BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

APPLYING FORMAL METHODS TO ANALYSIS OF SEMANTIC DIFFERENCES BETWEEN VERSIONS OF SOFTWARE

APLIKACE FORMÁLNÍCH METOD V ANALÝZE SÉMANTICKÝCH ROZDÍLŮ MEZI VERZEMI SOFTWARE

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. FRANTIŠEK NEČAS
SUPERVISOR VEDOUCÍ PRÁCE	Ing. VIKTOR MALÍK, Ph.D.

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

157112

Institut:	Department of Intelligent Systems (DITS)
Student:	**Nečas František, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Mathematical Methods
Title:	**Applying formal methods to analysis of semantic differences between versions of software**
Category:	Software analysis and testing
Academic year:	2023/24

Assignment:

1. Get acquainted with DiffKemp, a tool for static analysis of semantic differences between versions of large-scale C projects.
2. Identify patterns of semantic-preserving changes (so-called refactorings) which DiffKemp is not yet able to handle.
3. Propose a way to augment the analysis approach of DiffKemp by applying formal methods such it will be able to handle a subset of the identified refactoring patterns.
4. Implement the proposed solution within DiffKemp.
5. Evaluate the created solution on existing benchmarks for analysis of semantic equivalence or on real-world projects that DiffKemp targets (Linux kernel, various system libraries). Demonstrate that your solution helps DiffKemp to correctly compare more programs than before.

Literature:

- Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 329–339. IEEE (2021)
- S. Badihi, Y. Li and J. Rubin: EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In: 2021 18th IEEE/ACM International Conference on Mining Software Repositories (MSR). pp. 610-614. IEEE (2021)
- Leonardo de Moura and Nikolaj Bjørner: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS. pp. 337-340. Springer (2008)

Requirements for the semestral defence:
The first two points 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:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	6.11.2023

Abstract

The goal of this work is to propose an integration of formal methods into DiffKemp, a static analysis tool for analyzing semantic differences of large-scale C projects. The aim of this extension is to facilitate analysis of more complex code changes, which would typically be better handled by a tool based on formal methods, while also maintaining DiffKemp's scalability to large projects. To achieve this, whenever a possible semantic change is found, the equivalence of the relevant instructions is encoded into an SMT problem instance and the difference is either confirmed or refuted using an SMT solver. The proposed solution has been implemented in DiffKemp and our experiments on a set of benchmarks called EqBench show that it extends the capabilities of DiffKemp, mainly with regards to sound analysis of refactorings of arithmetic expressions.

Abstrakt

Cílem této práce je navrhnout integraci formálních metod pro DiffKemp, nástroj pro statickou analýzu sémantických rozdílů v rozsáhlých programech napsaných v jazyce C. Cílem tohoto rozšíření je umožnit analýzu složitějších změn, které by typicky byly analyzovatelné spíše nástroji založenými na formálních metodách, a zároveň zachovat škálovatelnost nástroje DiffKemp na velké projekty. Principem navrženého řešení je při analýze v případě nalezení možné sémantické změny zakódovat problém ekvivalence příslušných instrukcí jako instanci problému SMT. Tím je možné sémantický rozdíl potvrdit, nebo vyvrátit s pomocí SMT solveru. Navržené řešení bylo implementováno v nástroji DiffKemp a experimenty provedené na sadě programů zvané EqBench ukazují, že rozšiřuje schopnosti nástroje DiffKemp, převážně v oblasti přesné analýzy úprav aritmetických výrazů.

Keywords

DiffKemp, static analysis, semantic differences, LLVM IR, formal methods, code change pattern, SMT solving, Z3 solver, Linux kernel, EqBench benchmark

Klíčová slova

DiffKemp, statická analýza, sémantické rozdíly, LLVM IR, formální metody, vzory změn v kódu, řešení problému SMT, Z3 solver, Linuxové jádro, sbírka programů EqBench

Reference

NEČAS, František. *Applying formal methods to analysis of semantic differences between versions of software.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík, Ph.D.

Rozšířený abstrakt

V rozsáhlém světě vývoje software existují projekty (např. systémové knihovny), u kterých je klíčové zachovat sémantickou stabilitu mezi verzemi, protože i malá změna by mohla mít zásadní dopad na uživatele těchto projektů. Za účelem zjednodušení a automatizace identifikace možných změn sémantiky jsou vyvíjeny *statické analyzátory sémantických rozdílů*. Většina současných analyzátorů je založena na formálních metodách (např. LLRêve), které jsou sice velmi přesné, ale špatně škálují na rozsáhlejší programy. Na druhé straně spektra jsou nástroje založené na porovnání textu nebo syntaxe (např. nástroj `diff`), které jsou velmi rychlé, produkují však velké množství falešných hlášení.

Jeden z nástrojů, DiffKemp, se snaží najít kompromis mezi těmito dvěma přístupy. Tento nástroj byl vyvinut společností Red Hat s cílem analyzovat sémantickou stabilitu některých částí linuxového jádra, rychlost analýzy je tedy naprosto klíčová. K dosažení efektivní analýzy s rozumnou přesností DiffKemp využívá několik základních konceptů. Analyzované programy jsou přeloženy do vnitřní reprezentace LLVM (LLVM IR), která je následně porovnávána převážně po instrukcích. Porovnávání po instrukcích je velmi rychlé, avšak i malá změna v kódu by mohla způsobit falešné hlášení. Z tohoto důvodu DiffKemp kód předzpracovává (např. odstraňuje mrtvý kód), aby porovnávané verze byly syntakticky podobnější a bylo tedy možné více instrukcí porovnat přímo. Pro případ, že se instrukce neshodují, má DiffKemp v sobě zabudované tzv. *vzory změn zachovávající sémantiku*, které se pokouší aplikovat.

Přestože má v sobě DiffKemp zabudovaných již několik vzorů a další mohou uživatelé dodat manuálně, existují stále případy, kdy DiffKemp chybně zahlásí sémantickou nerovnost verzí, přestože se jedná o ekvivalentní kód. To je způsobeno tím, že některé změny jsou příliš složité a byly by analyzovatelné spíše nástroji založenými na formalních metodách. Příkladem takových změn jsou úpravy aritmeticko-logických výrazů, např. aplikace distributivních zákonů nebo jiných algebraických pravidel. Existuje mnoho takových případů, a tak není možné je všechny implementovat manuálně jako vzory.

Tato práce navrhuje rozšíření nástroje DiffKemp, které automaticky umožní analyzovat takové změny jako sémanticky ekvivalentní. Principem navrženého řešení je při analýze v případě nalezení možné sémantické změny, na kterou není možné aplikovat žádný vzor, zakódovat problém ekvivalence příslušných instrukcí jako instanci problému SMT. Tím je možné sémantický rozdíl potvrdit, nebo vyvrátit s pomocí SMT solveru. Z důvodu efektivního začlenění do analyzačního algoritmu nástroje DiffKemp se tato práce zaměřuje pouze na analýzu ekvivalence sekvenčních bloků kódu, které neprovádějí paměťové operace.

Navržené řešení bylo implementováno v jazyce C++ v nástroji DiffKemp s využitím SMT solveru Z3, který byl vyhodnocen na základě různých kritérií jako nejvhodnější pro tento problém. Vytvořené rozšíření bylo experimentálně ověřeno na úlohách různé velikosti, od jednoduchých ručně vytvořených programů, přes různé systémové knihovny, až po linuxové jádro. Experimenty se sadou programů zvanou EqBench ukazují, že výsledné řešení rozšiřuje schopnosti nástroje DiffKemp, převážně v oblasti přesné analýzy úprav aritmetických výrazů. Naopak experimenty s rozsáhlejšími projekty (např. linuxovým jádrem a systémovými knihovnami) ukazují, že přestože je řešení problému SMT NP-těžký problém, navržené řešení se soustředí pouze na slibné bloky kódu, a tedy nedochází k zásadnímu zpomalení nástroje DiffKemp. Nástroj tedy stále zůstává kompromisem mezi rychlostí a přesností, s navrženým rozšířením se ovšem o něco více přesností přibližuje nástrojům založeným na formálních metodách. Zůstává však výhoda dobré škálovatelnosti.

Applying formal methods to analysis of semantic differences between versions of software

Declaration

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

. . . .
František Nečas
May 13, 2024

Acknowledgements

I would like to thank my supervisor Ing. Viktor Malík, Ph.D. for his support and patience during the work on this thesis and other projects. I am grateful that I could pursue research in the field of static analysis under his guidance. I would also like to thank my loved ones who have always been there for me over the course of my studies.

Contents

Chapter 1

Introduction

In the vast world of software development, there are some projects where maintaining semantic stability between versions is crucial, because even a tiny change could negatively impact the users of the said projects. An example of such projects are various system libraries, e.g., implementations of the standard C library, where the semantics of the exported functions should remain unchanged between versions so that other programs can safely use these functions. In order to simplify and automate identifying potential changes in semantics, various static analyzers of semantic differences have been developed.

There are several tools focusing on finding semantic differences, their approaches vary significantly. On one hand, there are tools based on formal methods which can verify semantic equivalence with high precision but do not scale well (e.g., LLRêve). On the other side of the spectrum, there are light-weight static analyzers that are able to analyze large chunks of code very fast but produce numerous false warnings or errors (on the extreme end, the diff Unix utility could be considered an example of this approach). One of the tools, DiffKemp, tries to find a middle ground between these two extremes. DiffKemp is developed by Red Hat; its original goal was finding semantic differences between functions of the Linux kernel that are part of a so-called Kernel Application Binary Interface[1]. Therefore, scalability of the tool to large projects is essential.

To achieve high scalability without sacrificing precision, DiffKemp utilizes a few different techniques. First, the versions of the C program to be analyzed are translated into LLVM Intermediate Representation (IR). Second, the versions are pre-processed to bring the code to a form that can be compared more reliably. Then, the transformed programs are compared per-instruction in the LLVM IR. The idea behind this approach is that typically, refactoring is done on a small part of the code, while the rest stays intact, and therefore can be efficiently compared using instruction-wise comparison. If instructions differ, DiffKemp tries to check applicability of one of the patterns (either implemented in the tool itself or possibly user-defined) that are known to preserve semantics.

While there are numerous patterns already implemented in DiffKemp, there are still some cases where the tool reports inequality despite the versions being equal. This is due to the fact that some changes are too complex and would be better suited for analysis based on formal methods. A common example of this are changes to arithmetic and logic expressions, e.g., using distributive laws and other algebraic properties. There are a lot of such possible refactorings, therefore it is not feasible to implement them all manually

[1] A list of functions whose semantics should remain unchanged throughout the whole major release of the Red Hat Enterprise Linux distribution.

in the tool itself. To overcome this, we propose an integration of formal methods into the DiffKemp's analysis algorithm. Whenever a possibly differing instructions are found and no pattern is applicable, the equivalence of the relevant instructions is encoded into an SMT problem instance and the difference is either confirmed or refuted using an SMT solver.

The rest of this thesis is organized as follows. Chapter 2 introduces DiffKemp and explains its most important techniques in more detail. Chapter 3 is devoted to SMT solving, which lies in the center of the proposed solution, and discusses strengths and weaknesses of state-of-the-art SMT solvers. Design of integration of formal methods into DiffKemp's analysis algorithm is described in Chapter 4. Chapter 5 gives an overview of DiffKemp's architecture and goes over important implementation details. Chapter 6 details the experiments that were carried out in order to validate the effectiveness of our work and the results of those experiments. Finally, Chapter 7 gives a description of various existing approaches to static analysis of semantic equivalence and discusses DiffKemp's capabilities with our extension compared to other tools.

Chapter 2

DiffKemp

DiffKemp is a static analysis tool for finding semantic differences in large-scale C projects developed by Red Hat. As discussed in Chapter 1, it tries to find a middle ground between analyzers based on formal methods and light-weight static analysis tools. To achieve high scalability while producing a low number of false alarms, DiffKemp makes use of the following important concepts:

- The versions are compared per-instruction in the LLVM Intermediate Representation (IR). This is a very fast approach, however even a very subtle change could lead to a false alarm.
- To prevent some of these false alarms, DiffKemp tries to bring the two versions of a program closer together by using various code transformations, e.g., some optimization passes may be run.
- If instruction-wise comparison fails, DiffKemp checks if any pattern from its list of predefined code change patterns that are known to preserve semantics is applicable.

In the rest of this chapter, we introduce the main concepts behind DiffKemp. Section 2.1 introduces LLVM IR and the structure of programs in this representation. Section 2.2 introduces the notion of function equivalence and gives the algorithm that DiffKemp uses for finding semantic differences. Finally, Section 2.3 gives an overview of the semantics-preserving change patterns that are available in the tool.

2.1 Intermediate Representation of Programs

DiffKemp is built on top of the LLVM infrastructure (that is extensively used in compilers) and therefore uses its intermediate representation. Each function in LLVM IR can be viewed as a control flow graph consisting of basic blocks connected by edges representing branching in the program [18]. Each basic block is a sequence of instructions ending with exactly one terminator instruction (one of the available branch instructions or return at the end of the function). Furthermore, only the first instruction of the basic block can be a target of jumps. The inner instructions of the basic block have exactly one successor – the following instruction. The terminator instructions contain references to the successor basic blocks.

An instruction in LLVM IR performs an operation over some operands and stores its result into a local variable. Local variables in LLVM IR are so-called *virtual registers* and do not correspond to the local variables of the original program (these variables are allocated using the `alloca` instruction and then accessed using the `store` and the `load` instructions). The instruction set is similar to that of RISC computers, i.e., quite a limited set of instructions including arithmetic operations, function calls, branching, and instructions for memory manipulation. An operand can either be a local or a global variable, a constant, or a function (e.g., for the `call` instruction). Each function (represented by the CFG) satisfies the single static assignment (SSA) property, i.e., each variable is assigned to at most once. To achieve this, each instruction stores its result into a fresh variable. The instructions can be divided into several types based on various criteria [30]:

Terminator Instructions As mentioned above, every basic block ends with this type of instruction. They define the control flow of the program, i.e., they do not produce any values. The `ret` and `br` instructions are the most commonly used instructions of this type.

Unary Operations These instructions require a single operand, execute an operation on it and produce a result, e.g., the `fneg` instruction performs negation of a floating-point value.

Binary Operations Most of the computation in a program is carried out by binary operations. This type includes all kinds of arithmetic operations, e.g., `add` for addition or `mul` for multiplication of integer types. An important property of these instructions is that they return a value of the same type as its operands.

