BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

APPLYING CODE CHANGE PATTERNS DURING ANALYSIS OF PROGRAM EQUIVALENCE

POUŽITÍ ŠABLON ZMĚN KÓDU POČAS ANALÝZY EKVIVALENCE PROGRAMŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **PETR ŠILLING**
AUTOR PRÁCE

SUPERVISOR — **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

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

Bachelor's Thesis Specification

Student: **Šilling Petr**
Programme: Information Technology
Title: **Applying Code Change Patterns during Analysis of Program Equivalence**
Category: Software analysis and testing

Assignment:

1. Study existing works on identification and analysis of software refactoring patterns. Concentrate on identifying typical patterns of changes occurring in low-level production code, especially the GNU/Linux kernel.
2. Get acquainted with DiffKemp, a tool for automatic analysis of semantic equivalence of functions in the GNU/Linux kernel.
3. Propose an encoding of low-level code change patterns, using the intermediate representation of the Clang/LLVM complier.
4. Design an extension of DiffKemp that would allow it to accept custom semantics-preserving change patterns (using the proposed encoding) and to compare code matching the given patterns as semantically equal.
5. Implement the proposed extension in the DiffKemp framework.
6. Evaluate the implemented solution on at least 3 pairs of past versions of the Linux kernel, using a set of at least 10 custom patterns that commonly occurred in these versions.

Recommended literature:

- Garrido, Alejandra. *Software refactoring applied to C programming language*. PhD thesis. University of Illinois at Urbana-Champaign, 2000.
- Official website of DiffKemp: https://github.com/viktormalik/diffkemp
- Ullmann, Julian R. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23.1 (1976): 31-42.

Requirements for the first semester:

- The first 3 items of the assignment.

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

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

Abstract

The goal of this thesis is to propose a static analysis method for recognition of code change patterns describing recurrent changes between different versions of low-level code. The thesis proposes an encoding method of patterns, which uses the LLVM intermediate representation, and a pattern matching algorithm based on gradual comparison of instructions according to their control flow. The proposed analysis has been implemented as an extension of DiffKemp, a tool for analysing semantic differences between versions of large C projects. Results of experiments conducted on three pairs of past versions of the Linux kernel show that the extension is able to eliminate a substantial amount of false-positive or generally undesirable differences from the output of DiffKemp, which would otherwise require manual inspection.

Abstrakt

Cílem této práce je návrh statické analýzy pro rozpoznávání vzorů, popisujících často se vyskytující změny mezi různými verzemi nízkoúrovňového kódu. V rámci práce je navržen způsob kódování vzorů, využívající vnitřní reprezentaci LLVM, a algoritmus pro hledání vzorů založený na postupném porovnávání instrukcí podle toku řízení. Navržená analýza byla implementována jako rozšíření nástroje DiffKemp pro analýzu sémantických rozdílů různých verzí rozsáhlých projektů napsaných v jazyce C. Výsledky experimentů provedených na třech dvojicích minulých verzí linuxového jádra ukazují, že navržené rozšíření dokáže eliminovat podstatné množství falešně pozitivních či obecně nežádoucích rozdílů z výsledků porovnání nástrojem DiffKemp, které by jinak vyžadovaly manuální kontrolu.

Keywords

DiffKemp, SimpLL, LLVM, Clang, GNU/Linux kernel, code change pattern matching, code change pattern representation, subgraph isomorphism, semantic difference analysis, refactoring patterns, LLVM metadata, LLVM intermediate representation, control-flow graph, elimination of false-positive reports

Klíčová slova

DiffKemp, SimpLL, LLVM, Clang, GNU/Linux kernel, porovnávání vzorů změn kódu, reprezentace vzorů změn kódu, izomorfismus podgrafů, analýza sémantických rozdílů mezi programy, refaktorovací vzory, LLVM metadata, vnitřní reprezentace LLVM, graf toku řízení, odstraňování falešně pozitivních hlášení

Reference

ŠILLING, Petr. *Applying Code Change Patterns during Analysis of Program Equivalence.* Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšířený abstrakt

Při úpravách částí programu, které by v ideálním případě měly zůstat dlouhodobě stabilní (například po celou dobu života majoritní verze daného programu), může být zcela zásadní mít informaci o tom, které další části programu budou změnami ovlivněny a jaký bude dopad na sémantiku celého programu. Aby si vývojáři zjednodušili proces hledání těchto zpravidla nežádoucích vedlejších účinků prováděných změn, mohou teoreticky využít automatizované *statické analyzátory sémantické ekvivalence*, tedy nástroje, které dokáží porovnat programy, případně pak různé verze stejného programu, a najít mezi nimi rozdíly v sémantice. Aktuálně dostupné analyzátory sémantické ekvivalence jsou ovšem zpravidla založeny na vysoce výpočetně náročných formálních metodách. Vývojáři pracující na rozsáhlejších projektech se proto i nadále musí téměř zcela spoléhat na manuální analýzu sémantických rozdílů a vedlejších účinků, což je ovšem proces, který je obzvláště časově náročný a náchylný na lidské chyby.

Pomalu se však objevují i analyzátory, které se snaží o co největší škálovatelnost, a tedy i použitelnost na rozsáhlé projekty. Jedním z takových analyzátorů je i DiffKemp, nástroj pro analýzu sémantických rozdílů mezi programy napsanými v jazyce C, který se, vzhledem k tomu že je vyvíjen v rámci firmy Red Hat, zaměřuje zejména na linuxové jádro. DiffKemp využívá vysoce škálovatelnou techniku pro hledání sémantických rozdílů, která obvykle produkuje jen velmi malý počet falešných hlášení. Obecně lze tedy říci, že se DiffKemp principiálně nachází přímo mezi formálními metodami, které jsou zcela přesné, ovšem rovněž velmi výpočetně náročné, a jednoduchými, výpočetně nenáročnými metodami, obvykle z hlavní části založenými na prostém porovnávání textu.

Aby mohl DiffKemp dosáhnout požadované přesnosti a efektivity, převádí oba porovnávané programy z jazyka C do vnitřní reprezentace LLVM (LLVM IR) a následně se je pokouší porovnat po jednotlivých instrukcích. Pokud však oba programy nejsou syntakticky totožné, samotné porovnávání instrukcí generuje značný počet falešně pozitivních hlášení o sémantických rozdílech. Aby se jejich množství snížilo na co nejnižší úroveň, DiffKemp před samotným porovnáváním aplikuje několik sémantiku zachovávajících transformací kódu analyzovaných programů, aby je k sobě co nejvíce syntakticky přiblížil, a rovněž v obou programech hledá zabudované vzory změn kódu, o nichž je známo, že zachovávají sémantiku. Tyto *sémantiku zachovávající vzory změn kódu* jsou obzvláště důležité, protože s jejich pomocí má DiffKemp možnost korektně porovnat i poměrně složité refaktorovací změny. Protože však DiffKemp podporuje pouze staticky zabudované vzory, nemá možnost dodatečně reagovat na rozdílné potřeby odlišných vývojářů a zároveň nedokáže zajistit případnou podporu změn, které by ovlivňovaly sémantiku (a to ani pokud by byly dané změny sémantiky úmyslné a předem ověřené).

