Algorithm 6. At the input, it takes a function f that will be sliced and a set K that consists of instructions that should remain in the final slice. At the output, we get the sliced function. In this algorithm, the following is done:

1. We iterate through each instruction included in set K (lines 1-2).
2. We add each instruction k from K to the final slice (lines 3).

Algorithm 6: Second phase – producing the backward static slices.

```
   Input  : function f, set K representing the slicing criteria
   Output: sliced function f
   Function sliceFunction(f, K):
1      while K is not empty do
2          take any k ∈ K
3          add k to slice
4          for each operand o of k do
5              if o is the result of an instruction i then
6                  K = K ∪ {i}
7              if type(o) = pointer then
8                  K = K ∪ {all store instructions to o between definition of o and k}
9      return slice
```

3. Each instruction can have operands and we need to include instructions that may affect values of these operands to the final slice.

   (a) If these operands are local variables, i.e., results of instructions, we add the instructions to the slice (lines 4-6).

   (b) In the case that the type of the operand is a pointer, we search for each `store` instruction to the pointer between the instruction k and the definition of the pointer. We need to keep these instructions (lines 7-8) to satisfy the conditions of backward static slicing.

Chapter 5

Implementation

In the previous chapter, we proposed a method that removes as many semantically equivalent parts of two functions as possible. As we explained, it creates two program slices that can be further processed, e.g., analysed by some heavy-weight formal method that can give a more precise result of the semantic comparison. Before we describe the implementation details of the equivalence slicer, we give an outline how DiffKemp is implemented. We described how DiffKemp checks semantic equivalence in Chapter 2.

This chapter is organised as follows. In Section 5.1 we describe DiffKemp in terms of implementation where we present its two phases that are important for checking function equivalence. In Section 5.2, we describe an integration of the designed method into the DiffKemp tool.

5.1 Architecture of DiffKemp

Comparison of different versions of programs in DiffKemp is split to two phases, called **generate** and **compare** [14]. As we already described, DiffKemp analyses given programs, written in C language, in LLVM IR. The generate phase is used for compilation of C programs to this low-level representation. This phase is described in Subsection 5.1.1. After the LLVM representation is produced, the equivalence of programs is checked in the compare phase. This phase is described in Subsection 5.1.2.

5.1.1 Generate Phase

In the generate phase, the source files containing the definitions of the compared functions are compiled into the LLVM IR language, often used for program analysis. More information about LLVM IR can be found in Section 2.1. The output of the generate phase is a **snapshot**, on which the subsequent semantics comparison is based.

A snapshot is an abstraction of one version of the compared project compiled into LLVM IR. It is represented as a folder containing the appropriate LLVM IR files and a configuration file with a list of functions that will be compared. For each function from a list of functions or paramters given at the input of the generate phase, this configuration file specifies the LLVM IR file containing its definition. DiffKemp is primarily used on the Linux kernel and the list of functions is usually taken from a so-called KABI[1] list, which is a part of the Red Hat Enterprise Linux (RHEL) kernel source files and it is located in the

[1] **K**ernel **A**pplication **B**inary **I**nterface

`kabi_whitelist_x86_64` file. It is also possible to use a custom list, where each function name for comparison will be located on a separate line. The course of the generate phase is shown in Figure 5.1. It consist of two subphases:

1. The first subphase is called a **source finder**. DiffKemp uses the cscope utility[2] for finding the source definitions of the given functions or functions that use given kernel parameters.

2. The second subphase is **compilation**. DiffKemp uses the LLVM/Clang compiler for getting the internal representation (LLVM IR) from the C programs.

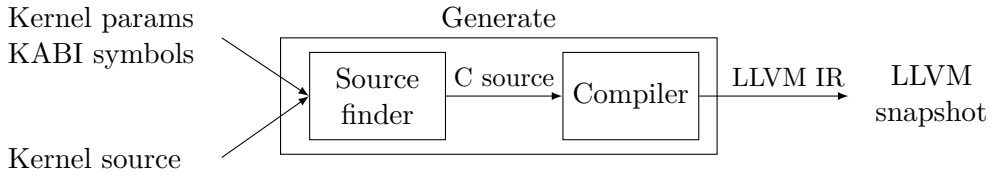


Figure 5.1: The principle of the generate phase.