Bitwise Binary Operations A special kind of binary operations that perform bit manipulations on its operands, e.g., bitwise shifts (`shl` for left shift, `lshr` for logical right shift or `ashr` for arithmetic right shift).

Vector Operations Represent vector operations in an architecture-independent manner. These instructions cover, for example, element access using the `extractelement` instruction.

Aggregate Operations Instructions for manipulating derived types that contain multiple members, e.g., arrays and structs.

Memory Access Operations The most commonly used instructions from this category are the `aloca`, `load` and `store` instructions for memory manipulation.

Conversion Operators These instructions take a single operand and a type and perform various bit conversions on the operand in order to get the requested type. An example of this is the `zext` instruction, which performs a zero extension of its operand.

Other Instructions Miscellaneous instructions that do not belong to other categories. The most relevant for this work are the `icmp` and `fcmp` instructions for comparison, `phi` used to implement the Φ-node in the SSA (i.e., a merge of values from multiple branches), `select` used as a ternary operator, and `call` used for calling functions.

Figure 2.1 shows an example conversion of a simple C program to LLVM IR (its human-readable serialized form, slightly abbreviated). There are three basic blocks in the program, one corresponding to the statements before an `if` condition, one for the true branch of the condition, and finally a basic block containing statements after the branching – the `return` instruction. The basic blocks are connected using edges that are represented using the

```c
#include <stdio.h>

int main(void) {
   int i;
   if (scanf("%d", &i) == 1) {
      printf("%d\n", i);
   }
   return 0;
}
```

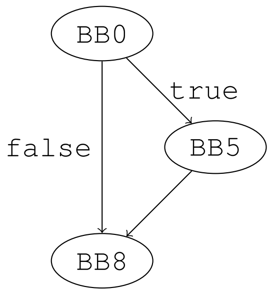
(a) A simple program in C

```llvm
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
  %3 = call i32 @scanf(ptr @.str, ptr %2)
  %4 = icmp eq i32 %3, 1
  br i1 %4, label %5, label %8

5: ; preds = %0
  %6 = load i32, ptr %2, align 4
  %7 = call i32 @printf(ptr @.str.1, i32 %6)
  br label %8

8: ; preds = %5, %0
  ret i32 0
}
```

(b) The equivalent LLVM IR



(c) The equivalent control flow graph of basic blocks

Figure 2.1: Example conversion of a simple C program to LLVM IR

`preds` metadata, e.g., the final basic block starting with label 8 can be reached via either of the other two basic blocks. We can observe that variables are assigned in increasing order; each instruction that has a result assigns to a new local variable (in the human-readable form, local variables are prefixed with a `%` symbol, whereas global variables are prefixed with `@`). The first basic block contains some variable allocations, and then a `call` instruction whose operand is the `scanf` function. The formatting string for `scanf` is defined in the program preamble in the section of constants (left out of the example for brevity), and simply passed here as an argument. Finally, there is a conditional branch instruction whose target depends on the result of the preceding comparison.

2.2 Checking Semantic Equivalence of Functions

The idea of DiffKemp's analysis of equivalence is trying to find so-called *synchronization points* in the compared functions and check that the code between pairs of synchronization points is semantically equal. Most of the time, synchronization points will be placed after each instruction, resulting in an instruction-wise comparison. In the case of pattern application, there may be several instructions between a pair of synchronization points.

Two pieces of code can be considered semantically equal if they both terminate and produce the same output for the same input, or if neither of them terminates. Input of a piece of code consists of the initial memory state and the values of input variables, including global variables. The output of the piece are the values of output variables and the final state of the memory. Furthermore, DiffKemp also checks that both pieces use the same synchronization mechanisms in order to support possible concurrency, and also checks that system and library calls are used in the same way.

Formally, given functions f_1 and f_2 with their instructions I_1, I_2 and sets of variables V_1, V_2, respectively, the problem of checking semantic equivalence can be viewed as finding sets of synchronization points $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$ and synchronization functions $smap : S_1 \leftrightarrow S_2$, mapping between synchronization points between f_1 and f_2, and $varmap : V_1 \leftrightarrow V_2$, defining a mapping of variables, such that chunks of code between the synchronization points are semantically equal with regard to the definition above [20]. The *varmap* bijection represents which pairs of variables are known to be semantically equivalent in the analyzed programs, e.g., given two variables $v_1 \in V_1, v_2 \in V_2$ if $varmap(v_1) = v_2$ then the value of v_1 in the first program is known to be equal to the value of v_2 in the second program. Figure 2.2 shows *smap* and *varmap* bijections for two implementations of a simple function that multiplies by 5 and adds a constant. The values in S_1 and S_2 represent a synchronization point being placed right after the line numbered with the value. We can observe that while typically, synchronization points are located after each instruction, in this case the multiplication is replaced by two other operations (left shift and addition) and the functions are synchronized with respect to this fact, i.e., the functions are synchronized on line 1 (before any operation is performed) and then after line 2 in `f1` and line 3 in `f2`.

```
1 define i32 @f1(i32 %x) {
2     %1 = mul i32 %x, 5
3     %2 = add i32 %1, 10
4     ret i32 %2
5 }
```

```
1 define i32 @f2(i32 %x) {
2     %1 = shl i32 %x, 2
3     %2 = add i32 %2, %x
4     %3 = add i32 %2, 10
5     ret i32 %3
6 }
```

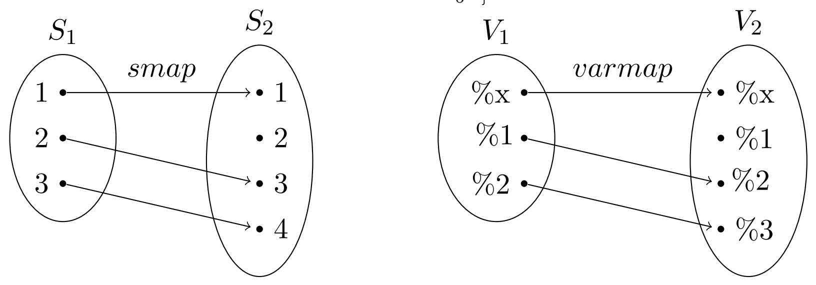


Figure 2.2: Two semantically equal implementations of a function that multiplies its input by 5 and adds a constant and the corresponding *varmap* and *smap*.

2.2.1 Algorithm for Equivalence Checking in DiffKemp

Building on the notion of function equivalence defined in the previous paragraph, we now show how DiffKemp tries to find the correct sets of synchronization points. Algorithm 1 describes how DiffKemp checks for semantic equality of functions f_1 and f_2 with their initial instructions i^1_{in}, i^2_{in}, parameters P_1, P_2 and sets of used global variables G_1, G_2, respectively. The algorithm begins by applying code transformations that do not alter the semantics (Line 1). Among other things, this includes removal of parameters which do not influence the result of the function. After these transformations, if the functions do not have the same number of parameters, they are certainly not semantically equal, since more parameters influence the result in one of the functions (Line 3).

The algorithm then initializes the maps mentioned above. Initially, only the first instructions of the functions can be considered synchronized (Lines 4-5). The parameters are mapped based on their order in the functions (Line 7), while the global variables are mapped based on their names (Line 9).

Algorithm 1: Checking semantic equivalence of functions [20]

```
Input: Functions $f_1$ and $f_2$ to be analyzed and their initial instructions $i^1_{in}$, $i^2_{in}$,
       parameters $P_1$, $P_2$ and used global variables $G_1$, $G_2$
Output: $true$ if $f_1$ and $f_2$ are semantically equal, $false$ otherwise
1  Pre-process $f_1$ and $f_2$ using code transformations
2  if $|P_1| \neq |P_2|$ then
3      return $false$
4  $S_1 = \{i^1_{in}\}$, $S_2 = \{i^2_{in}\}$
5  $smap(i^1_{in}) = i^2_{in}$
6  for $1 \leq i \leq |P_1|$ do
7      $varmap(p^1_i) = p^2_i$
8  for $g_1 \in G_1$ do
9      $varmap(g_1) = g_2 \in G_2$ s.t. $g_1$ has the same name as $g_2$
10 $Q = \{(i^1_{in}, i^2_{in})\}$
11 while $Q \neq \emptyset$ do
12     take any pair $(s_1, s_2)$ from $Q$
13     $p = detectPattern(s_1, s_2)$
14     for $(s'_1, s'_2) \in succPair_p(s_1, s_2)$ do
15         if $s'_1 \in S_1 \vee s'_2 \in S_2$ then
16             if $smap(s'_1) \neq s'_2$ then
17                 return $false$
18         check that blocks $(s_1, s'_1)$ and $(s_2, s'_2)$ are semantically equal
19         if the blocks are not semantically equal then
20             return $false$
21         $S_1 = S_1 \cup \{s'_1\}$, $S_2 = S_2 \cup \{s'_2\}$
22         $smap(s'_1) = s'_2$
23         update $varmap$ according to $p$
24         $Q = Q \cup \{(s'_1, s'_2)\}$
25 return $true$
```

The main loop of the algorithm maintains a worklist Q of synchronization points that have not been checked for semantic equality yet. In each iteration of the main loop, the following takes place:

1. Any pair of synchronization points is taken from the worklist (Line 12).

2. Function *detectPattern* checks if any of the pre-defined patterns is applicable to the code blocks starting at the current pair of synchronization points (Line 13).

3. Function $succPair_p$ (Line 14) finds all possible successor synchronization points with regard to the previously detected pattern p. If no pattern was applicable, there is only one or two possible pairs of successor synchronization points – the true and false branches in the case of a `branch` instruction or otherwise those placed at the following instructions. For each of the possible successor pairs, the algorithm does the following steps:

   (a) If any of the successor synchronization points has already been visited, there must already be a mapping between the synchronization points, otherwise, the functions are not equal (Lines 15-17).

   (b) Blocks of code between the current and successor synchronization points are checked for their semantic equality (Line 18). If no pattern was detected, this step corresponds to a comparison of instructions and their operands. In order for two instructions to be considered semantically equal, their operation codes must match and the operands (taken by their order in both instructions) must either be equal (e.g., in case of constants) or mapped via the *varmap* bijection. Otherwise, if a pattern has previously been detected, this comparison is specific to the pattern.

   (c) The synchronization maps are updated, and the successor synchronization points are enqueued into the worklist (Lines 21-24).

4. Finally, if all synchronization points in the worklist have been processed and no difference has been found, the functions are considered semantically equal (Line 25).

2.3 Semantics-preserving Patterns

There are several patterns defined in DiffKemp that cover some of the common refactorings. Some other refactorings may be covered by the code transformations that are done before analysis of the functions. Each pattern in DiffKemp defines the following [20]:

1. A test indicating applicability of the pattern at given starting lines,

2. A function for computing a pair of successor synchronization points,

3. A condition under which the pattern does indeed preserve semantics, and

4. Mechanism for updating *varmap*.

The idea is that the initial check (Point 1) should be done efficiently, since it is called often, while the condition under which the snippets are semantically equal (Point 3) may use more complex computations since it is used only on a bounded piece of code (as detected in Point 2). DiffKemp currently supports the following patterns:

Changes in Structure Data Types Fields of a structure are accessed using `getelementptr` instruction, which takes a pointer and a field index and returns a pointer to the field. If a field is added or removed, the offsets change and instruction-wise comparison would return false. The pattern adds support for changes of this kind.

Moving Code into Functions Occurs when a block of code is taken out of the analyzed functions and placed into a new function. This is handled by inlining and simplification of the inlined code (dead code elimination and constant propagation).

Changes in Enumeration Values If a value is added to or removed from an `enum`, the values of the enumeration (assigned by the compiler) may change, however, this is typically not considered to be a semantic change. DiffKemp, therefore, compares the identifier string rather than the value set by the compiler.

Changes in Source Code Location A pattern specific to the Linux kernel, in particular calls to warning functions. The content of the warning message is not important, and the call also typically contains the line number of the C file, which is not important either for the semantics of the program.

Inverse Branch Conditions A branching condition is replaced by its inverse and the true and false successors are swapped.

Reordered Binary Operations If an operation is associative and commutative, the operands can be freely reorganized, e.g., $(a + b) + c$ is semantically equal to $(a + c) + b$ if the variables a, b, and c are integers (this is not the case for floats). Such pattern is also an example of a use-case for application of formal methods into DiffKemp.

Local Variable Grouping Detects refactorings related to storage of local variables, e.g., several local integer variables being replaced with an array of integers, where each element corresponds to one of the original variables [32].

Code Relocations Moving a block of code into a different part of the function. It is necessary for the relocated code to be independent of the code that is skipped by the relocation. Currently, only a single relocation of a sequential block of code is supported. This pattern cannot be covered by the algorithm for handling patterns, since it consists of several interconnected phases and must be applied with the lowest priority. Therefore, it is handled explicitly as a part of Algorithm 1. The phases are as follows:

- *Relocation detection* is run if the current blocks on Line 18 are determined as non-equal. It tries to find another synchronization point in one of the functions that would match the current synchronization point in the other function. If such a synchronization point is found, the skipped block is marked as potentially relocated.
- *Relocation matching* is run if instructions were determined as non-equal and a block has previously been marked as potentially relocated. The algorithm moves back to the relocated block in the corresponding function and continues comparison from there. Before moving back, the current synchronization point s_c is remembered so that it can be jumped back to once the relocation has been matched.
- *Relocation checking* is run after the last instruction of the relocated block has been analyzed as equal. This phase checks that the code skipped by the relocation (from the current synchronization point up until s_c) is not data-dependent on the relocated block, i.e., the relocation is semantically equal. Two blocks are data-dependent if one of them reads a value that was written by another block.

Aside from the pre-defined patterns, DiffKemp also allows users to provide their own patterns in the LLVM IR format [19]. The patterns can be parametrized, e.g., to make the pattern applicable to multiple types. The successor synchronization points are determined automatically by DiffKemp, however the new variable mapping must be defined using the `diffkemp.mapping` function in the LLVM IR which is called in each pattern function just before its exit. The call contains a list of variables, and the new mappings to be added to *varmap* are determined based on their order.

Chapter 3

SMT Solving

In recent years, significant progress has been made in the field of effective automated decision procedures of various logics, mainly propositional logic (implemented in so-called SAT solvers) and theories in first-order logic (implemented in so-called SMT solvers). Consequently, a lot of today's tools for hardware and software verification employ automated decision procedures as a part of their verification mechanisms, since a lot of these problems can naturally be encoded using formulae. Furthermore, state-of-the-art solvers are typically very efficient for most common inputs despite the problem of checking satisfiability of a formula being very hard (NP-complete in the case of SAT solving and even undecidable for some first-order logic theories).