S ohledem na výše uvedené nedostatky pevně zabudovaných sémantiku zachovávající vzorů změn kódu tato práce navrhuje a implementuje rozšíření nástroje DiffKemp, které dává uživatelům možnost DiffKemp dynamicky rozšířit o vlastní vzory změň kódu, a tím i určit, jaké další změny by měl DiffKemp považovat za sémanticky ekvivalentní. Protože jde navíc o uživatelsky definované vzory, nemusí již ani nutně respektovat zachování sémantiky (respektive popisovat refaktorování). Naopak může v některých případech jít i o vzory popisující změny v sémantice, které jsou předem ověřené jako bezpečné (například nutné bezpečnostní opravy).

Konkrétněji práce zkoumá existující vzory změn kódu vyskytující se v nízkoúrovňových projektech napsaných v jazyce C, jako je například linuxové jádro, a připravuje jejich kódování založené na LLVM IR tak, aby měl DiffKemp možnost vzory dynamicky načítat. Vzhledem k druhům vzorů změn kódu identifikovaných v rámci analýzy různých verzí lin-

uxového jádra práce přímo navrhuje dvě reprezentace vzodů změn kódu, jednu univerzální a druhou specializovanou pro vzory popisující velmi jednoduhé, jednohodnotové změny. Oba druhy reprezentace jsou kódovány v LLVM IR, a sice pomocí dvou funkcí s pevně danou strukturou.

Práce dále navrhuje metodu detekce vzorů změn kódu v porovnávaných programech, která je založena na problému hledání izomorfních pografů větších grafů a využívá infrastrukturu LLVM, zejména fakt, že funkce jsou v LLVM reprezentovány jako grafy toku řízení. Přesněji řečeno, metoda se pro každý vzor pokouší najít takové podgrafy grafů toku řízení porovnávaných programů, které jsou *izomorfní* s grafy toku řízení příslušejícími danému vzoru. Podgrafy se metoda pokouší nalézt za pomoci postupného porovnávání instrukcí. Metoda byla implementována v jazyce C++ jako rozšíření nástroje DiffKemp.

Výsledné rozšíření bylo experimentálně ověřeno na třech párech předchozích verzí linuxového jádra z hlediska jeho dopadu na analýzu sémantických rozdílů prováděnou nástrojem DiffKemp. Výsedky experimentů ukázaly, že za předpokladu, že jsou načteny relevatní dynamické vzory změn kódu, rozšíření dokáže eliminovat značné množství potenciálně nežádoucích rozdílů z výstupu nástroje DiffKemp. Pro ověření, že zavedení rozšíření neovlivnilo zbylé části nástroje DiffKemp, bylo rovněž spuštěno všech 122 regresních testů používaných nástrojem DiffKemp. Nakonec bylo pro zjednodušení budoucího ověřování funkčnosti rozšíření dodáno i 16 nových regresních testů. Všechny regresní testy skončily úspěchem, což dále svědčí o správné funkčnosti rozšíření.

Applying Code Change Patterns during Analysis of Program Equivalence

Declaration

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

.....
Petr Šilling
May 10, 2021

Acknowledgements

I would like to thank my supervisor Ing. Viktor Malík for his help with the understanding of important parts of DiffKemp and LLVM and for consultations regarding the theoretical aspects of the thesis.

Contents

Chapter 1

Introduction

When modifying code that should ideally remain stable and consistent for extended periods of time (e.g., for the lifetime of a major software release version), it might be crucial to know which parts of the program will be impacted, or, perhaps even more importantly, how will the changes affect semantics. To ease the process of finding unintentional side-effects, developers may want to utilize automated *static analyzers of semantic equivalence*, i.e., tools that can compare programs (or separate versions of the same program), displaying any potential semantic differences to the user. The problem is that current techniques for sound checking of semantic equivalence typically depend on computationally intensive formal methods. Consequently, the applicability of such tools to large-scale projects is fairly limited, forcing developers to rely almost entirely on an especially time-consuming and error-prone manual analysis instead.

Nevertheless, analyzers that concentrate on scalability and usability on large projects are slowly emerging as well. One such tool is DiffKemp, an analyzer of semantic differences between C programs, focusing particularly on the Linux kernel due to the fact that it is being developed in Red Hat. DiffKemp tries to find the middle ground between formal methods, which are sound but heavy-weight, and simplified light-weight methods (often based on plain text similarity)—it introduces a highly scalable technique that usually produces only a small number of false non-equivalence results.

In order to achieve this, DiffKemp translates both analysed programs from C into the LLVM intermediate code representation (LLVM IR) and attempts to compare them *instruction-to-instruction*. Unfortunately, unless the programs are syntactically the same, the instruction-to-instruction comparison by itself generates many false non-equivalence reports. To lower their amount as much as possible, DiffKemp also applies several code transformations in an attempt to bring the compared programs syntactically closer together and searches both programs for predefined patterns that are known to preserve semantics. These *semantics-preserving change patterns (SPCPs)* are especially important since they allow DiffKemp to handle even rather complex refactorings. However, as SPCPs are defined statically, they cannot properly respond to the needs of different developers, nor can they support changes that impact semantics, even if the impact is purely intentional.

With respect to the above, this thesis proposes, designs, and implements an extension of DiffKemp that allows to dynamically extend DiffKemp with custom patterns of code modifications. Compared to the existing SPCPs, which are a fixed part of DiffKemp, dynamic patterns enable users to specify which kinds of changes should be ignored during the semantic comparison. These do not necessarily have to be *semantics-preserving* pat-

terns (representing refactorings), but also *semantics-altering* patterns that represent code modifications verified to be safe (e.g., security fixes).

This thesis studies existing patterns used in low-level C projects, such as the Linux kernel, and prepares their encoding based on LLVM IR so that the patterns can be loaded and used by DiffKemp. Furthermore, the thesis proposes a method for detecting patterns in compared programs. This *matching method* is based on the *subgraph isomorphism problem* and leverages the LLVM infrastructure—in particular, the fact that LLVM functions are represented as *control-flow graphs (CFGs)*. More specifically, for each pattern, it tries to find a subgraph of CFGs of analysed programs that is *isomorphic* to the CFG-based representation of the pattern. Last, the thesis evaluates the extension on multiple past versions of the Linux kernel in terms of its impact on the analysis of semantic differences conducted by DiffKemp, demonstrating that it can eliminate a substantial amount of potentially undesirable differences from the output of DiffKemp.

The rest of the thesis is organised as follows. First, Chapter 2 gives a more detailed description of DiffKemp. Second, Chapter 3 presents an analysis of existing code change patterns, as well as an overview of methods for detecting patterns in compared programs. Chapter 4 follows by introducing a novel representation of dynamically defined code change patterns based on the intermediate representation of LLVM. Chapter 5 describes the design of the proposed DiffKemp extension. After that, Chapter 6 provides details about the implementation of the extension. Then, Chapter 7 evaluates the extension on past versions of the Linux kernel. Finally, Chapter 8 concludes the thesis, discussing possibilities for future work.

Chapter 2

Analyzing Semantic Differences using DiffKemp

This chapter describes the underlying concepts behind DiffKemp [17], a *static analyzer of semantic differences* between multiple versions of programs that we use as the target platform for our work. Compared to other tools for analysis of semantic differences, DiffKemp aims to scale on large-scale C projects, such as the Linux kernel, while maintaining high accuracy of the results at the same time.