Example of running the generate phase:

```
bin/diffkemp generate kernel/linux-3.10.0-862.el7 snapshots/linux-
3.10.0-862.el7 kernel/linux-3.10.0-862.el7/kabi_whitelist_x86_64
```

In this example, a snapshot of the source files of the Linux kernel version `3.10.0-862.el7` will be created and located in `snapshots/linux-3.10.0-862.el7/` folder. The source files and the `kabi_whitelist_x86_64` are located in `kernel/linux-3.10.0-862.el7/` folder.

5.1.2 Compare Phase

The second phase of the DiffKemp tool is the compare phase, in which semantics of the previously created snapshots is compared. More information about function comparison was given in Section 2.2. It takes two snapshots and its output is a decision for each function found in both compared snapshots, whether the semantics of the function is the same or different. A majority of the compare phase is implemented in a component of DiffKemp called SimpLL, whose architecture is shown in Figure 5.2. SimpLL works in three subphases which correspond to the following course:

1. The first subphase is **code slicing and simplifying** where various function transformations take place. Their task is to simplify the code for easier comparison. These transformations are implemented using so-called *LLVM passes*.

2. The second subphase is **semantic diff**. After the simplification subphase, there is a communication between the classes `ModuleComparator` and `DifferentialFunctionComparator`, whose task is to compare the required functions.

3. The third subphase is **difference localisation**. This subphase is done only when the previous subphase compares functions as non-equal. The found differences are represented using the YAML language.

[2] http://cscope.sourceforge.net

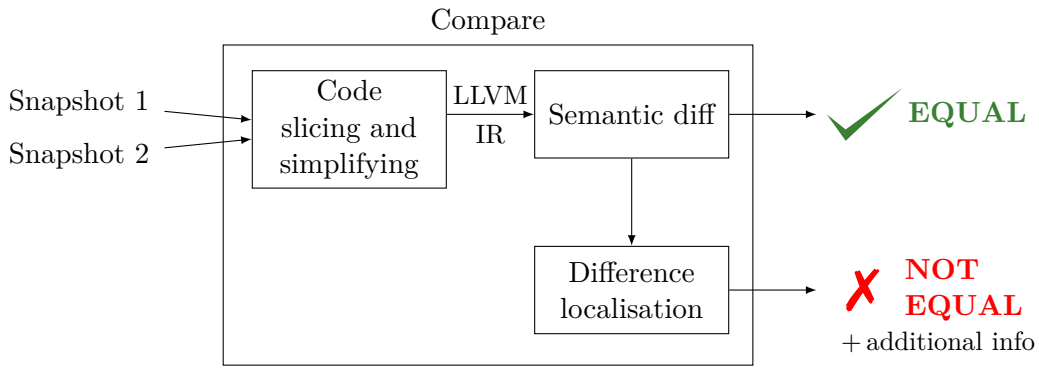


Figure 5.2: The principle of the compare phase.

Example of running the compare phase:

```
bin/diffkemp compare snapshots/linux-3.10.0-862.el7 snapshots/linux-3.10.0-957.el7 -f bio_add_page --show-diff
```

In this example, the `bio_add_page` function of the Linux kernels of versions `3.10.0-862.el7` and `3.10.0-957.el7` will be compared. The possible difference will be saved to a newly created folder `diff-linux-3.10.0-862.el7-linux-3.10.0-957.el7/` in `bio_add_page.diff` file.

5.2 Integration of Proposed Slicer

We implemented the method proposed in Chapter 4 in the DiffKemp tool. It is implemented in a new class named `EquivalenceSlicer` in C++ language as a post-processing step of the compare phase. This class contains a method `slice` that removes each equivalent part of compared functions based on the previous analysis. DiffKemp already contains one program slicer, called `VarDependencySlicer` which implements forward slicing. Parts of our implementation use parts of this slicer, in particular our second phase is implemented using this slicer.