This chapter briefly describes what SMT solving is and how it works from a verification tool's point of view, without diving deep into the inner workings of SMT solvers. Section 3.1 introduces the notion of SMT solving and gives an overview of the most common theories that many current decision procedures try to tackle. Section 3.2 examines the current state of SMT solvers, their feature set, while focusing on the intended usage within DiffKemp's analysis algorithm.

3.1 Principles of SMT Solving

Given a propositional logic formula, determining whether there exists an assignment of variables (a so-called model) under which the formula evaluates to true, is commonly referred to as the Boolean Satisfiability Problem, usually abbreviated as SAT. The problem of determining whether a formula is satisfiable has been proven to be NP-complete [9], i.e., it is unlikely that a polynomial algorithm for solving this problem exists. Despite being a very hard problem, a lot of effort has been put towards implementing a decision procedure that is efficient for a large part of the most common formulae, e.g., by employing various heuristics.

While a lot of problems in software verification and analysis can directly be expressed using propositional formulae, other problems are easier to encode using a more expressive language, for example using first-order logic [7]. However, with increasing expressiveness, it becomes harder to decide whether a given formula is satisfiable – the problem of checking satisfiability of a general first-order logic formula is undecidable. A common compromise can be achieved by restricting interpretation of certain functions and predicate symbols, and consequently, restricting semantics of the logic. Thanks to such restrictions, the satisfiability problem becomes decidable, which makes it possible to implement specialized decision procedures that exploit properties of the restrictions. Restricting the semantics of a first-order logic is commonly referred to as using first-order theories (e.g., a theory of integers, a theory of arrays, etc.). The problem of checking whether a formula is satisfiable in a theory is usually referred to as Satisfiability Modulo Theories (SMT).

Solving the SMT problem is a topic that is still under active research. On the highest level, there exist two approaches. One option is converting the first-order-logic formula restricted by a theory to a propositional formula and using a SAT solver to find the model of this formula. This approach, however, blows up exponentially, which makes it impractical for all but the simplest of theories and formulae. For this reason, most current SMT solvers use a different approach, often referred to as *lazy*. In this case, a SAT solver is used to reason just about the propositional connectives. The SMT solver then checks if the model returned by the SAT solver is consistent with the theory that is being used in a so-called theory solver. If the theory solver validates that the model is also consistent with the theory, the input formula is satisfiable. Otherwise, a new lemma is added as a conjunction to the examined formula that excludes the previously returned invalid model. Thanks to using the lazy approach, modern SMT solvers can leverage improvements in efficiency of SAT solvers. Furthermore, this approach is highly modular, making it possible to support lots of theories and possibly also their combinations.

3.1.1 Most Common Theories

There are quite a lot of theories that modern SMT solvers support, therefore giving the complete list would be impossible. This section aims to give an overview of the ones most relevant to software verification. On the highest level, theories can be differentiated based on their approach to quantifiers – a theory can either allow quantifiers or disallow them (so-called quantifier-free theories). Quantified theories are harder to decide from a time complexity standpoint. Then, arithmetic theories typically differentiate whether non-linear arithmetic is supported. A linear arithmetic may contain multiplication of variables only by a constant coefficient, i.e., multiplication of variables is disallowed, whereas non-linear arithmetic theories remove this restriction. Some theories are also a combination of multiple simpler theories, these are not considered in this text.

Excluding the high-level categorization described above, the following theories are common in today's SMT solvers [7]:

Uninterpreted Functions with Equality The formulae may contain any functional and predicate symbols. The symbols are differentiated by their names and arity, but do not have any semantic interpretation. The theory also allows checking for equality with the usual semantics.

Integer Arithmetic Allows use of all integer constants, function symbols $\{+, -, *\}$ and the predicate symbols for equality and inequality $\{=, \leq, \geq, \ldots\}$.

Real Arithmetic Same as Integer Arithmetic, but the values may be any rational number.

Difference Logic A subset of either Integer or Real Arithmetic where all atoms are limited to the form $x - y \bowtie c$, where $\bowtie \in \{=, \leq, \geq\}$, c is a constant and x and y are variables. This theory can be solved more efficiently than its more expressive counterparts.

Floating-point Arithmetic Similar to Real Arithmetic but considers behavior as specified in the IEEE 754 standard, e.g., overflows, rounding, undefined operations and special values (infinity and `NaN`).

Bit Vectors An extension of Integer Arithmetic, where every number is represented as a fixed-size sequence of bits. Aside from the standard arithmetic operations, bitwise operations are also typically allowed.

Theories for More Complex Data Structures There are also some theories that aim to support composite data types, such as arrays, strings, sets, multi-sets, lists, etc. These are, however, not as relevant to this thesis.

3.2 State-of-the-art SMT Solvers

A lot of SMT solvers have been developed in recent years. Different solvers have different strengths and weaknesses and have various additional features. In order to choose the right solver for a specific use-case, it is useful to compare the available solvers. There are lots of possible evaluation criteria that can be used, for example:

- the availability and the license of the solver,
- the API available in various programming languages,
- the supported theories,
- the performance on various classes of examples,
- the support for quantifiers (oftentimes, quantifiers are completely forbidden),
- the support for a standardized format called SMT-LIB [6],
- the ability to provide a model in case of a satisfiable formula, and
- the ability to provide proofs of unsatisfiability and generate unsatisfiable cores, i.e., clauses of the formula that are in conflict and make the whole formula unsatisfiable.

This section aims to give a brief overview of the current SMT solvers that are available under an open-source license and have a C++ API (since DiffKemp is implemented in this language). Furthermore, we require the solvers to support as many theories as possible in order to be able to encode various program fragments. For example, the theory of floating-point numbers is not very common since its time complexity is very high. However, lots of programs contain floating-point arithmetic and in this case, it may be better (or at least have the possibility) to be able to model such operations precisely rather than approximating with a theory of real numbers. On the other hand, support for unsatisfiable cores and for providing a model is not as important in our case, since we only want to check whether a formula is satisfiable or unsatisfiable and the reason is not as important.

Z3 A widely used solver developed by Microsoft Research. Z3 is provided under the MIT license and accepts the SMT-LIB format and provides APIs in C++, Java, Python, .NET and OCaml. Most theories related to arithmetic are supported, including floating-point arithmetic [23]. Both universal and existential quantifiers can be used within the formulae. Z3 is able to find unsatisfiable core, as well as return a model.

CVC5 Another commonly used SMT solver that builds on top of its predecessor – CVC4. All theories that are defined in the SMT-LIB format are supported, including but not limited to linear arithmetic, non-linear arithmetic, floating-point arithmetic, bit vectors, strings, arrays, and quantifiers [3]. CVC5 is provided under the BSD license, accepts input in the SMT-LIB format, as well as provides a C++, Python and Java APIs. It is able to provide models of satisfiable formulae, as well as find unsatisfiable core.

Bitwuzla A relatively new solver developed as a fork of Boolector. The solver supports both quantified and quantifier-free theories of fixed-sized bit vectors, arrays, floating-point arithmetic and uninterpreted functions [27]. Bitwuzla is available under the MIT license and can be used via its C++ or Python API. Bitwuzla extended Boolector with several new theories, including the theory of floating-point arithmetic, and also added support for unsatisfiable core extraction.

OpenSMT 2 and Yices 2 These two solvers offer a very similar feature set. They both provide a C, C++ and Python APIs and support quantifier-free theories, but neither of them has support for floating-point arithmetic [12, 8]. The solvers are available under open-source licenses (OpenSMT has MIT license, Yices has GPLv3 license).

Performance of Solvers Performance is a crucial aspect in software verification and DiffKemp is no exception to this, since it aims to scale to large projects as best as possible. Evaluating efficiency of SMT solvers is a complex task, since solvers use various heuristics and while one heuristic could work well for a class of problems, it could be very inefficient for some other class of formulae. A competition called SMT-COMP has been created with the goal of fairly assessing qualities of various SMT solvers [5]. There are various tracks in the competition, for example *Incremental Track* aims to evaluate solvers when interacting with an external verification framework, e.g., a model checker where multiple queries are incrementally made. On the other hand, *Single Query Track* is the exact opposite of this. For integration with DiffKemp, we mostly care about single query use, as will be discussed in later chapters. Investigating 2023 results shows that a lot of categories were won by CVC5, with Z3 trailing behind in the second place. There are, however, some categories, where other solvers shined, e.g., Bitwuzla won the `FPArith` category, which contains floating-point benchmarks.

Abstract Interfaces Each SMT solver provides its own API in the programming languages that it supports. The formula to be checked for satisfiability is therefore built using various classes and functions from the solver's API. There have been attempts to create an abstract interface that would allow building the formula on an abstract level without relying on implementation details of individual solvers. This allows easily swapping out a solver to experiment with its performance on the particular problem. Currently, the most promising example of this approach seems to be the SMT-SWITCH tool [21]. Many aforementioned solvers are supported according to the documentation, however, floating-point theories are currently not supported. This may be a limiting factor for use within DiffKemp, as discussed previously.

Chapter 4

Incorporating Formal Methods into DiffKemp

While DiffKemp scales really well to large programs thanks to its approach to analysis described in Chapter 2, there are still many cases where the tool reports semantic inequality despite the functions being equal. This is usually caused by code transformations not being sufficient and/or DiffKemp not having a pattern defined for such a change. Figure 4.1 shows two versions of an example program that DiffKemp currently fails to analyze as semantically equal. Multiplication by a constant 5 at line 3 of Figure 4.1a has been replaced by an equivalent left bitwise shift of 2 and addition at lines 3 and 4 of Figure 4.1b.

```
1 define i32 @f(i32 %0) {
2   %2 = sub i32 %0, 2
3   %3 = mul i32 %2, 5
4   %4 = udiv i32 %3, 2
5   %5 = add i32 %4, 2
6   ret i32 %5
7 }
```

(a) First version of the program using multiplication

```
1 define i32 @f(i32 %0) {
2   %2 = sub i32 %0, 2
3   %3 = shl i32 %2, 2
4   %4 = add i32 %3, %2
5   %5 = udiv i32 %4, 2
6   %6 = add i32 %5, 2
7   ret i32 %6
8 }
```

(b) Second version of the program using bitwise shift

Figure 4.1: Two equivalent versions of a program computing the function $f(x) = \frac{(x-2)*5}{2}+2$. The multiplication by 5 in one of the programs has been replaced with an equivalent bitwise shift and addition.

It would certainly be possible to define a new pattern that would be able to handle this case. However, there are many more similar examples related to refactoring of arithmetic and logic expressions that come to mind, e.g., application of distributive, associative, and commutative laws, various optimizations related to constants (as seen in the example), etc. Manually implementing all such cases as patterns in DiffKemp is, therefore, not feasible. The goal of this thesis is to facilitate analysis of these cases. In this chapter, we propose an integration of formal methods based on SMT solving into DiffKemp that allows an automated verification of such changes as semantically equal.

The overall approach can be summarized as follows. Whenever a possible semantic change is found and no pattern, including a relocation, is applicable (i.e., Line 20 in Algorithm 1), we try to perform the following steps:

1. Find the closest pair of instructions wrt. control flow, after which the code can be synchronized using *varmap* and *smap*. The code blocks between the differing instructions and this pair of instructions needs to be checked for semantic equality. In order to ensure an efficient integration into DiffKemp's analysis algorithm, we only focus on sequential code blocks that do not manipulate the memory and have no side effects.

2. If such a pair of code blocks was detected, encode the problem of checking their semantic equality into the SMT problem and use an SMT solver to check if the blocks are equal.

3. If the SMT solver verified equality of the blocks in the provided amount of time, update *varmap* and *smap* accordingly, and continue analysis based on Algorithm 1. Otherwise, report semantic inequality of the programs.

The following sections go into detail of the individual steps of the approach outlined above. First, Section 4.1 describes how we can detect the blocks to be analyzed by our SMT-based procedure. Building on the notion of semantic equivalence defined in the previous chapters, Section 4.2 details how to encode the equivalence property into a formula to be passed into an SMT solver. Finally, Section 4.3 describes an extension to this approach that facilitates verification of advanced inverse branch condition that the current pattern defined in DiffKemp cannot handle.

4.1 Identifying Relevant Code Snippets

Since SMT is an NP-hard problem, i.e., an SMT solver can take a really long time to return an answer, our solution needs to minimize the number of calls to the SMT solver. Therefore, we need to find blocks of code that are worthwhile analyzing. For example, checking equivalence of blocks of code is a waste of time, if the instructions following the given blocks are completely different, e.g., one performs an arithmetic operation, while the other one is a `ret` instruction. Even if the blocks were analyzed as equal using the SMT solver, the comparison would fail immediately afterward, since the following instructions are different.

To this end, we use a similar approach as is already used for relocation detection described in Section 2.3. Algorithm 2 shows how blocks to be analyzed are found. It tries to find a synchronization point after the provided differing instructions. All instructions after the found synchronization point must be synchronized in order for the analysis to be worthwhile. At first, the blocks to be analyzed are initialized as empty and the differing instructions marking the beginning of the blocks are backed up (Lines 1-2). The algorithm then checks every pair of instructions in the remainder of the basic blocks for synchronization. The instruction pairs are evaluated in the order of control flow in the current basic block. Function `cmpBasicBlocksFrom` (Line 6) used for checking synchronization compares instructions using Algorithm 1 but stops at the end of the current basic block. Since the procedure modifies the contents of *varmap* and *smap*, they need to be backed up (Line 5) and restored (Line 8) in order to allow multiple successive calls to this function. If the instructions can be synchronized until the end of the basic blocks, the collected blocks are returned (Line 7). As a part of the algorithm, once an instruction is added to R_1 it is going to be part of all the remaining blocks that can be found by the algorithm. For this reason, we check whether the instruction being added to R_1 is supported (the list of all supported instructions can be seen in Section 4.2.1) and stop searching for a synchronization point if it is not (Line 14). This is an optimization done to reduce the amount of computation in cases where SMT solving would not be successful.