This target objective of DiffKemp arises from two fundamental assumptions [17]: (1) existing techniques for sound equivalence checking[1] have difficulties concerning scalability because they generally depend on heavy-weight formal methods, and, on the other hand, (2) scalable light-weight analysers based on simple text similarity (such as the Unix `diff` tool) or abstract syntax tree matching [20] cannot conduct a proper analysis of semantic equivalence. In light of this, DiffKemp seeks the middle ground between the two approaches: it can analyse large-scale projects in a matter of minutes while being able to handle most common code refactorings. However, it is not sound and might fail to show the equality of some heavily refactored programs.

In its core, DiffKemp expects to receive two versions of the same program (with one being a refactoring of the other). To simplify the semantic comparison, the programs are translated into a lower-level language, in particular the LLVM *intermediate representation* (LLVM IR) [14]. Checking of semantic equality is then based on the following concepts:

- By default, an *instruction-to-instruction* comparison is conducted. This is a simple and very scalable approach which is especially useful if the compared programs are syntactically the same. On the other hand, it may lead to numerous false non-equivalence results, also known as *false-positives*.
- In order to bring the programs to a state where they can be compared instruction-to-instruction as often as possible, several *code transformations* are performed.
- Lastly, DiffKemp contains a list of predefined *semantics-preserving change patterns*. Changes matching these patterns are evaluated as semantically equal, even if they contain different instructions.

[1] **Analysis soundness**—the properties inferred by a sound analysis hold true for the given program in all of its possible executions [19].

The rest of this chapter is organized as follows. Section 2.1 gives a more detailed description of various existing approaches to static analysis of semantic equivalence. The representation of the compared programs—which relies on LLVM IR—is explained in Section 2.2. Section 2.3 formally defines the concept of semantic equality. Finally, the primary algorithm for equivalence checking used by DiffKemp is presented in Section 2.4.

2.1 Current State of Static Analysis of Semantic Equivalence

According to [13], *static analysis* is a technique for analyzing source code at compile time. In other words, it is the art of reasoning about the behaviour of computer programs without actually running them [19] (at least not with the original semantics), which derives properties that hold for all possible execution paths. This is in direct contrast with *dynamic analysis*, which derives properties that are valid for one or more execution paths of a *running* program [2].

Static analysis can provide a variety of insights about the analyzed code since its applications range from fairly simple programming error checkers to much more sophisticated formal analyzers and verifiers [13]. This thesis focuses on so-called *differential static analyzers*, which extract information about the differences between two programs. In particular, the thesis focuses on the analysis of *semantic differences* between these programs, which are expected to be separate versions of the same program.

In recent years, several projects on static analysis of semantic equivalence have emerged, creating a widely studied field of program analysis. The tools implemented based on these projects generally rely on costly formal methods, which—while eliminating false-positives common for more relaxed techniques—suffer greatly from scalability issues. Application on large enterprise projects, such as the Linux kernel, is therefore not feasible. Examples of such works are LLReve [11], SymDiff [12] or DiSE [22]. A more complete overview of analyzers of semantic differences can be seen in [17].

To give a concrete example, we present an experiment from [17], which evaluates LLReve—an open-source equivalence checker aimed at C programs compiled to LLVM IR. Because LLReve generates constraints in the form of Horn clauses, the analysis is quite time-consuming. Consequently, the tool fails to compare almost all functions from the Linux kernel in the 30-second long time frame provided by the experiment. Furthermore, LLReve does not support some operations common in industrial applications, such as calls via function pointers, floating-point arithmetic, and general bit operations. This results in several crashes during the comparison, which is unacceptable for commercial use.

Contrary to approaches based on formal methods, simpler and faster alternatives like the Unix `diff` tool also exist. While such tools are easily applicable to projects with the size of the Linux kernel, the information they provide originates from plain text comparison, making it unsuitable for a proper analysis of semantic differences.

Additionally, more advanced light-weight techniques exist as well. For example, [20] proposes to parse both compared programs and to produce their abstract syntax trees (ASTs). ASTs can then be traversed in parallel, creating a mapping for both variable names and types. As a result, simple semantics-preserving changes, such as variable renaming and type aliasing, may be handled with relative ease. However, even such tools would fail when confronted with more elaborate refactorings.

Considering the examples above, two essential properties of industrially applicable semantic equivalence analysers can be identified: (a) high evaluation speed and scalability, and (b) the ability to support complex refactoring patterns utilized in real-life projects.

While tools satisfying these conditions already exist, most of them concentrate only on providing an effective description of differences between the compared programs, and not on semantic equivalence itself. This applies to, e.g., JDiff [1], which is able to compare Java programs using their control-flow graphs (defined in Section 2.2).

Nevertheless, tools focusing on both practicality and the analysis of semantic equivalence exist as well. For example, BinHunt [9] generates an intermediate representation from binary files and uses it to construct control-flow graphs. The resulting graphs are then matched according to the subgraph isomorphism problem (described in Chapter 3). However, the usage of binary files may be quite limiting compared to DiffKemp, which primarily targets low-level C projects, because the loss of direct access to the corresponding source files also results in the loss of source code metadata. Therefore, some changes, e.g., regarding offsets of structure members, may be hard to correlate.

2.2 Representation of Compared Programs

Before the analysis of semantic differences may begin, DiffKemp needs to lower the level of abstraction of compared programs. Doing so not only makes the analysis easier and more language-independent but also hides multiple semantics-preserving changes. These may be, e.g., the syntactical changes between `for` and `while` loops, which are present only in higher level languages like C.

In particular, DiffKemp translates analysed C programs into a low-level source-language-independent code representation called LLVM IR—the *intermediate representation* of LLVM [14]. In general, LLVM is a modular compiler framework designed to provide high-level metadata useful for program analysis and code transformations. For example, LLVM provides an explicit function representation based on *control-flow graphs*, which can be used to retrieve information about control flow or to traverse analysed programs.

To formalize this representation, we provide the following definitions, which are based on [1, 14, 16, 17] and describe the relevant parts of LLVM IR. All examples and instructions assume that the C language and the LLVM infrastructure are used, although they may be simplified for the sake of brevity.

Control-flow graph (CFG) A directed graph in which nodes represent *basic blocks*, and edges represent the flow of control between *program branches*. Each LLVM *function* corresponds to a single CFG and may be perceived as one.

Basic block (BB) A sequence of LLVM *instructions* where only the first instruction (called the *entry instruction*) may be the target of jumps and which ends with exactly one *terminator instruction* and contains no other branching instructions.

Instruction An *operation* performed over a (possibly empty) list of operands, where each *operand* is a variable, a constant, or a function. The operation is characterized by the associated instruction kind, and all variables and constants are typed. Instructions may produce a result. For example, arithmetic instructions, such as `add` for integer addition and `fsub` for floating-point subtraction, need to store the resulting value. Instruction kinds and the LLVM type system are defined by [16].