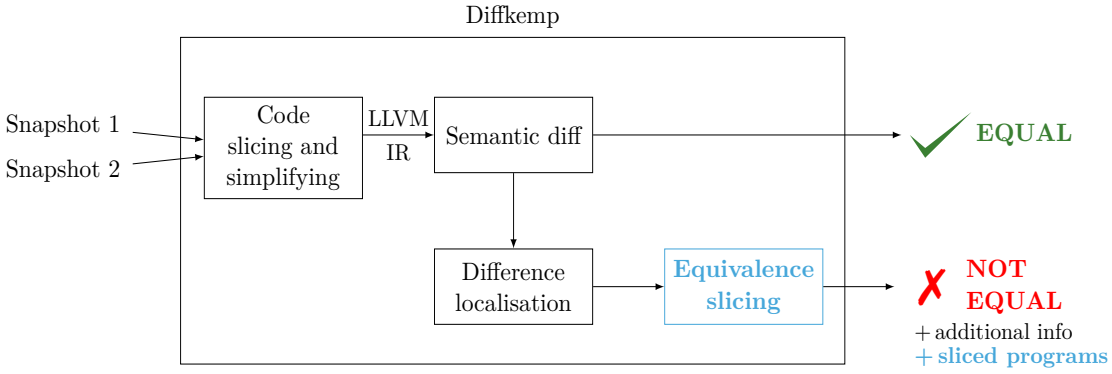


Figure 5.3: Integration of equivalence slicer.

Equivalence slicer can be enabled by a new command-line option `--equivalence-slicer`. The slicing is done by enabling the equivalence slicer (only in case of non-equal functions)

in the `ModuleComparator` class. This class contains a method `compareFunctions` where is also slicing done. By using the command line option `--output-llvm-ir`, we can get an LLVM IR file for both compared versions of the program. These produced files contain entire modules, where the compared functions are already sliced. The integration of the equivalence slicer into compare phase is shown in Figure 5.3 highlighted by blue colour.

Now, every pair of functions compared as non-equal can be sliced. These slices can be given to some heavy-weight formal tool that could be able to check the semantic difference and give us a more precise result than DiffKemp.

Chapter 6

Experimental Evaluation

We have proposed and implemented a method that removes equivalent parts of the compared functions in DiffKemp. In this chapter, we perform several experiments to evaluate our approach. The validation and experiments are done with disabled inlining, because our extension does not support the inlining pattern. Therefore, the results provided in this section and in the paper [15] may be different. The experiments were run on a 4 core, 2.6 GHz Intel Xeon Ivy Bridge machine with 8GB RAM in LLVM 11 version.

This chapter is organised as follows. In Section 6.1, we show that implementation of this method does not change the results of checking the semantic equivalence and we provide other validations of our solution. In Section 6.2, we evaluate benefits of the implemented method.

6.1 Cross-validation

In this section, we show that our extension is valid and does not change the decision made by DiffKemp. To show it, we made a cross-validation in five different RHEL versions. It is shown in Table 6.1. The cross-validation is done by slicing some functions and then comparing the sliced functions using DiffKemp. We show results of DiffKemp without the extension where original functions were compared. Then, we show results of DiffKemp with the extension, where we run `EquivalenceSlicer` on the original functions, which produces slices. Subsequently, we run the compare phase again on the sliced functions. The results show that the implemented slicing technique preserves the semantically non-equivalent parts, that DiffKemp again identifies. Moreover, the cross-validation shows that slices are valid programs, i.e., all instructions have their dependencies in the slice. Otherwise, the comparison would end with errors what would be reflected in a different number of compared functions according to the results without the extension in Table 6.1. The only difference is in RHEL versions 7.4-7.5 in non-equal results and it is caused by a new error. After investigation, we found out that this error is caused by a bug in DiffKemp which did not show up before. It did not show up, because the analysis ended when it found the first difference in functions and it did not get to the point where our extension did.