Algorithm 2: Finding blocks to be analyzed using an SMT solver

```
    Input: Differing instructions $s_1$, $s_2$
    Output: Blocks $R_1$, $R_2$ that are followed by a synchronization point, i.e., are
            worth analyzing
 1  backup $s_1$ and $s_2$
 2  $R_1 = R_2 = []$
 3  while $op(s_1) \neq$ branch do
 4      while $op(s_2) \neq$ branch do
 5          backup $varmap$ and $smap$
 6          if $cmpBasicBlocksFrom(s_1, s_2) = equal$ then
 7              return $R_1, R_2$
 8          restore $varmap$ and $smap$
 9          append $s_2$ to $R_2$
10          $s_2 = succ(s_2)$
11      restore $s_2$
12      $R_2 = []$
13      if $s_1$ is not a supported instruction then
14          return $[], []$
15      append $s_1$ to $R_1$
16      $s_1 = succ(s_1)$
17  restore $s_1$ and $s_2$
18  return $[], []$
```

Applying Algorithm 2 to the example given in Figure 4.1 would result in blocks containing line 3 in the first version and lines 3-4 in the second version being returned. This is caused by the fact that the functions can be synchronized on the `udiv` instruction, since the instructions in the rest of the functions have the same operation codes and their operands can either be mapped via *varmap* or their values are equal in the case of constants.

4.1.1 Multiple Synchronization Points

The example in Figure 4.1 is rather simple, i.e., there is only one possible synchronization point. However, in some cases, there may be several synchronization points. For example, consider a case corresponding to the *reordered binary operations* pattern that our integration aims to cover as well, e.g., replacing $(a+b)+(c+d)$ with $(a+c)+(b+d)$. This results in three `add` operations being created, as can be seen in Figure 4.2. The closest synchronization point found by Algorithm 2 is on lines 3 in both the programs. Since the first two lines were skipped, `cmpBasicBlocksFrom` can map %5 and %6 in Figure 4.2a to %5 and %6 in Figure 4.2b, respectively, and, therefore, evaluate the remainder of the basic blocks as equivalent. However, the corresponding blocks that were skipped are not semantically equal – the semantically equal blocks that we are looking for are the blocks containing all three `add` instructions, i.e., the synchronization point at the `ret` instruction in both programs.

```
%5 = add i32 %0, %1
%6 = add i32 %2, %3
%7 = add i32 %5, %6
ret i32 %7
```

(a) (%0 + %1) + (%2 + %3)

```
%5 = add i32 %0, %2
%6 = add i32 %1, %3
%7 = add i32 %5, %6
ret i32 %7
```

(b) (%0 + %2) + (%1 + %3)

Figure 4.2: The LLVM IR of an equivalent refactoring corresponding to the *reordered binary operations* pattern. Correct analysis requires searching for multiple synchronization points.

This problem could be remedied by analyzing the largest possible blocks, i.e., looking for a synchronization point from the end of the basic blocks rather than the beginning as shown in Algorithm 2. However, trying to compare semantics of larger blocks using an SMT solver is computationally more complex and has a higher likelihood of not finishing in a reasonable time. For this reason, we opted for a different approach. We start with the smallest blocks, i.e., the closest synchronization point, and check their equivalence. If the blocks are found to not be equal, we search for a different synchronization point. We repeat this process until all possible synchronization points have been exhausted.

While in some cases, the first outlined possible solution may yield better results wrt. performance (e.g., in the *reordered binary operations* example), the difference seems negligible, especially since multiple possible synchronization points seem to be quite rare from our experiments.

4.2 Encoding Equivalence into SMT Formulae

Now that we have detected the two blocks of code to be analyzed, we need to check whether they are equivalent. The definition of equivalence of blocks is similar to that of functions described in Section 2.2. Two blocks of code are semantically equal if they both terminate and for the same input, their execution produces the same output, or neither of them terminates. We can ignore termination, because we are only dealing with sequential blocks of code in this work, hence there are no loops and no possibility of the code not terminating.

By input in this case, we mean values of local variables that were defined outside the blocks and are used for computation in the analyzed blocks. Unlike with functions, the mapping of inputs between the two analyzed blocks is slightly more complicated than just mapping function parameters by their order. In the case of a general block of code, we need to map inputs based on their semantics – this can be done nicely using *varmap* that was computed by DiffKemp's main analysis algorithm. Similarly, the output of a block corresponds to the local variables defined in the analyzed block that are used outside the block. Since in this work we only focus on blocks of code that do not manipulate memory, we can ignore reads and writes through pointers and to global variables. While it would be possible to model the memory, e.g., using the theory of arrays, it would make the problem significantly more difficult and more expensive to compute, which is not desirable wrt. DiffKemp's focus on scalability.

In order to check equivalence of two blocks using an SMT solver, we need to construct a first-order logic formula describing their equivalence as defined above. Let $InVar_1$ be the set of input variables of the first block, $OutVar_1$ the set of output variables of the first block, *outmap* a bijection between output variables in the two blocks (discussed in more detail in Section 4.2.2), and $Block_1$ and $Block_2$ the encoding of operations in (i.e., the formula representing the semantics of) the first and the second block, respectively. $Block_1$ and $Block_2$ are semantically equal, if and only if Formula 4.1 is unsatisfiable. Intuitively, if we give both blocks the same input, they need to produce the same output in order for the blocks to be equivalent. If the formula has a model, the model corresponds to the inputs under which the outputs differ.

$$\begin{aligned}&\bigwedge_{v_1 \in InVar_1} v_1 = varmap(v_1) \wedge \\ &Block_1 \wedge Block_2 \wedge \\ &\neg \bigwedge_{out_1 \in OutVar_1} out_1 = outmap(out_1)\end{aligned} \tag{4.1}$$

4.2.1 Encoding LLVM IR into SMT Formulae

In order to encode operations in the first and the second block into $Block_1$ and $Block_2$ in Formula 4.1, we can exploit the fact that we are dealing with sequential blocks of code and the SSA property of LLVM IR. Thanks to the SSA property, each instruction in each block performs a single operation over its operands and stores its result into a new virtual register. Since we do not support memory manipulation, the instructions are going to be either arithmetic-logic operations, comparisons, cast, `select` or `call` instructions.

With regard to all of the above, encoding LLVM IR instructions into an SMT formula is rather simple. We create a new SMT variable for every virtual register that is used or defined in the analyzed block. The variable needs to be of the type corresponding to the virtual register, e.g., a float virtual register is encoded as a float SMT variable. Integer types in LLVM IR are encoded as bit vectors in the SMT formula. The only exception to this is an integer type with a bit width of 1, i.e., boolean (LLVM IR does not have an explicit `bool` type), that needs to be encoded as a boolean variable in the SMT formula. Then, we create a conjunction of clauses, each corresponding to one instruction and having the form $res = operation(op_1, op_2, \ldots)$. For example, the instruction `%2 = add %1, 2` would be encoded as a clause $var_2 = var_1 + 2$. Table 4.1 gives an overview of the supported instructions and their encoding, the assignment to the result variable has been omitted for brevity. The encoding was constructed based on semantics in the LLVM Reference Manual [30]. All unary, binary and bitwise binary instructions are supported. The remaining instruction types are not supported, or supported only partially.

Encoding of Function Calls As mentioned above, our solution also aims to support the `call` instruction. If we wanted to support calls to all kinds of functions, the effects of the called function would have to be encoded into the SMT formula. This could blow up exponentially, hence we limit ourselves to built-in functions with well-defined semantics, e.g., calls to trigonometric functions. While some modern SMT solvers (e.g., Z3 [24]) offer a limited support for trigonometric functions, only the `Real` type is usually supported, whereas in programming languages, floating-point numbers are used. For this reason, we encode

Table 4.1: A list of supported LLVM IR instructions and their encoding into SMT. The encoding uses an infix rather than prefix notation that is used in the SMT-LIB format.

Type	**Instruction**	**Encoding**
Unary	`fneg float %1`	$fp.neg(op_1)$
Binary	`add %1, %2`	$bvadd(op_1, op_2)$ [1]
	`fadd %1, %2`	$fp.add(op_1, op_2)$
	`sub %1, %2`	$bvsub(op_1, op_2)$ [1]
	`fsub %1, %2`	$fp.sub(op_1, op_2)$
	`mul %1, %2`	$bvmul(op_1, op_2)$ [1]
	`fmul %1, %2`	$fp.mul(op_1, op_2)$
	`udiv %1, %2`	$bvudiv(op_1, op_2)$ [2]
	`sdiv %1, %2`	$bvsdiv(op_1, op_2)$ [2]
	`fdiv %1, %2`	$fp.div(op_1, op_2)$
	`urem %1, %2`	$bvurem(op_1, op_2)$
	`srem %1, %2`	$bvsrem(op_1, op_2)$
	`frem %1, %2`	$fp.rem(op_1, op_2)$
Bitwise binary	`shl %1, %2`	$bvshl(op_1, op_2)$ [1]
	`lshr %1, %2`	$bvlshr(op_1, op_2)$
	`ashr %1, %2`	$bvashr(op_1, op_2)$
	`and %1, %2`	$bvand(op_1, op_2)$ [3]
	`or %1, %2`	$bvor(op_1, op_2)$ [3]
	`xor %1, %2`	$bvxor(op_1, op_2)$ [3]
Conversion	`trunc %1 to ty`	$extract(op_1, 0, ty.bits - 1)$
	`zext %1 to ty`	$zero\_extend(op_1, ty.bits - op_1.bits)$
	`sext %1 to ty`	$sign\_extend(op_1, ty.bits - op_1.bits)$
	`fptrunc %1 to ty`	$to\_fp(op_1, ty)$
	`fpext %1 to ty`	$to\_fp(op_1, ty)$
	`fptoui %1 to ty`	$fp.to\_ubv(op_1, ty)$
	`fptosi %1 to ty`	$fp.to\_sbv(op_1, ty)$
	`uitofp %1 to ty`	$to\_fp(op_1, ty)$
	`sitofp %1 to ty`	$to\_fp(op_1, ty)$
Other	`icmp ugt %1, %2` [4]	$bvugt(op_1, op_2)$
	`fcmp ogt %1, %2` [4]	$\neg fp.isNaN(op_1) \wedge \neg fp.isNaN(op_2) \wedge fp.gt(op_1, op_2)$
	`fcmp ugt %1, %2` [4]	$fp.isNaN(op_1) \vee fp.isNaN(op_2) \vee fp.gt(op_1, op_2)$
	`select` %1, %2, %3	$ite(op_1, op_2, op_3)$
	`call`	Uninterpreted function call

[1] The operation may use the `nsw`/`nuw` flags, i.e., it may overflow – cf. Section 4.2.1.
[2] The operations may use the `exact` flag, cf. Section 4.2.1.
[3] If the operands are boolean (i.e., LLVM IR integers with a width of 1 bit), the logical counterparts are used instead of bit-vector operations.
[4] The other comparison conditions (e.g., `ult`, `ule`, ...) are implemented in the same manner.

the operations using uninterpreted functions. This approach may lead to imprecision, but it ensures soundness of the encoding.

Encoding of Integer Overflows In most modern programming languages, arithmetic operations may result in undefined behavior under certain circumstances. A common example of this are integer overflows and underflows in signed arithmetic. LLVM IR denotes a possible overflow using the `nuw` (no unsigned wrap) and the `nsw` (no signed wrap) flags [30]. For example, if the `nuw` or the `nsw` keywords are present in the `add` instruction, the result of the operation is a so-called *poison value* if unsigned or signed overflow, respectively, occurs. A poison value can be interpreted as an undefined value, i.e., any value of the type. To ensure sound encoding of overflowing operations, we need to encode clauses corresponding to overflowing operations as $\mathit{cond} \implies \mathit{res} = \mathit{operation}(\mathit{op}_1, \mathit{op}_2, \ldots)$ where *cond* defines the conditions under which the *operation* does not cause an undefined behavior. Such encoding ensures that if there is a possibility of an overflow, *res* will remain a free variable, i.e., it may have any value. The *cond* formula can be encoded thanks to the bit-vector operations, e.g., for unsigned addition on x bits, we can perform the addition on $x + 1$ bits and then extract the sign bit to check for overflow.

Encoding of Exact Division The `udiv` and `sdiv` instructions may contain the `exact` flag. If the keyword is present, the result value of the division is a *poison value* if the result would be rounded. We encode these instructions similarly to the overflowing integer operations – the precondition *cond* in this case checks that the remainder of the division is zero.

4.2.2 Identifying Output Variables

The last piece of information required to construct the formula for an SMT solver are output variables and their mapping *outmap* between the two blocks. An output variable of a block is a variable defined in the block that is used outside the block. The output variables can be easily collected using LLVM's API for navigating the intermediate representation. On the other hand, creating the *outmap* bijection mapping output variables between the two blocks analyzed for equivalence is a more complex problem.

There are two cases that need to be distinguished. The first one being an output variable of a block that is used in the same basic block in the LLVM IR as the block that it was defined in. Notice that in Algorithm 2, when a synchronization point has been found (Line 7), *varmap* is not restored to its original state. Since the function `cmpBasicBlocksFrom` compares the remainder of the basic blocks, it establishes the necessary mapping of output variables of this kind into *varmap* – we can transfer this mapping into *outmap*.

On the other hand, if an output variable of the analyzed block is used in a different basic block (typically in a Φ-node that merges values from multiple branches), we cannot use *varmap* to construct *outmap* since `cmpBasicBlocksFrom` may not have analyzed the other basic blocks, yet. In this case, we limit ourselves to cases with only one output variable of such a kind in Block_1 and Block_2. From our experiments, this did not have a negative effect on the precision of our solution. However, if this limitation becomes restricting in the future, DiffKemp could either check all possible mappings of the output variables of this kind or employ additional heuristics to determine the correct output variable mapping.

4.2.3 Example Encoding

Following up on the motivation example given in Figure 4.1, we can demonstrate how the equivalence is encoded into an SMT formula. As discussed in Section 4.1, the compared blocks are line 3 in the first program and lines 3 and 4 in the second program. Assume that DiffKemp's main analysis algorithm has established a mapping (as a part of *varmap*) of the `%2` register in the first version to the `%2` register in the second version. The output variable in the first version is the `%3` register, whereas in the second version, it is the `%4` register. The *outmap* mapping in this case can be established based on *varmap* because `cmpBasicBlocksFrom` during Algorithm 2 analyzes the `udiv` instruction on lines 5 and 4, respectively, and maps its operands, i.e., the output variables of the analyzed blocks.