Variable Variables may be local to a given function, or global. Local variables in LLVM do not necessarily correspond to local variables from original programs. In fact, there are two kinds of local variables: virtual registers and variables allocated on the stack by the `alloca` instruction. Registers are temporary variables produced as results of instructions and that satisfy the requirement that each of them is assigned to at most once (also known as the *static single assignment* property, or SSA). Registers can get accessed directly. On the other hand, variables that get allocated on the stack are operated through pointers and, therefore, `load` and `store` instructions have to be used to read and write their values, respectively. Only stack-allocated variables may correspond to variables from original programs. Global variables need to be accessed via `load` and `store` as well. Function parameters are a subset of all local variables. In the human-readable representation of LLVM IR, identifiers of local variables (and types) start with `%`, while the identifiers of global variables (and functions) are prefixed by `@`.

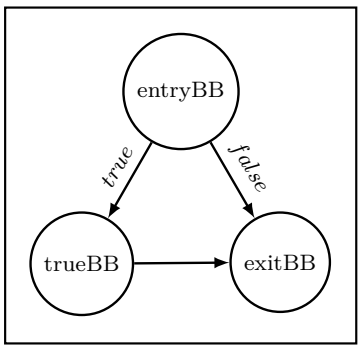
Metadata Additional information attached to instructions, functions, or global variables. Metadata are not typed and, in the human-readable representation of LLVM IR, are prefixed in syntax by `!`. Metadata can be either strings of characters or nodes. Metadata nodes group other metadata and values together (similarly to structured data types). Attached metadata get identified by name, which may be shared across metadata of the same kind.

Branching The (possibly conditional) flow of program control between connected basic blocks. Generally, branching occurs when one of the following three instructions gets executed: (a) the branching `br` instruction, which can branch both *conditionally* and *unconditionally*, (b) the `call` instruction, which represents an ordinary function call, or (c) the `ret` instruction, used to terminate the current function. Instructions that might be executed immediately after an instruction i are called the *successor instructions* of i.

C Code

```
1 int example(int param) {
2     int result;
3     if (param > 0)
4         result = fun(param);
5     return result;
6 }
```

Control-flow graph



LLVM IR Code

```
1  define i32 @example(i32) {
2  entryBB:
3    %1 = alloca i32 ; result allocation
4    %2 = icmp sgt i32 %0, 0
5    br i1 %2, label %trueBB, label %exitBB
6
7  trueBB:
8    %3 = call i32 @fun(i32 %0)
9    store i32 %3, i32* %1
10   br label %exitBB
11
12 exitBB:
13   %4 = load i32, i32* %1
14   ret i32 %4
15 }
```

Figure 2.1: Sample function defined in both C and LLVM IR, and the associated CFG.

Figure 2.1 shows a simplified structure of LLVM IR for a sample function written in C, and the corresponding CFG. Additionally, it shows that unconditional branches and standard non-branching instructions have only a single successor, while conditional branches (Line 5 in LLVM IR) typically have two successors.

2.3 Definition of Function Equality

After the analysed programs get translated into LLVM IR, a low-level code representation described in Section 2.2, DiffKemp may start comparing them. It does so by analyzing pairs of functions, utilizing one of the key aspects of LLVM IR, where each function is represented by a CFG composed of basic blocks. This not only divides compared programs into smaller, easily manageable segments (which is important for scalability) but also gives direct access to all instructions present in currently compared functions through their CFGs.

Since checking semantic equality of entire functions at once would be fairly complicated, DiffKemp achieves it by splitting each function into the same number of blocks that can be compared separately. These small blocks of code are delimited by so-called *synchronisation points*, at which both compared functions are *synchronized*, i.e., have the same state of memory (defined by both the stack and the heap). For each block of code located between two synchronisation points s_1 and s_1' taken from the first compared function, a block of code between two corresponding synchronisation points s_2 and s_2' that is semantically equal must exist in the second compared function [17]. Two blocks of code are considered semantically equal if and only if the following two conditions hold [17]:

1) During execution, the blocks either both terminate, or both do not terminate.

2) If the blocks terminate, they produce the same output for the same input, where input and output represent the values of all input variables and the initial state of memory, and the values of all output variables and the final state of memory, respectively.

Generally, the sets of (mainly local) variables used in the corresponding blocks of code are not the same. Therefore, a variable mapping indicating which variables from the first compared function are corresponding to which variables from the second compared function has to be created as well.

More formally, let f_1 and f_2 be two compared functions, and I_1, I_2 and V_1, V_2 their sets of instructions and variables, respectively. The *problem of checking semantic equality* can be then defined as the problem of finding two sets of synchronisation points $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$ and two synchronisation mapping functions: $smap : S_1 \leftrightarrow S_2$, creating a mapping of synchronisation points between f_1 and f_2, and $varmap : V_1 \leftrightarrow V_2$, creating an analogous mapping of variables, such that the blocks of code between pairs of corresponding synchronisation points are semantically equal. The above definition is a simplified version of the formal definition presented in [17].

Such mapping functions are rather hard to produce. Generally, both syntactical and control flow transformations and a sophisticated matching algorithm are required. An example containing a graphical representation of suitable *smap* and *varmap* mappings, and the associated LLVM IR of the two compared functions, is presented in Figure 2.2.

As can be seen in Figure 2.2, synchronisation points are typically located at each instruction. However, the example also shows that in certain scenarios, e.g, when using a syntactically different, although semantically equal algorithm, this might not be the case. For instance, it may be observed that the instruction i_1 in f_1 performs the same operation

Function f_1

```
1 define i32 @f1(i32 %x, i32 %y) {
2     %1 = mul i32 %x, 5
3     %2 = mul i32 %y, 4
4     %3 = add i32 %1, %2
5     %4 = mul i32 %3, %3
6     %5 = mul i32 %3, %4
7     ret i32 %5
8 }
```

Function f_2

```
1 define i32 @f2(i32 %x, i32 %y) {
2     %1 = shl i32 %x, 2
3     %2 = add i32 %x, %1
4     %3 = shl i32 %y, 2
5     %4 = add i32 %2, %3
6     %5 = call i32 @ipow(i32 %4, i32 3)
7     ret i32 %5
8 }
```

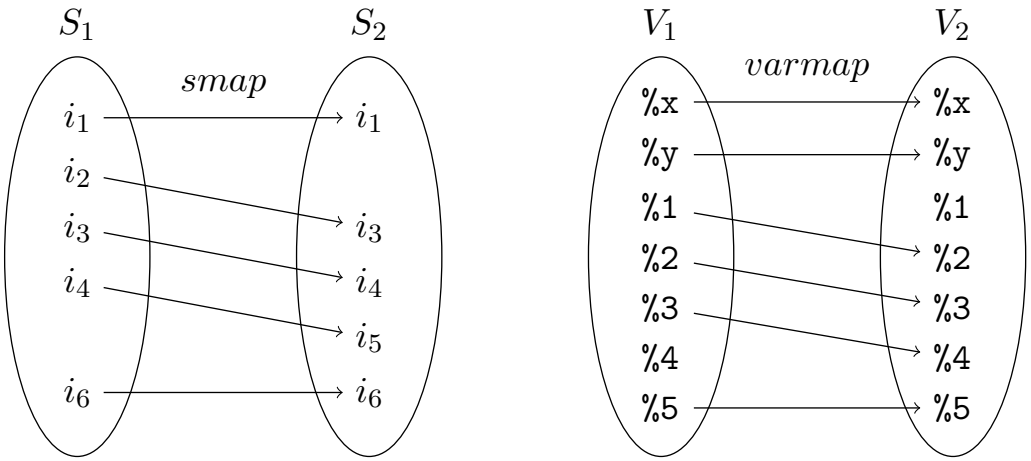


Figure 2.2: Two compared functions f_1 and f_2 with the associated *smap* and *varmap* mappings. For $n \in \{1, 2, \ldots, 6\}$, instructions from both functions are represented in order by i_n. Variables used in f_1 and f_2 are represented by the corresponding LLVM IR identifiers. For two parameters, x and y, f_1 and f_2 are semantically equal, as they both calculate the result of $(5x + 4y)^3$.

as the sequence of instructions i_1, i_2 in f_2. Therefore, the functions are synchronised before the execution of i_2 in f_1 and i_3 in f_2.