In addition, we show that our extension does not break any of the existing code by running the existing set of unit and regression tests that successfully passed. We also implemented a new set containing twenty-five regression tests in six RHEL versions. These tests serve to check the functionality of the `EquivalenceSlicer` class. The functionality is

RHEL versions	Results: equal/non-equal/unknown	
	w/o extension	with extension
7.3-7.4	403/269/6	403/269/6
7.4-7.5	544/184/6	544/183/6
7.5-7.6	605/128/6	605/128/6
8.0-8.1	361/85/25	361/85/25
8.1-8.2	330/165/26	330/165/26

Table 6.1: Cross-validation of equivalence slicer.

checked by comparison of the program slices from the output of `EquivalenceSlicer` and prepared model slices. The correctness of model slices was manually checked.

6.2 Evaluation of the Slicing Extension

Now, when the validity of our method is shown, we can discuss the benefits of the implemented solution. In this section, we provide two experiments shown in Table 6.2. In the first experiment, we show how long the comparison takes with and without the extension. In average, DiffKemp compares one pair of kernels without the extension in `07:11` minutes. With our extension, DiffKemp compares one pair of kernels in `7:25` minutes in average. Therefore, we can declare that our extension takes only **14 more seconds** of comparison in average. This shows that our method is rather efficient in practice.

In the second experiment, we show how many instruction of non-equal functions our extension removed, whereas the semantics of different parts of functions is preserved. At the end of the comparison in DiffKemp without the extension, we have 16 786 instructions in non-equivalent functions. With the extension, we get only 7 588 instructions which is **54.8% less** than without the extension.

RHEL versions	Runtime: mm:ss		Number of instructions	
	w/o extension	with extension	w/o extension	with extension
7.3-7.4	07:08	07:31	5 182	2 358
7.4-7.5	09:31	09:40	3 544	1 178
7.5-7.6	08:03	08:23	3 272	1 501
8.0-8.1	05:08	05:20	2 119	1 308
8.1-8.2	06:08	06:15	2 669	1 243

Table 6.2: Runtime and number of instructions with and without the extension.

Chapter 7

Conclusion

In this thesis, we proposed a method that is able to remove as much semantically equal parts of two given non-equal functions as possible. Before simplification, we need to detect these semantically equivalent parts of programs that should not stay in the result functions. Since we work with LLVM IR language, where functions are represented as control flow graphs, we are using a graph algorithm to find such semantically equivalent parts. Specifically, we are finding the solution to the maximum common subgraph problem. When we find the maximum common subgraph, we use static backward slicing technique to keep only differences (our slicing criteria) and instructions that can affect them. After this, we get two program slices that consist of non-equivalent parts of programs and their dependencies.

The proposed method was successfully implemented in the DiffKemp tool as a post-processing step. DiffKemp is a fast static analyser. The main goal of DiffKemp tool is a high scalability to real source code. It provides information about syntactic differences of two given functions while it ignores changes that do not affect the semantics. Since the check of semantic equality is a difficult problem, we can use some heavy-weight tool to get more precise result of semantic equality. These tools are not able to process large programs. With a help of our solution we remove semantically equivalent parts of programs when Diffkemp compares two functions as non-equal. Then, these simplified functions can be further given to some heavy-weight tool for more precise semantic comparison.

The contribution of our method is shown in several experiments. They show that our method is able to reduce the size of functions by 54.8% in only 3.2% more time of comparison while it preserves the semantics of nonequivalent parts of two given function. The validity of implemented method is also shown in several ways. For further improvements, we have prepared multiple regression tests that check the correct functionality of equivalence slicing.

In future, we would like to fix the known limitation and prepare some heavy-weight tool for immediate semantic comparison of sliced non-equal functions.