We denote var_i^f, $f \in \{1, 2\}$ as a variable corresponding to the virtual register `%i` in function f. All the variables are 32-bit bit vectors. Considering all of the above, the formula to solve in order to check equivalence of the two blocks is:

$$\begin{aligned}
& var_2^1 = var_2^2 \land && \text{Input mapping} \\
& var_3^1 = bvmul(var_2^1,\ 5) \land && Block_1 \text{ encoding} \\
& var_3^2 = bvshl(var_2^2,\ 2) \land && Block_2 \text{ encoding} \\
& var_4^2 = bvadd(var_3^2,\ var_2^2) \land && \\
& \neg(var_3^1 = var_4^2) && \text{Output mapping}
\end{aligned}$$

This formula is unsatisfiable in the theory of bit vectors, i.e., the blocks are semantically equal. The analyzed blocks can, therefore, be skipped, *varmap* can be updated with a mapping of `%3` to `%4`, and DiffKemp's main analysis algorithm can continue comparing line 4 in the first function with line 5 in the second function. Since the remaining instructions are identical, the two functions will be analyzed as semantically equal.

4.3 Detection of Advanced Inverse Branch Condition

Even though DiffKemp has a built-in pattern for detecting inverse branching conditions, it only works in the most straight-forward cases, i.e., when the condition in one function is exactly an inverse of a condition in the other function and the true and false successors are swapped. An example of this is replacing `icmp ule` (unsigned less or equal) with `icmp ugt` (unsigned greater than). In our experiments, we have observed more advanced cases that are not covered by the existing pattern and that would be suitable for application of formal methods.

Figure 4.3 gives an example of such a case from the EqBench benchmark (cf. Section 6.2). The condition $a > 100$ has been replaced with $x < 101$, i.e., the comparison operator is not inverted from a syntactic point of view. However, given that the variables a and x are integers, it is indeed a case of an *inverse branch condition*. It would be possible to extend the built-in pattern to cover this case, but doing so in a general and sound manner seems too complicated – using an SMT solver is more suitable to cover this case.

We extend our original approach to cover this case as follows. If at first the SMT solver returns SAT, i.e., the blocks are not semantically equal, we check whether one of the blocks contains a possibly invertible comparison instruction that we could apply the

```c
int f(int a) {
  int r;
  r = 0;
  if (a > 100) {
    r = a - 10;
  } else {
    r = f(a + 11);
    r = f(r);
  }
  return r;
}
```

(a) First version of the program

```c
int f(int x) {
  int r;
  r = 0;
  if (x < 101) {
    r = f(11 + x);
    r = f(r);
  } else {
    r = x - 10;
  }
  return r;
}
```

(b) Second version of the program

Figure 4.3: Two semantically equal versions of a program, where a condition is inverted and the branches are swapped. This example is taken from the EqBench benchmark. [2]

inverse branch condition pattern to. A compare instruction is possibly invertible if its result is an output variable of the analyzed block and the result is used in a `branch` instruction. If such an instruction exists in one of the blocks, we encode the equivalence of blocks into an SMT formula once more, but this time, we invert the invertible condition. If the solver returns UNSAT, the blocks are semantically equal provided that the successors in the `branch` instruction are indeed swapped. To check that this is the case, we follow the same algorithm as the standard *inverse branch condition* pattern, i.e., we swap the successors and return to DiffKemp's main analysis algorithm that will take care of comparing the branch instructions and their successor basic blocks.

```llvm
1  define i32 @f(i32 %0) {
2    %2 = icmp sgt i32 %0, 100
3    br i1 %2, label %3, label %5
4
5  3:
6    ; base case
7    br label %9
8
9  5:
10   ; recursive calls
11   br label %9
12
13 9:
14   %.0 = phi i32 [ %4, %3 ], [ %8, %5 ]
15   ret i32 %.0
16 }
```

(a) First version of the program

```llvm
1  define i32 @f(i32 %0) {
2    %2 = icmp slt i32 %0, 101
3    br i1 %2, label %3, label %7
4
5  3:
6    ; recursive calls
7    br label %9
8
9  7:
10   ; base case
11   br label %9
12
13 9:
14   %.0 = phi i32 [ %6, %3 ], [ %8, %7 ]
15   ret i32 %.0
16 }
```

(b) Second version of the program

Figure 4.4: The LLVM IR of the two programs given in Figure 4.3.

Figure 4.4 shows the shortened LLVM IR of the two programs given in Figure 4.3. DiffKemp's main analysis algorithm identifies the differing instructions on line 2 in both of the functions. The *inverse branch condition* pattern is not applied in this case, since the comparison operators are not an exact inverse of each other. At first, our extension identifies the nearest synchronization point on line 3, i.e., on the `branch` instructions. Using an SMT solver as described in Section 4.2 results in the `icmp` instructions being evaluated as not equal. However, the `icmp` instruction is possibly invertible, since its result is used as an operand for the `br` instruction. Therefore, we try to invert the condition – we encode the left program as $var_2^1 = \neg bvsgt(var_0^1,\ 100)$ instead of $var_2^1 = bvsgt(var_0^1,\ 100)$. With such encoding, the SMT solver evaluates the blocks as equal, hence it is a possible case of the *inverse branch condition*. Therefore, we must swap the successors of the `branch` instruction and let DiffKemp's main analysis compare the successor basic blocks.

Chapter 5

Implementation of the Extension

The solution proposed in Chapter 4 has been implemented in DiffKemp. At the time of writing, a pull request introducing an SMT-based checking of code snippet equivalence is undergoing code review in the upstream repository of DiffKemp on GitHub[1]. While working on the implementation, we have encountered several bugs in the implementation of DiffKemp's main analysis algorithm. The fixes to these bugs have already been merged into the main development branch[2].

The rest of this chapter is organized as follows. Section 5.1 gives a brief overview of DiffKemp's overall architecture. Section 5.2 describes the integration of our solution into the existing architecture, as well as some of the more important implementation details.

5.1 DiffKemp Architecture

DiffKemp's analysis is split into two phases:

1. **Snapshot Generation** During this phase, source code of the analyzed project is compiled into LLVM IR and additional metadata is added, resulting in a so-called snapshot.

2. **Comparison Phase** Takes two snapshots (typically two versions of the same project) and performs comparison of their functions using principles described in Chapter 2.

The implementation of DiffKemp consists of two parts – a Python front end and a C++ analysis library, called SimpLL. The Python front end takes care of snapshot generation and provides a user-friendly way of running DiffKemp, e.g., it provides more convenient access to configuration of SimpLL and also simplifies its output. SimpLL, implemented in C++ to achieve high efficiency, implements the snapshot comparison phase. It is usually invoked from the Python front end via Foreign Function Interface (FFI)[3], however it can also be run as a standalone binary. Aside from the main analysis algorithm, SimpLL also implements a number of so-called LLVM passes that implement code transformations used as a part of DiffKemp's analysis algorithm described in Section 2.2.1.

[1] https://github.com/diffkemp/diffkemp/pull/322

[2] https://github.com/diffkemp/diffkemp/pull/323, https://github.com/diffkemp/diffkemp/pull/325, and https://github.com/diffkemp/diffkemp/pull/330

[3] A mechanism which allows a program in one programming language to call functions of a program in a different programming language. In the case of DiffKemp, C++ functions are called from Python.

Our solution extends the comparison phase, i.e., SimpLL, thus we give a brief overview of the components that handle the analysis (components that are not relevant to this work are omitted). The overall architecture of SimpLL can be seen in Figure 5.1:

LLVM IR Parser A component coming from the LLVM infrastructure that takes care of reading the analyzed LLVM programs that are typically a part of DiffKemp snapshots.

Module Analyzer Applies code transformations to the analyzed programs and starts comparison of the selected pair of functions.

Module Comparator Handles comparison of a pair of functions, takes care of producing the final result of the analysis, including the verdict and information about what the semantic difference is. If a semantic difference is caused by a function call, the Module comparator tries to inline the function call and calls the Differential function comparator again in order to possibly prevent a false positive.

Differential Function Comparator Implements the main analysis algorithm described in Section 2.2.1 that is used for comparing two versions of a function.

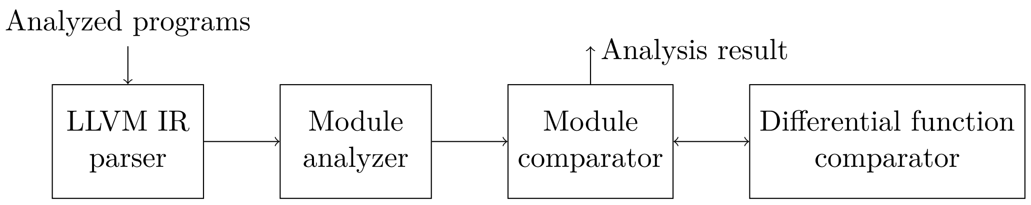


Figure 5.1: Architecture of SimpLL

5.2 Extension of SimpLL

We have implemented our extension proposed in Chapter 4 as a new SimpLL component called **SMT block comparator**. This component interacts with the Differential function comparator as described in the aforementioned chapter, i.e., when a pair of differing instructions is found and no pattern can be applied, the SMT block comparator is called. If the instructions are analyzed as equal using an SMT solver, the control is returned to the Differential function comparator, and the analysis continues. Figure 5.2 shows the new architecture of SimpLL after integrating our proposed changes.

For now, the extension is implemented as opt-in rather than opt-out, because it is a new and experimental feature of DiffKemp. Furthermore, running an SMT solver can, in the worst case, significantly increase the overall runtime of the analysis. The extension can be enabled through the Python front end using the `--use-smt` option. In order to reduce the risk of the analysis not terminating, we have also introduced an option to set timeout for SMT solving. This can be done using the `--smt-timeout` option. The default is set to 500 milliseconds, as this turned out to be sufficient to solve reasonably complex cases in our experiments (cf. Section 6.5). Note that this timeout is applied to each call of the SMT block comparator separately, but it is shared between all possible synchronization points that are found for one pair of differing instructions (cf. Section 4.1.1).

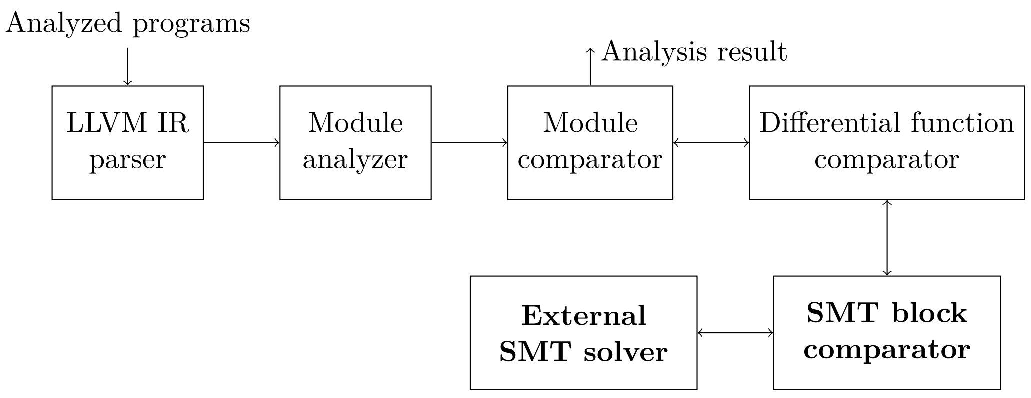


Figure 5.2: Architecture of SimpLL after integration of SMT solving

5.2.1 Choice of an SMT Solver

From the solvers described in Section 3.2, CVC5, Z3 and Bitwuzla seem to be the best suited for our integration thanks to their extensive support of various theories. We have opted against using Bitwuzla since it does not support theory of integer and real arithmetic. In the end, we have decided to use Z3 due to its good availability on most Linux distributions and a mature API. While CVC5 seems to slightly outperform Z3 based on the results of SMT-COMP, one of the limiting factors turned out to be its C++ API. It is not possible to limit the runtime of the solver when CVC5 is used as a library [10], which is a significant downside for integration into DiffKemp.

We have also tried checking if using CVC5 instead of Z3 has any effect on the overall results of our experiments. Thanks to the fact that all the aforementioned solvers support SMT-LIB [4, 28], we were able to export the SMT instance into the SMT-LIB format using the Z3 API and then pass it to CVC5 binary that was run in a different process (this allowed us to apply a timeout). The results of our experiments remained unchanged when using CVC5 instead of Z3. If more solvers emerge in the future, SMT-LIB could be nicely utilized to make use of each solver's strengths – DiffKemp could construct an SMT formula using, for example, the Z3 API, export it to SMT-LIB and run an array of solvers in parallel and stop whenever the first solver returns a result. We decided not to implement this feature for now, as it did not seem that it would provide any benefits.

Chapter 6

Results and Experiments

To evaluate the impact of our extension of DiffKemp, we have performed a series of experiments. Firstly, we checked our solution on a small set of simple hand-crafted examples in order to verify the basic functionality. This is described in Section 6.1. Section 6.2 details experiments with the EqBench benchmark that contains a collection of equivalent and non-equivalent program pairs. Section 6.3 evaluates the extension on two different implementations of the standard C library – diet libc and musl libc. Section 6.4 reproduces the Linux KABI experiment from [20] in order to evaluate the performance of our solution on large-scale projects and verify that our extension does not introduce any regression. Finally, Section 6.5 gives the runtime of Z3 on cases from the other sections where our extension was successful.

We performed other experiments with various C libraries – cryptographic (e.g., `mbedtls`, `sodium` and `nettle`), compression (e.g., `zlib`, `libzip` and `lz4`) and communication (e.g., `libusb` and `libcurl`) libraries. In these experiments, we ran DiffKemp with and without our extension on pairs of consecutive minor releases and compared functions that are part of the library's API. These experiments, however, did not yield any interesting result that would be worth including in this thesis. This is caused by the fact that our extension targets very specific types of refactoring that are not as common. And even if there was a case of such a refactoring, it may be accompanied by other changes that DiffKemp is not able to handle yet.

6.1 Simple Experiments

In order to verify basic functionality of our extension, we created a number of simple programs by hand. One of them, replacing multiplication with an equivalent bitwise shift and addition, can be seen in Figure 4.1. Figure 6.1 shows application of distributive law over unsigned integers that DiffKemp was able to analyze as semantically equal thanks to our extension.