DiffKemp currently supports many of these special cases using the list of predefined *semantics-preserving change patterns (SPCPs)*. When a SPCP is identified, the blocks of code are considered semantically equal—even if the standard per-instruction comparison does not succeed. However, there are plenty of different refactorings which create a diverging synchronisation mapping, and a static list of SPCPs may be unable to handle all of them appropriately. Therefore, this thesis proposes an extension of DiffKemp, which provides support for dynamically defined, configurable patterns as well.

2.4 Algorithm for Checking Function Equality

This section introduces an algorithm for checking semantic equality of compared functions, based on the problem presented in Section 2.3. Specifically, for two functions f_1 and f_2, this section describes a method for finding the appropriate sets S_1 and S_2 of synchronisation points and the associated mapping functions *smap* and *varmap*. The code between all

matched synchronisation points should be semantically equal. The algorithm is a slightly simplified version of the top-level matching algorithm proposed by [17].

The most straightforward approach would be to place synchronisation points at each instruction, although this is achievable only when comparing programs that are syntactically the same, or at least extremely similar. However, by applying code and control flow transformations before the main algorithm begins, even different program constructions can be brought syntactically closer to each other (i.e., into a state where the ordinary per-instruction mapping of synchronisation points might be achievable). DiffKemp supports several of such transformations, all of which maintain semantic equivalence. In particular, transformations like function inlining, constant propagation, indirect call substitution, and redundant instructions, dead code, and dead parameter elimination are performed. To avoid scalability issues, exhaustive transformations, e.g., the inlining of function calls, are executed only lazily. Further details on supported transformations can be seen in [17].

When a per-instruction mapping of synchronisation points is found, all matching instructions are simply compared against each other, instruction-to-instruction. Two instructions are considered semantically equal if and only if they perform the same operation on the same number of operands that are (a) the same, or (b) can be mapped to each other using the *varmap* function [17].

In all other scenarios, the selected group of instructions has to be matched against the *semantics-preserving change patterns (SPCPs)*. As described in Section 2.3, if an SPCP match is found, the corresponding group of instructions is considered semantically equal as well and the algorithm may continue. However, if such a match cannot be found and the instruction-to-instruction comparison fails nonetheless, the functions are evaluated as semantically non-equal (and the difference may be displayed to the end-user).

Algorithm 2.1, which is based on [17], formalises these concepts using the ideas from Section 2.3. For brevity, the constituting functions are described only informally, and the implementation details behind SPCPs have been omitted. The full explanation can be found directly in [17].

Algorithm 2.1 operates on two compared functions, f_1 and f_2, and, for $i \in \{1, 2\}$, introduces P_i as the parameter list of f_i and G_i as the set of all global variables used in f_i. The algorithm returns *true* if the comparison succeeds, and operates on a queue Q of synchronisation point pairs prepared for further analysis.

The algorithm starts by applying all available code transformations (Line 1). This, among other things, removes all parameters which do not affect the output, i.e., those parameters that cannot change the semantics. If f_1 and f_2 do not have the same number of parameters (after the mentioned transformations), the functions are considered semantically different (Line 3).

Then, the initialisation of the synchronisation sets and the mapping functions begins. In the beginning, only the first instructions in the entry basic blocks of functions f_1 and f_2 are considered synchronized. These are denoted by i^1_{in} and i^2_{in} for f_1 and f_2, respectively, and are placed inside the corresponding synchronisation sets (Line 4). On Line 5, a synchronisation mapping is formed between i^1_{in} and i^2_{in} as well.

Furthermore, a variable mapping between pairs of parameters (Lines 6–7) and between pairs of global variables (Lines 8–9) is created. Parameters of f_1 and f_2 are mapped based on their order, while the global variables used in f_1 and f_2 are mapped according to their names. The remaining instructions and variables are processed lazily.

Afterwards, the primary comparison loop begins, operating until the queue Q of synchronisation point pairs is empty. Initially, before entering the main loop, the first synchronized

Algorithm 2.1: Analysing functions for semantic equivalence [17]

Input: Compared functions f_1 and f_2
Result: *true* if f_1 and f_2 are semantically equal, *false* otherwise

1: perform *code transformations* of f_1 and f_2
2: **if** $|P_1| \neq |P_2|$ **then** *// Require the same number of parameters*
3: **return** *false*

// Initialise the synchronisation sets and maps
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 and g_2 have the same name

// Start the main comparison loop
10: $Q := \{(i^1_{in}, i^2_{in})\}$ *// Begin with entry instructions*
11: **while** $Q \neq \emptyset$ **do**
12: take any pair (s_1, s_2) from Q
13: $p := detectPattern(s_1, s_2)$
14: **foreach** $(s'_1, s'_2) \in succPair_p(s_1, s_2)$ **do**
15: **if** s'_1 *or* s'_2 *has already been visited* **then**
16: ensure that s'_1 has already been mapped to s'_2
17: check *semantic equality* of blocks (s_1, s'_1) and (s_2, s'_2)

// Update the synchronisation sets and maps
18: $S_1 := S_1 \cup \{s'_1\},\ S_2 := S_2 \cup \{s'_2\}$
19: $smap(s'_1) := s'_2$
20: update *varmap* according to p
21: insert (s'_1, s'_2) into Q
22: **return** *true*

pair of instructions (i^1_{in}, i^1_{in}) is queued up for analysis (Line 10). The operations performed during each iteration are described below [17].

1) A single pair of synchronisation points (s_1, s_2) is taken from the queue Q (Line 12).

2) Function *detectPattern* checks whether the code blocks starting at s_1 and s_2 match some predefined semantics-preserving change pattern p (Line 13).

3) Function $succPair_p$ retrieves all possible successor synchronisation point pairs following (s_1, s_2). If no pattern has been identified, the successor pairs are placed at the instructions immediately following s_1 and s_2, i.e., a single pair or two pairs (for conditional branches) of synchronisation points are returned. SPCPs may define their own implementation of $succPair_p$, typically returning the instructions located immediately after the blocks that match the pattern. Each of the returned pairs is processed using the following steps:

i) If any synchronisation point in (s'_1, s'_2) has already been visited, it is required that both s'_1 and s'_2 are already mapped to each other (Lines 15–16). If no such mapping exists, the functions are considered semantically different, and $false$ gets returned.

ii) Blocks of code between the current and the next synchronisation point get semantically compared (Line 17). Unless a pattern is used, these blocks always contain only a single instruction, which is matched directly. If the blocks are semantically different, the algorithm claims that f_1 and f_2 are semantically non-equal as well, returning $false$.

iii) The synchronisation sets and maps get updated, and the pair (s'_1, s'_2) is inserted into the queue Q (Lines 18–21).