Bibliography

[1] Abu-Khzam, F. N., Samatova, N. F., Rizk, M. A. and Langston, M. A. The Maximum Common Subgraph Problem: Faster Solutions via Vertex Cover. In: *2007 IEEE/ACS International Conference on Computer Systems and Applications*. 2007, p. 367–373. DOI: 10.1109/AICCSA.2007.370907.

[2] Adve, V. and Lattner, C. *LLVM Language Reference Manual*. 2021. Available at: https://llvm.org/docs/LangRef.html.

[3] Binkley, D. and Harman, M. A large-scale empirical study of forward and backward static slice size and context sensitivity. In: *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* 2003, p. 44–53. DOI: 10.1109/ICSM.2003.1235405.

[4] Binkley, D. W. and Gallagher, K. B. Program Slicing. In: Zelkowitz, M. V., ed. Elsevier, 1996, vol. 43, p. 1–50. Advances in Computers. DOI: https://doi.org/10.1016/S0065-2458(08)60641-5. ISSN 0065-2458. Available at: https://www.sciencedirect.com/science/article/pii/S0065245808606415.

[5] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N. and Zadeck, F. K. An Efficient Method of Computing Static Single Assignment Form. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1989, p. 25–35. POPL '89. DOI: 10.1145/75277.75280. ISBN 0897912942. Available at: https://doi.org/10.1145/75277.75280.

[6] De Lucia, A. Program slicing: methods and applications. In: *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*. 2001, p. 142–149. DOI: 10.1109/SCAM.2001.972675.

[7] Duesbury, E., Holliday, J. and Willett, P. Maximum Common Subgraph Isomorphism Algorithms. *MATCH Communications in Mathematical and in Computer Chemistry*. April 2017, vol. 77, no. 2, p. 213–232. © 2016 MATCH. This is an author produced version of a paper subsequently published in MATCH Commun. Math. Comput. Chem. Uploaded with permission from the copyright holder. Available at: http://eprints.whiterose.ac.uk/102232/.

[8] Harman, M., Danicic, S., Sivagurunathan, Y. and Simpson, D. *The Next 700 Slicing Criteria*. 1996.

[9] Harman, M. and Hierons, R. M. An overview of program slicing. *Softw. Focus*. 2001, vol. 2, no. 3, p. 85–92. DOI: 10.1002/swf.41. Available at: https://doi.org/10.1002/swf.41.

[10] Horwitz, S., Reps, T. and Binkley, D. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. Program. Lang. Syst.* New York, NY, USA: Association for Computing Machinery. january 1990, vol. 12, no. 1, p. 26–60. DOI: 10.1145/77606.77608. ISSN 0164-0925. Available at: https://doi.org/10.1145/77606.77608.

[11] Kann, V. On the approximability of the maximum common subgraph problem. In: Finkel, A. and Jantzen, M., ed. *STACS 92*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, p. 375–388. ISBN 978-3-540-46775-5.

[12] Kerrisk, M. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1stth ed. USA: No Starch Press, 2010. ISBN 1593272200.

[13] Kiefer, M., Klebanov, V. and Ulbrich, M. Relational Program Reasoning Using Compiler IR. In: Blazy, S. and Chechik, M., ed. *8th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2016), Revised Selected Papers*. Springer, November 2016, vol. 9971, p. 149–165. Lecture Notes in Computer Science. DOI: 10.1007/978-3-319-48869-1_12.

[14] Malík, V. *Diffkemp* [online]. Github, april 2021 [cit. 2021-01-08]. Available at: https://github.com/viktormalik/diffkemp.

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

[16] MINOT, M. and Ndiaye, S. N. Searching for a maximum common induced subgraph by decomposing the compatibility graph. In: *Bridging the Gap Between Theory and Practice in Constraint Solvers, CP2014-Workshop*. Lyon, France: [b.n.], September 2014, p. 1–17. Available at: https://hal.archives-ouvertes.fr/hal-01301095.

[17] Sarda, S. and Pandey, M. *LLVM Essentials*. Packt Publishing, 2015. ISBN 1785280805.