Figure 6.2 shows another example which is similar to Figure 4.1, however, signed integers are used instead of unsigned integers. This means that the produced LLVM IR code contains `nsw` flags on all the operations, i.e., overflow results in a poison/undefined value being produced. Thanks to our encoding of the `nsw`/`nuw` flag described in Section 4.2, the example is analyzed as not equal.

```c
unsigned func(unsigned x) {
    unsigned y = x - 5;
    unsigned z = x + 5;
    unsigned res = z * x + z * y;
    return res;
}
```

```c
unsigned func(unsigned x) {
    unsigned y = x - 5;
    unsigned z = x + 5;
    unsigned res = z * (x + y);
    return res;
}
```

Figure 6.1: A semantically equivalent pair of programs which contains an application of distributive law over unsigned integers.

```c
int func(int x) {
    int y = x - 2;
    int z = y * 5;
    int a = z / 2;
    return a + 2;
}
```

```c
int func(int x) {
    int y = x - 2;
    int z = (y << 2) + y;
    int a = z / 2;
    return a + 2;
}
```

Figure 6.2: A program pair where multiplication has been replaced with bitwise shift and addition. The programs are not equal due to a possible overflow of signed integer arithmetic.

These cases, along with a few others, e.g., inspired by the EqBench benchmark (cf. Section 6.2), have been added to DiffKemp's suite of unit tests in order to ensure that our extension remains functional with newer versions of DiffKemp, LLVM, and Z3.

6.2 EqBench Benchmark

To evaluate our solution in more depth, we made use of a benchmark called EqBench. EqBench is a collection of equivalent and non-equivalent program pairs written in Java and C [2]. Some of the cases have been collected from existing benchmarks that were used to evaluate tools for checking semantic equivalence, such as Rêve [13] and Clever [22]. The authors extended these benchmarks with even more test cases, resulting in a collection of 147 equivalent and 125 non-equivalent pairs of programs [1]. A lot of the cases have been generated automatically by injecting changes of certain type into a piece of code [2]. A significant part of these changes are related to arithmetic expressions and their refactorings, which is where our extension should perform well. Therefore, the benchmark partly served as an inspiration for collecting use-cases for application of this work, e.g., the motivation example given in Section 4.3 is a real case taken from the benchmark. The main downside to these examples is that a lot of them use floating-point arithmetic, which SMT solvers do not handle very well.

The EqBench benchmark is structured as follows. The cases are divided into subdirectories corresponding to the original programs. Each subdirectory can have either an equivalent or a non-equivalent modification or both, these are present in the `Eq` and `Neq` directories, respectively. For example, the `REVE/mccarthy91/Eq` case contains an equivalent refactoring of the `REVE/mccarthy91` program (which comes from the Rêve benchmark in this particular case). Each of these `Eq`/`Neq` subdirectories contains the program pair to be analyzed (`oldV.c` and `newV.c` files) and a metadata file (`C-Desc.json`) describing the test case. The description contains the number of lines of the programs, the number of loops, and also a list of changes that were done to transform the old version of the program to the new version. A similar set of files is also present for the Java version, we, however, ignore these as DiffKemp focuses solely on C projects.

Due to the nature of DiffKemp's analysis algorithm, the effectiveness of the analysis inherently depends on how the LLVM IR is obtained during snapshot generation – how much optimization is applied, which optimization passes are run and what other transformations are applied to the analyzed functions to bring them syntactically closer to each other. For this reason, we evaluated our solution on a number of configurations. As for optimization level, we restricted ourselves to `-O1` and `-O2`. Using no optimization would not work well with our solution, since the clang compiler by default produces lots of unnecessary `load` and `store` instructions which we do not support and, therefore, we rely on the compiler's optimizer to remove the unnecessary memory manipulation. The optimization level used determines which optimization passes are run. Furthermore, we experimented with overriding them with passes that DiffKemp uses for building the Linux kernel (referred to as *custom* later in this text). At the time of writing, this includes the following passes[1] [31]:

- `lower-switch` rewrites the `switch` instruction to a sequence of `branch` instructions,
- `mem2reg` replaces unnecessary memory manipulation using registers,
- `loop-simplify` transforms loops into a simpler form,
- `simplifycfg` eliminates dead code and merges basic blocks,
- `gvn` performs global variable numbering to eliminate redundant instructions,
- `dce` eliminates dead code,
- `constmerge` merges duplicate global constants together into a single constant,
- `mergereturn` ensures that functions have at most one `ret` instruction inside them, i.e., they have a single exit.

There are 4 possible outcomes that can result from running DiffKemp on a case from the benchmark:

- DiffKemp reports equality of an equal program (*true negative*),
- DiffKemp reports non-equality of a non-equal program (*true positive*),
- DiffKemp reports non-equality of an equal program (*false positive*),
- DiffKemp reports equality of a non-equal program (*false negative*).

Table 6.1 shows DiffKemp results of the 3 configurations with SMT solving on and off. Since a lot of the programs contain floating-point arithmetic, which SMT solvers do not handle very efficiently, we extended the timeout to 30 seconds for this experiment. We can observe that the optimization passes that are used have quite a significant effect on the number of cases analyzed correctly. We can also see that the introduction of SMT solving did not introduce any unsoundness in the form of false negatives, and that it improved the number of equal cases that DiffKemp analyzed as equal.

The following cases have been newly analyzed as equal thanks to our extension:

[1]Taken from https://github.com/diffkemp/diffkemp/blob/master/diffkemp/llvm_ir/optimiser.py#L17-L25

Table 6.1: A comparison of DiffKemp with and without our extension across multiple configurations of snapshot generation

Optimization	`-O1`		`-O2`		*custom*	
SMT	Off	On	Off	On	Off	On
True negative	99	99	99	100	57	62
True positive	125	125	125	125	125	125
False positive	48	48	48	47	90	85
False negative	0	0	0	0	0	0

`REVE/mccarthy91` An example of advanced inverse branch condition. The source code can be seen in Figure 4.3. This example has been analyzed as equal in all the configurations, except `-O1` with default optimization passes. The problem in this configuration is that the compiler generated a `tail call` instruction along with a slightly different basic blocks where the value used for `icmp` comes from a Φ-node, i.e., it is not mapped through *varmap* due to DiffKemp's lazy analysis of Φ-nodes.

`bess/bessj0` Contains some dead code (that is eliminated by DiffKemp's code transformations) and also a condition modified from `ax < 8` to `-ax > -8` that an SMT solver managed to analyze as equivalent.

`bess/bessj1` Contains redundant multiplication of a variable by a float constant `1.0`.

`ell/rc` An expression has been extracted into a new variable, which resulted in a slightly different LLVM IR. In one of the versions, the compiler produced `%5 = or %3, %4`, while in the other version, it produced `%5 = select %3, true, %4`, which the SMT solver managed to check as equivalent.

`ran/gamdev` A condition has been modified similarly to `bess/bessj1`.

We can observe that a lot of these cases can be covered by a compiler's optimizer (and they indeed are if default optimization passes are used). While in this experiment, optimizations are helpful and provide better results, in other cases they may not be desirable. For example, some functions are inlined as a part of optimizations. Even though inlining may increase the precision of analysis, it also makes it more difficult to precisely locate the cause of the difference, which is an important feature of DiffKemp. Furthermore, optimizers are still subject to research and development, e.g., they may contain bugs and their capabilities can change between compiler versions. Therefore, our extension provides the advantage that DiffKemp does not have to rely on the compiler's optimizer as extensively.

While evaluating our work on the benchmark, we have identified one case that has an incorrect label, namely `ell/ellpi/Eq`, which refactors an arithmetic expression. The gist of the change is modifying an expression of the form $(a+b)\cdot(a-b)$ to a^2-b^2. However, the benchmark modified the expression to a^2+b^2 instead. SMT block comparator managed to provide us with a counterexample which resulted in different behavior in the two versions of the program. We have reported and fixed this problem in the GitHub repository of the benchmark, and our changes have already been merged[2].

Unfortunately, even after our fix, DiffKemp was not able to analyze this task correctly – the SMT block comparator identified a different behavior due to floating-point rounding

[2]https://github.com/shrBadihi/EqBench/pull/8

errors, which, however, the benchmark excludes through assertions that are present further down in the code, i.e., SMT block comparator does not take them into consideration. A similar problem occurred with multiple other test cases, where the programs were equivalent (e.g., used distributive laws) but the SMT block comparator was missing further context that excluded values that would lead to this different behavior. Encoding usage context of output variables that would help properly analyze such cases may be part of our future work.

Comparison with other tools Since a part of the benchmark was taken from other tools, namely RÊVE [13] and CLEVER [22], we can compare DIFFKEMP's performance with these tools. Note that the tools are based on formal methods, i.e., do not scale very well but are very precise. Therefore, we cannot expect DIFFKEMP to outperform these tools. For comparison, we consider DIFFKEMP's best performing configuration – `-O2` with default passes. Each row of Table 6.2 shows one part of the benchmark, the first two coming from the corresponding tools and the last one created by the authors of EQBENCH. The table shows the numbers of correctly and incorrectly analyzed equal programs, i.e., true negatives (TN) and false positives (FP), and the numbers of correctly and incorrectly analyzed non-equal programs, i.e., true positives (TP) and false negatives (FN)[3]. We can observe that despite DIFFKEMP's focus on scalability, it still performs reasonably well. Most notably, no false negatives were produced.

Table 6.2: Comparison of DIFFKEMP in its best configuration with other tools for checking semantic equivalence.

	Total programs		Tool results (TN/FP TP/FN)			
Benchmark	equal	non-equal	DIFFKEMP SMT Off	DIFFKEMP SMT On	RÊVE	CLEVER
RÊVE	23	9	6/17 9/0	7/16 9/0	20/0 8/0	
CLEVER	29	21	22/7 21/0	22/7 21/0	8/0 0/0	20/0 11/0
EQBENCH	95	95	71/24 95/0	71/24 95/0		

6.3 Standard C Libraries

In the world of software development, there are certain projects which need to maintain semantic stability throughout their releases. This includes various system libraries, e.g., the standard C library that almost all programs written in C make use of. There are multiple implementations of the standard C library, however, most Linux distributions these days include an implementation called GLIBC. While this implementation is complete wrt. its feature set, it has some disadvantages – most notably the library itself is large and so it is not very suitable for static linking. For this reason, various alternative implementations have been developed, e.g., the MUSL LIBC [25] and DIET LIBC [11] implementations have been developed with the goal of being light-weight and fast.

[3]The results of other tools have been taken from the papers written by their authors. For some programs, we were not able to find the result of the tool, hence the discrepancy between the number of cases between tools.

6.3.1 Comparing Minor Releases of musl libc

In this experiment, we compared 10 consecutive minor releases of the MUSL LIBC implementation with and without our extension. The snapshots were built using DIFFKEMP's standard configuration, i.e., `-O1` with custom optimization passes. Since the upstream version of MUSL LIBC cannot be directly built using the Clang compiler, we used a fork of the library called MUSLLVM [26]. Table 6.3 shows verdicts of the performed comparisons.

Table 6.3: Comparison of verdicts of DIFFKEMP analyzing minor releases of MUSL LIBC with and without our extension.

	Results: equal/non-equal	
Versions	**No-SMT**	**SMT**
1.1.10-1.1.11	1521/100	1521/100
1.1.11-1.1.12	1582/48	1582/48
1.1.12-1.1.13	1546/79	1546/79
1.1.13-1.1.14	1632/7	1632/7
1.1.14-1.1.15	1604/35	1604/35
1.1.15-1.1.16	1582/61	1582/61
1.1.16-1.1.17	**1531/109**	**1532/108**
1.1.17-1.1.18	1644/4	1644/4
1.1.18-1.1.19	1592/55	1592/55

We can observe that our extension managed to identify one more function as equal when comparing versions 1.1.16 and 1.1.17 – the `ftok` function. Figure 6.3 shows the difference in the function between these two versions. We can see that a constant was changed from a signed integer to an unsigned integer. The effect of this on the LLVM IR is that zero extension is used instead of signed extension before the `or` operator is applied. However, since the result is truncated back to 32 bits after the operation, the type of extension is irrelevant.

```
return ((st.st_ino & 0xffff) | ((st.st_dev & 0xff) << 16) | ((id & 0xff) << 24));
```

(a) C code in version 1.1.16

```
return ((st.st_ino & 0xffff) | ((st.st_dev & 0xff) << 16) | ((id & 0xffu) << 24));
```

(b) C code in version 1.1.17

```
%16 = sext i32 %15 to i64
%17 = or i64 %13, %16
%18 = trunc i64 %17 to i32
```

(c) LLVM IR in version 1.1.16

```
%16 = zext i32 %15 to i64
%17 = or i64 %13, %16
%18 = trunc i64 %17 to i32
```

(d) LLVM IR in version 1.1.17

Figure 6.3: Difference in the `ftok` function of the MUSL LIBC library between versions 1.1.16 and 1.1.17.

6.3.2 Comparing Semantics of diet libc and musl libc

When a C program uses functions from the standard C library, the developers should not have to worry about which implementation of the standard C library is used, i.e., the exported functions should be semantically equivalent between the various implementations. We compiled a list of 242 functions that can be considered a public interface of a standard C library based on [14] and used DiffKemp to compare the implementations of these functions from musl libc with diet libc.

```
int wcscmp(...)
{
      for (; *l==*r && *l && *r; l++, r++);
      return *l - *r;
}
```

(a) musl C source

```
int wcscmp(...) {
  while (*a && *a == *b)
    a++, b++;
  return (*a - *b);
}
```

(b) diet C source

```
1 %4 = load i32, ptr %.0, align 4
2 %5 = load i32, ptr %.01, align 4
3 %6 = icmp eq i32 %4, %5
4 %7 = icmp ne i32 %4, 0
5 %or.cond = and i1 %6, %7
6 br i1 %or.cond, label %8, label %9
```

(c) musl LLVM IR

```
1 %4 = load i32, ptr %.0, align 4
2 %5 = icmp ne i32 %4, 0
3 %.pre = load i32, ptr %.01, align 4
4 %6 = icmp eq i32 %4, %.pre
5 %or.cond = select i1 %5, i1 %6, i1 false
6 br i1 %or.cond, label %7, label %8
```

(d) diet LLVM IR

Figure 6.4: The C source code of the `wcscmp` function in diet libc and musl libc along with the differing parts of LLVM IR. Arguments of the functions have been omitted for brevity – both functions take two arguments of type `const wchar_t *`.

Even though comparing various implementations of a function from different projects is not the primary goal of DiffKemp, and such analysis would be more suited for tools based on formal methods, DiffKemp managed to analyze 10 functions as semantically equal. Our extension allowed DiffKemp to analyze one more function as equal – the `wcscmp` function. Figure 6.4 shows the C source code of the function along with the differing parts of the LLVM IR. The relocation pattern present in DiffKemp facilitated analysis of the different order of memory loads, while our extension managed to analyze the difference on lines 5 in Figure 6.4c and Figure 6.4d, i.e., the `and` instruction being replaced with a `select` instruction, as semantically equal.