Finally, if the queue Q is successfully emptied, the functions f_1 and f_2 are considered semantically equal (Line 22), and the comparison of the next pair of functions may begin.

Chapter 3

Code Change Pattern Matching

Software development is a never-ending process, with numerous new features, patches, and enhancements getting implemented each year. When stability is a concern, developers might utilize tools like DiffKemp (introduced in Chapter 2) to ensure that the changes do not create unwanted semantic differences in code which should remain consistent for longer periods of time. Several of these changes can be described using so-called *code change patterns (CCPs)*, i.e., patterns of *recurrent* software modifications [18], which are especially important for DiffKemp due to the per-instruction nature of its semantic comparison. CCPs get formally defined in Section 3.1.

DiffKemp can already handle many CCPs in the form of semantics-preserving change patterns (SPCPs; described in Section 2.3). However, since the number of existing patterns is theoretically unbounded (each developer may use a completely different set of patterns), supporting all of them with predefined SPCPs is simply not feasible. Additionally, not all code change patterns have to describe semantics-preserving changes—on the contrary, quite a large subset of CCPs is composed of, e.g., security fixes, safety assertions, and other desirable semantic changes. As a result, two major kinds of code change patterns may be identified, both of which get described in more detail in Sections 3.2 and 3.3, respectively:

1) *Semantics-preserving patterns*, commonly known as *refactoring patterns*, i.e., patterns that modify code in a way that preserves its observable behaviour [8]. Some of these, e.g., the addition of a new value into an enumeration type, are already handled by SPCPs [17].

2) *Semantics-altering patterns*, corresponding to changes which cause semantic differences. For example, they might add, remove, or rewrite a conditional expression in an attempt to fix programming mistakes [21].

Consequently, while some highly repetitive CCPs may get incorporated into DiffKemp directly, it would be most beneficial to introduce dynamically defined patterns that can be tailored to specific developer needs, and a new pattern detection method capable of finding both types of CCPs in the compared programs.

The process of detecting CCPs may be considered a *pattern matching problem* since it compares generic descriptions of patterns with segments of code from analysed programs. In other words, it *matches* them against each other. Section 3.4 briefly introduces different approaches to pattern matching, in particular those presented in [5], i.e., *naive linear matching*, *control-flow graph matching*, and *dependence graph matching*, with emphasis on control-flow graph matching due to its straightforward applicability to LLVM functions.

3.1 Code Change Pattern Definition

Code change patterns (CCPs) are essential for the DiffKemp extension proposed later in this thesis. Therefore, before the different kinds of CCPs get presented, this section formally defines CCPs. The definition is tailored specifically to the proposed extension and builds on the idea that CCPs describe recurrent software modifications. Furthermore, the definition uses the notions of input, output, and mapping presented in Section 2.3.

Generally, a code change pattern can be understood as a pair of code fragments whose input and output can be mapped together. In other words, the code fragments describe two different ways to transform a semantically equivalent input to a semantically equivalent output. Additionally, both code fragments should have the following properties:

- One code fragment should be a transformation of the other. The transformation does not necessarily have to be semantics-preserving, although the semantics are typically either unchanged or changed only slightly.
- Each code fragment should come from a different version of the same program (or, in broader terms, from a different program). It is important to know which code fragment belongs to which version of the program. Therefore, the rest of this thesis will refer to the original code fragment from the older program version as the *old side* of the CCP, and to the modified one as the *new side* of the CCP.

The above structure of CCPs is depicted visually in Figure 3.1.

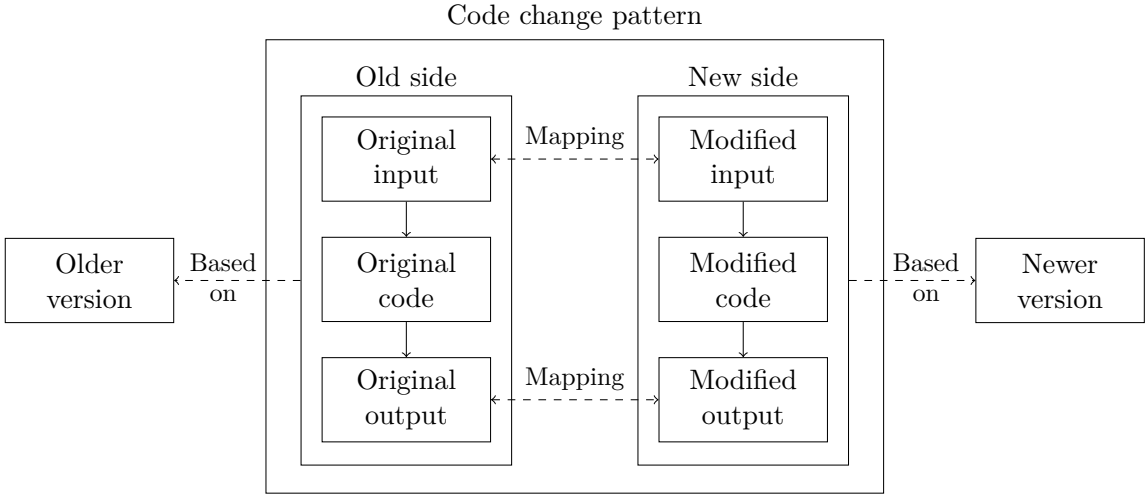


Figure 3.1: Structure of a code change pattern. The pattern consists of two fragments of code with mapped input and output. Both fragments correspond to a different version of the same program.

Formally, a code change pattern p is a tuple

$$p = (c_o, c_n, imap, omap)$$

where

- c_o and c_n are the *code fragments* associated with the older and newer versions of compared programs, respectively (i.e., the old side and the new side of p), and

- *imap* and *omap* are mapping functions that map the input of c_o to the input of c_n, and the output of c_o to the output of c_n, respectively.

A code fragment c can then be defined as a tuple

$$c = (i, o, b)$$

where

- b is the main body of c, composed of instructions and statements that describe how to transform an input into an output, and
- i and o are the input and the output of c, respectively.

For brevity, we also define functions $in(c)$ and $out(c)$, which denote the input and the output of a code fragment c, respectively.

3.2 Refactoring-Based Code Change Patterns

Refactoring is a crucial, albeit often overlooked, part of software development. It is the process of restructuring programs by applying series of transformations without changing the observable behaviour, making the programs easier to understand and to modify, and preventing their decay caused by advancements of other technologies [8].

This process is especially important for DiffKemp because, typically, two versions of the same program get compared (and not two separate programs). Additionally, it is expected that the purpose of the whole comparison is to check for the semantic equality of the two programs. In other words, it is suspected that one of the compared programs is actually a refactoring of the other. As stated in Chapter 2, a large number of refactorings is already supported—either by LLVM IR and code transformations or by SPCPs. However, there are still patterns which would cause DiffKemp to report semantic differences even when none are present. Therefore, this section evaluates existing catalogues, i.e., lists, of *refactoring patterns* (so-called *semantics-preserving patterns*), trying to find those that are not yet supported by DiffKemp but could be handled by an extension for matching dynamically defined patterns.