[18] Sasirekha, N., Robert, A. E. and Hemalatha, D. M. *Program slicing techniques and its applications*. 2011.

[19] Vojnar, T. *Static Analysis and Verification* [online]. 2020 [cit. 2021-04-11]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/Lectures/sav-lecture-01.pdf.

[20] Weiser, M. Program Slicing. In:. IEEE Press, 1981, p. 439–449. ICSE '81. ISBN 0897911466.

[21] Weisstein, E. W. *NP-Problem* [online]. MathWorld–A Wolfram Web Resource [cit. 2021-04-18]. Available at: https://mathworld.wolfram.com/NP-Problem.html.

Appendix A

Content of CD

The enclosed CD contains following files and directories:

```
/
├── bin/ ..... DiffKemp bin
├── diffkemp/ ..... DiffKemp source files
├── docker/ ..... Setup for docker
├── rpm/ ..... Build for Fedora
├── tools/ ..... DiffKemp useful scripts
├── tests/ ..... Unit and regression tests
├── tex/ ..... LaTeX source files of this thesis
├── README.md ..... README file
├── requirements.txt ..... DiffKemp requirements
├── setup.py ..... DiffKemp setup
├── build.sh ..... DiffKemp build script
└── count_instructions.py ..... Script for counting instructions
```

Implemented `EquivalenceSlicer` class can be found in folder `/diffkemp/simpll/`, specifically in `EquivalenceSlicer.cpp` and `EquivalenceSlicer.h`. The model slices used in regression tests are located in `tests/regression/sliced_functions/`. The implementation of regression tests is in `tests/regression/slicer_test.py`.

Appendix B

How to Build and Test

The project can be built and run by using development container image[1]. After retrieving the container, it can be run by using:

```
docker/diffkemp-devel/run-container.sh
```

Once it is loaded, we can build the DiffKemp, download necessary Kernel souce codes and create snapshots by using attached script:

```
sh build.sh
```

B.1 Regression Tests

To run all regression tests, we use:

```
pytest tests
```

Or we can run only regression tests created for the `EquivalenceSlicer` by using:

```
pytest tests/regression/slicer_test.py
```

B.2 Reproduction of Experiments

To reproduce experiments, we need to compare pairs of RHEL versions. E.g., for RHEL versions 7.3-7.4 without the extension we run command:

```
bin/diffkemp -v compare snapshots/linux-3.10.0-514.el7/ snapshots/
linux-3.10.0-693.el7/ --stdout --show-diff --inlining-off
--report-stat > 514-693_without.log 2>&1
```

[1] It is available at: https://hub.docker.com/r/viktormalik/diffkemp-devel/

For the same version with the extension we run:

```
bin/diffkemp -v compare snapshots/linux-3.10.0-514.el7/ snapshots/
linux-3.10.0-693.el7/ --stdout --show-diff --equivalence-slicer
--report-stat > 514-693_with.log 2>&1
```

After this, we have compared functions in given versions. The results of these comparisons are stored in `514-693_without.log` and `514-693_with.log` files. At the end of these files, we can find in statistics the durations of the comparisons. To get the number of instructions, we use attached script with the results of comparisons:

```
python3 count-instructions.py 514-693_without.log
python3 count-instructions.py 514-693_with.log
```

B.3 How to Get Sliced Functions

To see how compared functions were sliced, we use command line options:

1. `--equivalence-slicer` for enabling the implemented `EquivalenceSlicer` and
2. `--output-llvm-ir` for creating the modules that contain sliced functions.

E.g., by using following command:

```
bin/diffkemp -v compare snapshots/linux-3.10.0-514.el7/ snapshots/
linux-3.10.0-693.el7/ --stdout --show-diff --equivalence-slicer
--report-stat --output-llvm-ir -f __ethtool_get_settings
```

We can find the modules containing sliced functions in files:

- `snapshots/linux-3.10.0-514.el7/net/core/ethtool-simpl.ll`
- `snapshots/linux-3.10.0-693.el7/net/core/ethtool-simpl.ll`