6.4 Linux Kernel

In this section, we show that our extension does not cause any unwanted side effects when running on large-scale projects, and that it does not negatively impact DiffKemp's performance. To this end, we ran the compare phase on 10 pairs of consecutive minor releases of the Red Hat Enterprise Linux (RHEL) Kernel and tracked DiffKemp's runtime. In each pair of releases, only functions from the so-called *Kernel Application Binary Interface* (KABI) were compared. The KABI consists of functions whose semantics should be stable across the entire RHEL major release. While there are only several hundreds functions that are part of the KABI, DiffKemp still needs to compare thousands of functions and tens of thousands of lines of code due to nested function calls.

The experiments were run on a computer with AMD Ryzen 7 5700G 3.8GHz running Fedora 39. Table 6.4 shows the results of the KABI comparisons along with their runtime. We can see that our changes did not introduce any notable regression. Unfortunately,

Table 6.4: Comparison of verdicts and runtime of DiffKemp analyzing the RHEL KABI with and without our extension. The runtime for the version with SMT solving shows the runtime with the default 500-millisecond and with an extended 30-second timeout. The verdicts were the same regardless of the timeout.

	Results: eq/neq/unk		**Runtime: s**		
Versions	**No-SMT**	**SMT**	**No-SMT**	**SMT 0.5s**	**SMT 30s**
7.3-7.4	408/264/66	408/264/66	342	392	394
7.4-7.5	550/178/66	550/178/66	310	312	313
7.5-7.6	609/124/66	609/124/66	193	209	208
7.6-7.7	642/120/72	642/120/72	224	231	230
7.7-7.8	613/176/74	613/176/74	176	180	179
8.0-8.1	362/84/75	362/84/75	204	201	201
8.1-8.2	334/161/78	334/161/78	194	204	202
8.2-8.3	422/178/87	422/178/87	450	403	404
8.3-8.4	450/153/88	450/153/88	390	402	400
8.4-8.5	442/170/88	442/170/88	374	392	394

there were no cases of a false non-equal verdict that our extension was able to resolve. Furthermore, the runtime was longer by only 7 seconds on average. This fact highlights that we only try to apply SMT solving in cases which seem promising. On the other hand, we can see that the runtime is very similar even if the timeout is extended to 30 seconds. This suggests that most of the overhead of our extension comes from searching for a synchronization point (e.g., backing up *varmap* can be quite expensive) rather than from SMT solving itself. As a part of our future work, we would like to further try reducing this overhead.

An interesting fact can be seen in the comparison of version 8.2 with version 8.3 – the runtime got shorter when SMT solving was turned on. We attribute this unexpected change to the way how DiffKemp caches results of comparisons of called functions – searching for a synchronization point may lead to analysis of some functions earlier than they would have been analyzed without SMT solving, which can possibly remove the need for performing some inlining.

```
// reg, mask, val are unsigned int
if (reg & 0xff000000) {
   unsigned char size, offset;
   size = (reg >> 24) & 0x3f;
   offset = (reg >> 16) & 0x1f;

   mask = ((1 << size) - 1) << offset;
   return (val & mask) >> offset;
} else {
   return val;
}
```

(a) Before the commit

```
// reg, mask, val are unsigned int
if (reg & 0xff000000) {
   unsigned char size, offset;
   size = (reg >> 24) & 0x3f;
   offset = (reg >> 16) & 0x1f;

   mask = (1 << size) - 1;
   return (val >> offset) & mask;
} else {
   return val;
}
```

(b) After the commit

Figure 6.5: The change made to the `snd_emu10k1_ptr_read` function in commit `2e9bd50`. Unnecessary bitwise shifts have been removed to optimize the code.

Optimizing and Refactoring Commits Even though our extension did not seem to be efficient when applied to the KABI, we conducted another experiment with the Linux kernel. We collected git commits from the upstream Linux kernel repository that contain the words `optimize` or `refactor` in the commit title, and tried checking semantic equivalence of such commits using our extension. Out of the more than 10000 commits that match this criterion, we manually investigated roughly 500 of them. A lot of them either contained very complex refactorings, which are too complex for DiffKemp, or in a lot of cases were not semantically equal according to the definition from Section 2.2, e.g., some optimizations removed unnecessary mutex locking. We managed to identify a commit that DiffKemp managed to analyze as equivalent only thanks to our extension – commit `2e9bd50`. Figure 6.5 shows the content of the commit, unnecessary bitwise shifts back and forth in order to apply a mask have been removed to reduce the number of instructions necessary to perform the operation.

6.5 Evaluating Z3 Runtime on Successful Cases

To round out the description of our experiments, we give an overview of how long SMT solving using Z3 took on cases from the previous sections where our extension facilitated correct analysis. Table 6.5 gives Z3 runtime in milliseconds for each successful case described in the previous sections. The results show that even though SMT solving is a very hard computational problem, our experiments usually took only a few milliseconds thanks to focusing only on short code blocks, i.e., the formulae checked for satisfiability were quite simple. The only exception to this is the `bess/bessj1` case from EqBench which highlights how inefficient SMT solvers can be when analyzing floating-point arithmetic – despite performing only a trivial operation (multiplication by one), the analysis took significantly longer than more complex analyses in the theory of bit vectors. These results highlight that most of the time, we can keep the timeout quite low, e.g., 500 milliseconds is sufficient, however when floating-point operations are present, the timeout should be extended by the users of DiffKemp.

Table 6.5: Z3 runtime on cases where our extension facilitated correct analysis

Section	**Case**	**Z3 Runtime milliseconds**
6.1	Simplification using multiplication (Figure 4.1)	2
	Distributive law over unsigned (Figure 6.1)	2
6.2	`REVE/mccarthy91`	2
	`bess/bessj0`	8
	`bess/bessj1`	17 526
	`ell/rc`	2
	`ran/gamdev`	7
6.3	`ftok` change in musl libc (Figure 6.3)	2
	`wcscmp` in diet libc and musl libc (Figure 6.4)	2
6.4	Linux kernel optimizing commit (Figure 6.5)	26

Chapter 7

Related Work

As already pointed out in the previous chapters, there exists a number of approaches and tools for tackling static analysis of semantic differences. In this chapter, we give a brief overview of them and make a comparison with DiffKemp and the extension proposed in this work. For a more complete overview, refer to [17].

Most state-of-the-art tools for differential static analysis employ some sort of formal methods to perform the analysis, e.g., there is a group of tools that encode equivalence of functions using formulae and then use decision procedures, such as SMT solvers, or program verifiers to prove equality. For example, SymDiff [16] uses Z3 to check equivalence. Similarly, Clever [22] makes use of symbolic execution combined with SMT solving using Z3. Rêve [13] and its successor LLRêve [15] translate the programs into Horn clauses and pass them to a Horn solver, e.g., Z3 or Eldarica. On the other hand, UC-KLEE [29] uses the KLEE verifier as its back end. While these tools are very precise and rarely produce false positives, their analysis is very costly and does not scale well – they can typically handle only tens of lines of code.

On the other side of the spectrum, there exist extremely fast light-weight tools based on text similarity, e.g., the `diff` tool, or on comparison of abstract syntax trees. Such tools can handle lots of code, up to millions of lines of code, in a matter of seconds or minutes, however, even the simplest changes typically result in false alarms.

DiffKemp's approach tries to find a middle ground between these two extremes. While it is not as fast as text-based comparison tools, it still performs reasonably well on large codebases, e.g., the Linux kernel, and it can identify much more complicated changes as being semantically equal. On the other hand, when compared to the tools based on formal methods, DiffKemp is not able to show equality of more complex refactorings, especially when the control flow of the program is changed. These claims can be evidenced in our experiments in Chapter 6. For example, the KABI experiment in Section 6.4 shows that DiffKemp is able to analyze tens of thousands of lines of code in a matter of minutes. On the other hand, the experiments with EqBench in Section 6.2 highlighted that while DiffKemp can handle most common semantics-preserving changes, there are still a lot more that are too complex and can be analyzed better for example by Rêve. We could also observe that while tools based on formal methods usually time out in case of an overly complicated refactoring, i.e., the result is unknown, DiffKemp will report a false positive if it does not have a defined pattern for the change.

The extension proposed in this work brings some elements from the tools based on formal methods to DiffKemp's analysis algorithm. Instead of entire functions, only the equivalence of small blocks of code is checked using an SMT solver. The main advantage of this is clearly performance, since the formulae for the SMT solver are simpler, and they are built only from promising blocks of code. The downside is that since we limit ourselves to sequential blocks, the formulae are missing context, e.g., about what comes after the analyzed block or about the origin of the values used in the current block. Due to this fact, if an equivalent refactoring is scattered across multiple basic blocks, we are not able to analyze it as semantically equal. Our experiments showed that the proposed integration of SMT solving into the analysis core of DiffKemp improves analysis capabilities of the tool and brings its precision closer to tools based on formal methods without negatively impacting the tool's performance.

Chapter 8

Conclusion

In this work, we proposed an integration of formal methods into the DiffKemp static analysis tool for checking semantic equivalence of large-scale C programs. When a potential semantic difference is found and none of DiffKemp's built-in semantics-preserving patterns are applicable, our extension tries to find a pair of sequential blocks of code that may be causing the difference. It then encodes the equivalence of the blocks into a first-order logic formula and uses an SMT solver to check whether the blocks are equal, i.e., DiffKemp can continue its analysis, or non-equal.

The proposed solution has been implemented in DiffKemp. We performed several experiments with the implementation on projects of various sizes, ranging from simple hand-made examples, through various system libraries, to analysis of the RHEL Kernel Application Binary Interface. Our experiments with the EqBench benchmark show that the new extension extends analysis capabilities of DiffKemp, i.e., it can analyze programs as equal that the tool was not able to before. Furthermore, we were able to identify a wrong test case in the EqBench benchmark using our integration, report it to the authors and cooperate on fixing the issue. The experiments with projects of larger scale show that while SMT solving is an NP-hard problem, our extension only tries to apply it to promising blocks of code, hence there is no significant regression in performance.

Checking semantic equality soundly and precisely is a difficult problem. There are some tools based on formal methods that are able to achieve very high precision but can only analyze small programs in a reasonable time. On the other hand, there are tools based on syntactic and textual comparison that are very fast but also very imprecise. DiffKemp tries to find a middle ground between these two approaches – its goal is to provide a highly scalable, yet reasonably precise analysis. Our solution brings DiffKemp a bit closer to tools based on formal methods with regard to precision, while not negatively impacting the performance.

In future, our integration could be extended with support for more complex programming constructs – e.g., branching and memory manipulation. Adding support for branching would enable encoding more context into the formulae about the way how output variables are used. Furthermore, there are most likely other places in DiffKemp where SMT solver could be employed, e.g., in the custom pattern comparator component. Lastly, more experiments with various SMT solvers could be performed, and it may be beneficial to run multiple solvers on the same problem in parallel, to utilize each solver's strengths.

Bibliography

[1] Badihi, S. *EqBench.* Online. 2024. Available at: https://github.com/shrBadihi/EqBench/tree/main. [cit. 2024-09-04].

[2] Badihi, S.; Li, Y. and Rubin, J. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In: IEEE/ACM. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* 2021, p. 610–614. ISBN 978-1-7281-8710-5.

[3] Barbosa, H.; Barrett, C.; Brain, M.; Kremer, G.; Lachnitt, H. et al. Cvc5: A Versatile and Industrial-Strength SMT Solver. In: Fisman, D. and Rosu, G., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Cham: Springer International Publishing, 2022, p. 415–442. ISBN 978-3-030-99524-9.

[4] Barrett, C.; Fontaine, P. and Tinelli, C. *The Satisfiability Modulo Theories Library (SMT-LIB) – SMT Solvers.* Online. 2016. Available at: https://smt-lib.org/solvers.shtml. [cit. 2024-23-04].

[5] Barrett, C.; Moura, L. de and Stump, A. SMT-COMP: Satisfiability Modulo Theories Competition. In: Etessami, K. and Rajamani, S. K., ed. *Computer Aided Verification.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, p. 20–23. ISBN 978-3-540-31686-2.

[6] Barrett, C.; Stump, A.; Tinelli, C. et al. The SMT-LIB Standard: Version 2.0. In: Gupta, A. and Kroening, D., ed. *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK).* 2010, vol. 13, p. 14.

[7] Barrett, C. and Tinelli, C. Satisfiability Modulo Theories. In: Clarke, E. M.; Henzinger, T. A.; Veith, H. and Bloem, R., ed. *Handbook of Model Checking.* Cham: Springer International Publishing, 2018, p. 305–343. ISBN 978-3-319-10575-8. Available at: https://doi.org/10.1007/978-3-319-10575-8_11.

[8] Bruttomesso, R.; Pek, E.; Sharygina, N. and Tsitovich, A. The OpenSMT Solver. In: Esparza, J. and Majumdar, R., ed. *Tools and Algorithms for the Construction and Analysis of Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, p. 150–153. ISBN 978-3-642-12002-2.

[9] Cook, S. A. The complexity of theorem-proving procedures. In: ACM. *Proceedings of the Third Annual ACM Symposium on Theory of Computing.* New York, NY, USA: Association for Computing Machinery, 1971, p. 151–158. STOC '71. ISBN 9781450374644. Available at: https://doi.org/10.1145/800157.805047.

[10] cvc5. *Overall time limit*. Online. 2024. Available at: https://cvc5.github.io/docs/cvc5-1.1.2/resource-limits.html#overall-time-limit-tlimit-option. [cit. 2024-09-04].

[11] *Diet libc - a libc optimized for small size*. Online. 2024. Available at: https://www.fefe.de/dietlibc/. [cit. 2024-23-04].

[12] Dutertre, B. Yices 2.2. In: Biere, A. and Bloem, R., ed. *Computer Aided Verification*. Cham: Springer International Publishing, 2014, p. 737–744. ISBN 978-3-319-08867-9.

[13] Felsing, D.; Grebing, S.; Klebanov, V.; Rümmer, P. and Ulbrich, M. Automating regression verification. In: ACM/IEEE. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, p. 349–360. ASE '14. ISBN 9781450330138. Available at: https://doi.org/10.1145/2642937.2642987.

[14] IBM Corporation. *Standard C Library Functions Table, By Name*. Online. 2023. Available at: https://www.ibm.com/docs/en/i/7.5?topic=extensions-standard-c-library-functions-table-by-name. [cit. 2024-23-04].