Most refactoring catalogues, e.g., the renowned Fowler's catalogue [8], are created for high-level, object-oriented languages such as Java. This inadvertently limits their application to source code written in C, which is targeted by DiffKemp. However, while some refactoring patterns in these catalogues are inapplicable to C code, e.g., because they operate on classes or interfaces, patterns that can be generalized to lower-level languages get presented as well. For example, the analysis of refactoring patterns provided by Fowler's catalogue revealed the following patterns which are currently not supported by DiffKemp:

- *Combine functions into transform*—combines related operations from multiple functions into a single, transformed function. While this pattern could, in some cases, be handled by, e.g., function inlining, it may result in much more complicated (and unsupported) refactorings as well.
- *Substitute algorithm*—replaces a complicated algorithm with a simpler alternative which preserves the original behaviour.

- *Code relocation*—the process of relocation of some pieces of code within a compared function. While DiffKemp supports code relocation, it only handles its most simplest cases, e.g., the extraction of independent variables outside of a loop [17]. Therefore, more sophisticated code relocation, such as the intertwining blending of two previously independent loops, remains unsupported. This can be considered as a complex extension of the *Slide Statements* pattern, which relocates some lines of related code so that they are placed closer together.

Refactoring catalogues for lower-level languages like C are, on the other hand, far less common. Nevertheless, some comprehensive lists of C patterns do exist. One such list is proposed by [10]—the same list that has been used in [17] to evaluate the number of semantics-preserving changes supported by DiffKemp. However, by examining the patterns from [10] that were not handled by DiffKemp, we were unable to discover any significant patterns. This is because (1) the three patterns related to pointer-to-variable and variable-to-pointer conversions were flagged by [17] as practically non-existent in the Linux kernel (and are, therefore, not relevant), and (2) the two remaining patterns, i.e., the conversion of a global variable into a parameter and the grouping of a set of variables into a new structure, could still not be fully supported without a proper analysis of the global state of the compared programs.

Consequently, we have conducted our own study of refactoring patterns used within the Linux kernel. The analysis has been performed by running DiffKemp on 42 past refactoring commits pushed to the Linux kernel GitHub repository[1], and by manually reviewing the reported semantic differences, searching for possibly repetitive patterns. In particular, four refactoring patterns have been discovered in these commits:

- *Replace check before list retrieval*—replaces a linked list emptiness check with a function that returns a `NULL` pointer if the accessed list is empty.
- *Extract code to function*—extracts related code into a new function. Unless it results in complex control flow refactorings, this pattern could also get handled by, e.g., function inlining.
- *Extract conditionally executed statements*—moves conditionally executed blocks of code outside of their respective `switch` or `if-else-if` statements, introducing a new flag variable that chooses which code to execute instead.
- *Introduce loop flag*—replaces a condition present inside of a loop with a new flag variable. The application of this pattern is demonstrated in Figure 3.2.

In total, seven suitable refactoring patterns have been identified. These are, however, only refactoring patterns, and developers might want to add certain semantics-altering changes as well. While such changes should hardly be considered refactorings, they do tend to accompany changes performed during refactoring sessions quite frequently. For example, a developer might introduce a new pointer value validity check when rewriting an old function. Therefore, Section 3.3 follows with an examination of catalogues of patterns that do affect program semantics.

[1] **GitHub repository of the Linux kernel**—https://github.com/torvalds/linux.

Original code

```
1 for (int i = 0; i < 5; i++) {
2     if (is_prepared(i)) {
3         perform_action();
4         break;
5     }
6 }
```

Modified code

```
1  bool flag = false;
2  for (int i = 0; i < 5; i++) {
3      if (is_prepared(i)) {
4          flag = true;
5          break;
6      }
7  }
8
9  if (flag) {
10     perform_action();
11 }
```

Figure 3.2: Example application of the *Introduce loop flag* refactoring pattern. The observable behaviour of both code samples is the same.

3.3 Semantics-Altering Code Change Patterns

Section 3.2 explains the importance of refactoring patterns. However, most software modifications actually do affect semantics. These changes can also be highly repetitive, and may already be known to be safe, in which case they do not have to reviewed again (even though they impact semantics). For example, security fixes might introduce new conditional assertions, undeniably changing program semantics in the process. While occasionally, such changes do get created as byproducts of poorly executed refactoring attempts, i.e., are purely accidental, in which case the developer should be notified about their existence, they might also be completely intentional, meaning that the developer may not want to receive any notifications about them at all.

Despite that, DiffKemp currently cannot distinguish between intentional and accidental semantic differences in any way, and always displays all of them. Therefore, this section examines catalogues of code change patterns that—contrary to the refactoring patterns presented in Section 3.2—do not preserve the semantics of affected programs, i.e., are *semantics-altering*. Again, a particular focus is given to patterns that are applicable to our problem domain, and to patterns discovered within the Linux kernel.

Similarly to refactoring catalogues, lists of semantics-altering code change patterns for high-level languages exist as well. For example, [21] presents a list of so-called *bug fix patterns*, i.e., descriptions of common changes performed when correcting programming mistakes. Even though the patterns in [21] are based solely on Java projects, they rarely depend on high-level language constructs. As a result, the vast majority of patterns presented in [21] are directly applicable to C code and, therefore, to DiffKemp as well. The list of relevant patterns from [21] is presented below. Note that some patterns, e.g., the addition and removal of conditions, have been grouped together for the sake of brevity.

- *Add or remove condition*—adds or removes a condition, typically a precondition or a postcondition of the related operation.
- *Add or remove conditional branch*—adds or removes a conditional branch from either a `switch` statement, or an `if-else-if` statement.

- *Change conditional expression*—changes the expression used within a conditional statement, e.g., by inverting the whole expression.
- *Change assignment expression*—changes the expression on the right-hand side of a variable assignment.

Moreover, for semantics-altering changes, even lists of patterns focusing directly on the Linux kernel can be found. In particular, [15] presents a study of bug fix patterns used within the Linux kernel (among other software), based on which the following patterns can be identified:

- *Allocate longer buffer*—increases the size of an allocated buffer to prevent its overflow.
- *Assign fewer bytes to buffer*—reduces the number of bytes written into a buffer because its capacity is smaller than the original number of assigned bytes.
- *Move `NULL` check before dereference*—moves a check for a `NULL` pointer before the corresponding dereference of the same pointer.
- *Add memory release*—adds a new statement for releasing allocated memory. The same can be achieved by relocating existing memory releasing statements so that they free the specified blocks of memory in all possible execution paths.

Last, we have again conducted our own analysis of frequent changes occurring in the Linux kernel. This time, the analysis has been performed on the kernel of the eighth major version of RHEL[2] due to its focus on long-term stability. Multiple functions from different release candidate versions of RHEL 8 have been compared using DiffKemp, and the reported semantic differences have been manually examined for the existence of semantics-altering code change patterns. The list of discovered patterns can be seen below.

- *Add single statement*—adds a previously missing statement (typically a function call) to a function.
- *Introduce assertion*—introduces a kernel-specific assertion statement, which (compared to traditional assertions) does create semantic differences.
- *Introduce flag parameter*—adds a flag parameter to a function. Generally, this also results in changes of the original function name and semantics (for all execution paths).
- *Substitute macro constant*—replaces a macro constant with a global variable or a function call. Doing so often modifies program semantics or—at the very least—changes the underlying LLVM IR code in a way that causes DiffKemp to detect semantic differences.
- *Wrap expression in macro*—envelops an expression using a macro function. Typically, a conditional expression gets wrapped in either the `likely` or the `unlikely` macro, as can be seen in Figure 3.3.