[15] Kiefer, M.; Klebanov, V. and Ulbrich, M. Relational Program Reasoning Using Compiler IR. In: Blazy, S. and Chechik, M., ed. *Verified Software. Theories, Tools, and Experiments*. Cham: Springer International Publishing, 2016, p. 149–165. ISBN 978-3-319-48869-1.

[16] Lahiri, S. K.; Hawblitzel, C.; Kawaguchi, M. and Rebêlo, H. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In: Madhusudan, P. and Seshia, S. A., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, p. 712–717. ISBN 978-3-642-31424-7.

[17] Lahiri, S. K.; Vaswani, K. and Hoare, C. A. R. Differential static analysis: opportunities, applications, and challenges. In: ACM. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. New York, NY, USA: Association for Computing Machinery, 2010, p. 201–204. FoSER '10. ISBN 9781450304276. Available at: https://doi.org/10.1145/1882362.1882405.

[18] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In: IEEE. *CGO*. San Jose, CA, USA: [b.n.], March 2004, p. 75–88. ISBN 0-7695-2102-9.

[19] Malík, V.; Šilling, P. and Vojnar, T. Applying Custom Patterns in Semantic Equality Analysis. In: Koulali, M.-A. and Mezini, M., ed. *Networked Systems*. Cham: Springer International Publishing, 2022, p. 265–282. ISBN 978-3-031-17436-0.

[20] Malík, V. and Vojnar, T. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In: IEEE. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2021, p. 329–339. ISBN 978-1-7281-6836-4.

[21] Mann, M.; Wilson, A.; Zohar, Y.; Stuntz, L.; Irfan, A. et al. Smt-Switch: A Solver-Agnostic C++ API for SMT Solving. In: Li, C.-M. and Manyà, F.,

ed. *Theory and Applications of Satisfiability Testing – SAT 2021*. Cham: Springer International Publishing, 2021, p. 377–386. ISBN 978-3-030-80223-3.

[22] Mora, F.; Li, Y.; Rubin, J. and Chechik, M. Client-specific equivalence checking. In: IEEE/ACM. *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 441–451. ASE '18. ISBN 9781450359375. Available at: https://doi.org/10.1145/3238147.3238178.

[23] Moura, L. de and Bjørner, N. Z3: An Efficient SMT Solver. In: Ramakrishnan, C. R. and Rehof, J., ed. *Tools and Algorithms for the Construction and Analysis of Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 337–340. ISBN 978-3-540-78800-3.

[24] Moura, L. de and Passmore, G. O. Computation in Real Closed Infinitesimal and Transcendental Extensions of the Rationals. In: Bonacina, M. P., ed. *Automated Deduction – CADE-24*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, p. 178–192. ISBN 978-3-642-38574-2.

[25] *Musl libc*. Online. 2024. Available at: https://musl.libc.org/. [cit. 2024-23-04].

[26] *Musllvm*. Online. 2024. Available at: https://github.com/SRI-CSL/musllvm. [cit. 2024-23-04].

[27] Niemetz, A. and Preiner, M. Bitwuzla. In: Enea, C. and Lal, A., ed. *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*. Springer, 2023, vol. 13965, p. 3–17. Lecture Notes in Computer Science. ISBN 978-3-031-37703-7. Available at: https://doi.org/10.1007/978-3-031-37703-7_1.

[28] Niemetz, A. and Preiner, M. *Bitwuzla – Command Line Interface*. Online. 2023. Available at: https://bitwuzla.github.io/docs/binary.html. [cit. 2024-23-04].

[29] Ramos, D. A. and Engler, D. R. Practical, Low-Effort Equivalence Verification of Real Code. In: Gopalakrishnan, G. and Qadeer, S., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, p. 669–685. ISBN 978-3-642-22110-1.

[30] The LLVM Project. *LLVM Language Reference Manual*. Online. 2024. Available at: https://llvm.org/docs/LangRef.html. [cit. 2024-07-04].

[31] The LLVM Project. *LLVM's Analysis and Transform Passes*. Online. 2024. Available at: https://llvm.org/docs/Passes.html. [cit. 2024-26-04].

[32] Žáčik, P. *Analyzing semantic stability of cryptography libraries using Diffkemp*. Online. 2024 [cit. 2024-04-03]. Master's thesis. Masaryk University, Faculty of Informatics, Brno. Available at: https://is.muni.cz/th/ponkv/. SUPERVISOR : Martin Ukrop.

Appendix A

Contents of the Included Storage Media

The included storage media contains the source files of DiffKemp with our extension, as well as the source files of the thesis. The structure of the root directory is the following:

```
/
├── diffkemp/ ........................................ DiffKemp directory
│   ├── diffkemp/ .............................. Source files of DiffKemp
│   │   └── simpll/ ............................. Source files of SimpLL
│   ├── tests/ ................ Regression and unit tests of DiffKemp
│   └── README.md ............... DiffKemp compilation instructions
├── doc/ ................................ LaTeX source files of this text
├── experiments/ ........... Source files of experiments from Section 6.1
└── README ...................................................... README file
```

The /diffkemp/diffkemp directory contains the implementation of DiffKemp including our extension. Most of the implementation described in this work was done in the SimpLL library, i.e., in the /diffkemp/diffkemp/simpll/ directory, most notably in the SMTBlockComparator.cpp and SMTBlockComparator.h files. Some other minor changes were done in the DifferenticalFunctionComparator.cpp source file, as well as the source code for the Python front end, i.e., cli.py and config.py. The directory /doc contains the source LaTeX files of this text as well as the PDF version of this text, and the accompanying poster. Finally, the /experiments directory contains C source files of our simple experiments described in Section 6.1 that can be used for validating the basic functionality of our extension.

Appendix B

Compilation and Running

The project can be compiled and run using the source files present on the included storage media. Compilation instructions can be found in the `/diffkemp/README.md` file, this appendix aims to give a summary of the process. The following dependencies need to be installed before starting the compilation process:

- Clang and LLVM (supported versions are 9, 10, 11, 12, 13, 14, 15, 16),
- Python 3,
- the CMake and Ninja build systems,
- Python packages from `/diffkemp/requirements.txt` (can be installed using `pip install -r requirements.txt`), as well as the Python CFFI package,
- the `cscope` utility,
- Z3 SMT solver, along with its headers (e.g., the `z3-devel` package on Fedora),
- `gtest` for running the C++ tests.

Alternatively, the `nix` package manager[1] can be used to simplify the setup. See the DiffKemp's README for more information on this matter. The build can be created using the following commands in the `/diffkemp` directory:

```
mkdir build
cd build
cmake .. -GNinja
ninja
cd ..
pip install -e .
```

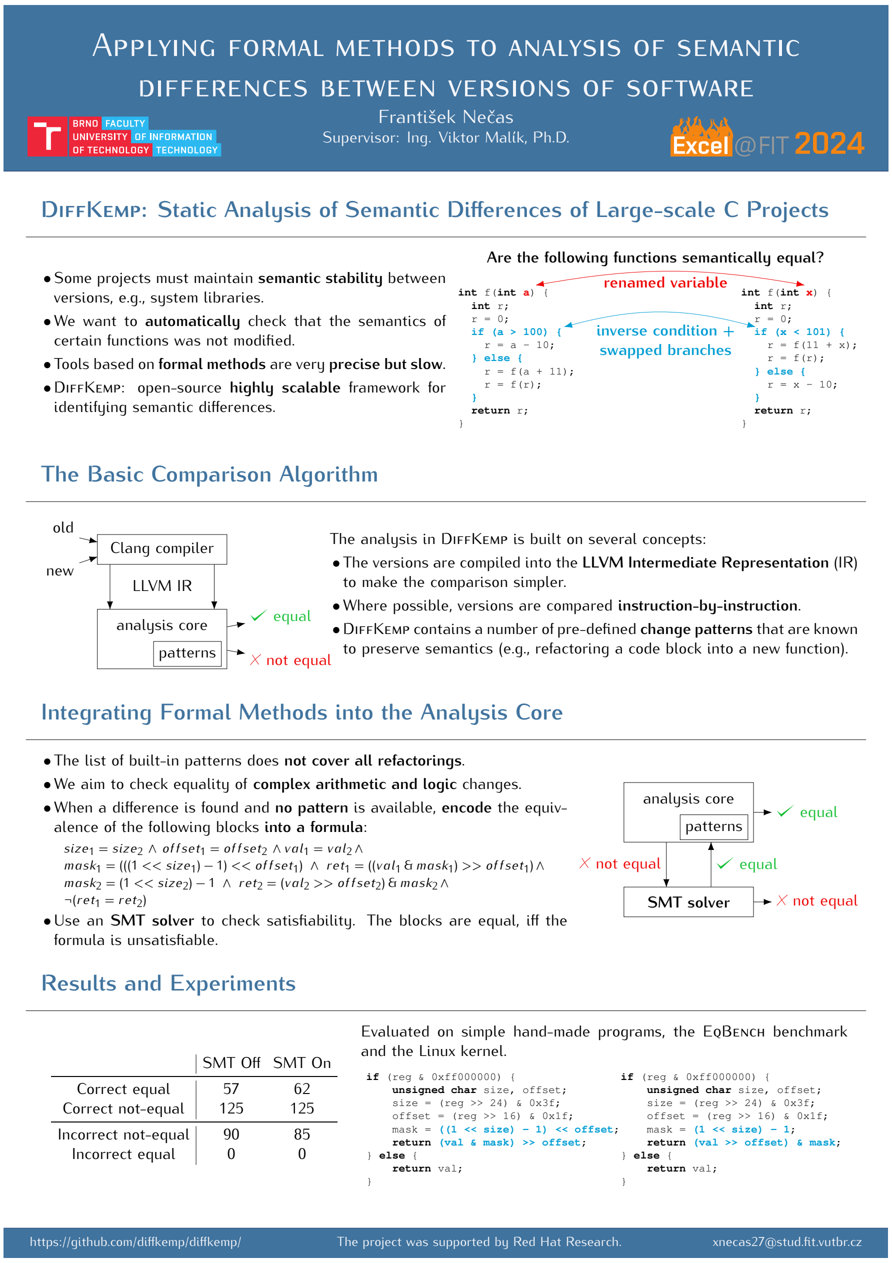
The DiffKemp binary is then located in `bin/diffkemp` (shortened to `diffkemp` in the rest of this text). The C++ unit tests can be run by using `ninja test` command from the `build` directory. In order to execute the regression tests, one needs to have several kernel versions (refer to the README for the exact versions required) downloaded and then run `pytest tests`.

[1]https://nixos.org/download/

Before running our extension, snapshots need to be generated either by using the `diffkemp build` or `diffkemp build-kernel` commands. For instance, the example from Figure 6.1 can be built by going to the `/experiments/distributive` directory and running `diffkemp build old.c old` and `diffkemp build new.c new`. Finally, the comparison phase can be run on the generated snapshots by using `diffkemp compare old new --report-stat --use-smt`. The Z3 timeout can be extended by using the `--smt-timeout` option that accepts the timeout in milliseconds. More information on running the tool can be found either in the README or in the program's help message.

Appendix C

Poster



Applying formal methods to analysis of semantic differences between versions of software

František Nečas
Supervisor: Ing. Viktor Malík, Ph.D.
Brno University of Technology, Faculty of Information Technology
Excel@FIT 2024

DiffKemp: Static Analysis of Semantic Differences of Large-scale C Projects

- Some projects must maintain **semantic stability** between versions, e.g., system libraries.
- We want to **automatically** check that the semantics of certain functions was not modified.
- Tools based on **formal methods** are very **precise but slow**.
- DiffKemp: open-source **highly scalable** framework for identifying semantic differences.

Are the following functions semantically equal?

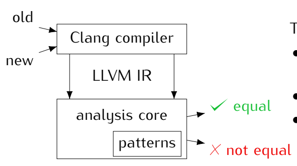
renamed variable

inverse condition + swapped branches

```
int f(int a) {
  int r;
  r = 0;
  if (a > 100) {
    r = a - 10;
  } else {
    r = f(a + 11);
    r = f(r);
  }
  return r;
}
```

```
int f(int x) {
  int r;
  r = 0;
  if (x < 101) {
    r = f(11 + x);
    r = f(r);
  } else {
    r = x - 10;
  }
  return r;
}
```

The Basic Comparison Algorithm



The analysis in DiffKemp is built on several concepts:

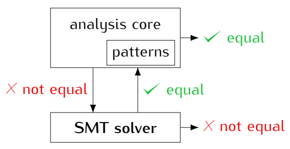
- The versions are compiled into the **LLVM Intermediate Representation** (IR) to make the comparison simpler.
- Where possible, versions are compared **instruction-by-instruction**.
- DiffKemp contains a number of pre-defined **change patterns** that are known to preserve semantics (e.g., refactoring a code block into a new function).

Integrating Formal Methods into the Analysis Core

- The list of built-in patterns does **not cover all refactorings**.
- We aim to check equality of **complex arithmetic and logic** changes.
- When a difference is found and **no pattern** is available, **encode** the equivalence of the following blocks **into a formula**:

$$size_1 = size_2 \wedge offset_1 = offset_2 \wedge val_1 = val_2 \wedge$$
$$mask_1 = (((1 << size_1) - 1) << offset_1) \ \wedge \ ret_1 = ((val_1 \,\&\, mask_1) >> offset_1) \wedge$$
$$mask_2 = (1 << size_2) - 1 \ \wedge \ ret_2 = (val_2 >> offset_2) \,\&\, mask_2 \wedge$$
$$\neg(ret_1 = ret_2)$$

- Use an **SMT solver** to check satisfiability. The blocks are equal, iff the formula is unsatisfiable.



Results and Experiments

	SMT Off	SMT On
Correct equal	57	62
Correct not-equal	125	125
Incorrect not-equal	90	85
Incorrect equal	0	0

Evaluated on simple hand-made programs, the EqBench benchmark and the Linux kernel.

```
if (reg & 0xff000000) {
    unsigned char size, offset;
    size = (reg >> 24) & 0x3f;
    offset = (reg >> 16) & 0x1f;
    mask = ((1 << size) - 1) << offset;
    return (val & mask) >> offset;
} else {
    return val;
}
```

```
if (reg & 0xff000000) {
    unsigned char size, offset;
    size = (reg >> 24) & 0x3f;
    offset = (reg >> 16) & 0x1f;
    mask = (1 << size) - 1;
    return (val >> offset) & mask;
} else {
    return val;
}
```

https://github.com/diffkemp/diffkemp/ The project was supported by Red Hat Research. xnecas27@stud.fit.vutbr.cz