Overall, twenty code change patterns have been identified (including the refactoring patterns presented in Section 3.2).

[2] **Red Hat Enterprise Linux (RHEL)**—a commercial Linux distribution developed by Red Hat—https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux.

Original code

```
1 if (is_prepared()) {
2     perform_action();
3 }
```

Modified code

```
1 if (unlikely(is_prepared())) {
2     perform_action();
3 }
```

Figure 3.3: Example application of the *Wrap expression in macro* code change pattern. Compared to the example presented in Section 3.2, this pattern changes the semantics of the sample C code, and, therefore, cannot be considered as a refactoring pattern.

3.4 Finding Change Patterns in Code

Having a set of code change patterns (CCPs) is only a part of the problem. The patterns still need to get detected when present in the compared programs. This creates a *matching problem*, where patterns have to be systematically compared with analysed programs in order to possibly find a match, i.e., to identify the CCP used within the program (if such a CCP exists). Note that the matching procedure has to find a suitable match in *both* compared programs. That is because—as presented in Section 3.1—a CCP is characterized by both the original and the modified code (i.e., by its old side and its new side, respectively).

Due to the importance of pattern matching, this section briefly describes the three matching techniques presented in [5]—*naive linear matching*, *control-flow graph matching*, and *dependence graph matching*. A particular focus is given to the CFG matching since it can be applied directly to LLVM IR functions, which are the backbone of the semantic analysis conducted by DiffKemp. Patterns are expected to be represented by small, specialized segments of LLVM IR code (the representation is explained in detail in Chapter 4).

The naive matching method is, as its name implies, rather simple: it completely ignores control flow and strictly linearly iterates over program instructions, trying to match them to a pattern (using instruction-to-instruction comparison). Consequently, any patterns that depend on control flow branching cannot be discovered by naive matching, which makes it unsuitable for DiffKemp because many patterns presented in Sections 3.2 and 3.3 do rely on program branching. Nonetheless, this approach might be sufficient for some smaller patterns and may even be used as the backbone of more sophisticated procedures.

On the other hand, control-flow graph matching does analyse control flow because it uses CFGs (introduced in Section 2.2), meaning that it can be applied directly to LLVM functions—even when branching is involved. Specifically, it searches for subgraphs of program CFGs that are *isomorphic* to CFGs produced by patterns (which are expected to be much smaller), i.e., it tries to find a *subgraph isomorphism* between the two CFGs. Two graphs are isomorphic if and only if there exists a bijection between their sets of vertices that preserves edge adjacency [23] (other definitions of subgraph isomorphism and related terms can be found, e.g., in [6]).

As an example, let us consider the pattern graph and the program graph from Figure 3.4. There, the pattern graph is isomorphic to a subgraph of the larger program graph, i.e., a subgraph isomorphism exists between them. This is because for all vertices and edges present in the pattern graph, corresponding vertices connected by edges that have the same orientation can be found in the program graph.

It should be noted that finding a subgraph isomorphism is not an easy task—on the contrary, in its most general form, it is an NP-complete problem [4]. Consequently, many algorithms focusing on the subgraph isomorphism problem already exist. The most well-known

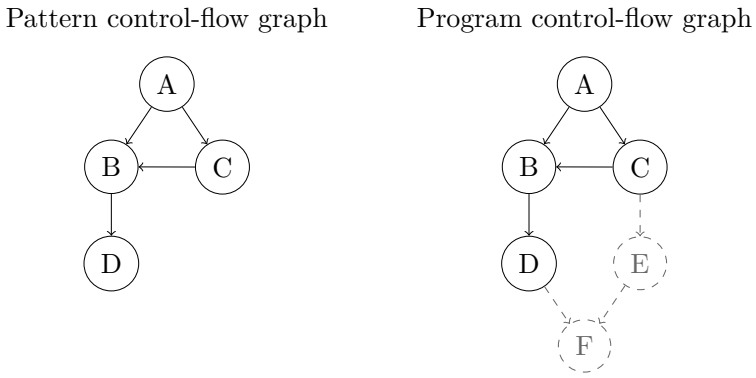


Figure 3.4: Example of a subgraph isomorphism between two control-flow graphs. The pattern CFG (left) is isomorphic to a subgraph (denoted by nodes and edges that are not dashed) of the larger program CFG (right). In order to successfully match a CCP, an isomorphism of this kind has to be found for both of its sides.

explore a tree-structured state space, where states represent feasible solutions. Examples of these include Ullmann's algorithm [23], which systematically iterates over matrices that encode possible subgraph isomorphisms, or the VF3 algorithm [4], which uses so-called *feasibility rules* to ensure the consistency of visited states. However, these approaches would be rather hard to integrate into the robust LLVM architecture since they typically aim at general graphs and not specifically at CFGs. Therefore, the matching process that we present in Chapter 5 does not rely on any of the previous algorithms. Despite that, its implementation utilizes certain heuristics that are mentioned in previous works (most notably, the pruning of graph nodes based on the number of neighbours).

Finally, dependence graph matching is a method that works with so-called program dependence graphs (PDGs)—graphs that portray not only control dependencies but also data dependencies for each performed operation [7]. In other words, PDGs extend CFGs with data dependencies. As a result, the general approach to the matching problem is the same as with CFG matching, with the only exception being that a PDG is matched instead of a CFG. Compared to CFG matching, PDG matching has one major advantage due to its ability to analyze data dependencies: it can successfully find a match even if extra instructions that are unrelated to patterns, e.g., initialization instructions for isolated blocks of memory, get inserted into the control flow. Despite that, explicit PDG matching is not very suitable for DiffKemp since DiffKemp analyses data dependencies using mapping functions (introduced in Section 2.3). Therefore, the matching procedure we introduce works on CFGs but exploits data dependencies as well.

Chapter 4

Representation of Change Patterns

Section 3.4 follows the introduction to *semantics-preserving* and *semantics-altering* code change patterns (CCPs), discussing essential preliminaries necessary for the pattern matching extension proposed in this thesis. However, before the extension can be described in further detail, it is important to define exactly how to encode real-world CCPs—originating directly from fragments of C code—in a way that is suitable for DiffKemp. An encoding may be considered *suitable* if, among other things, it (1) uses the two-side pattern structure proposed in Section 3.1, (2) can describe the patterns presented in Sections 3.2 and 3.3, and (3) requires a pattern parsing process that is inexpensive in terms of both time and implementation complexity.

Since DiffKemp utilizes the LLVM infrastructure, the most straightforward approach that satisfies the above conditions would be the direct use of LLVM IR, as doing so would not require any new libraries nor sophisticated parsing tools. Additionally, with a pattern representation based on LLVM IR, it would theoretically be possible to encode any pattern that might appear in the compared programs since DiffKemp represents programs using LLVM IR as well.

On the other hand, LLVM IR is a very low-level language. Therefore, larger patterns could be rather hard to produce manually, especially without prior knowledge of LLVM. Moreover, while it might be possible to encode all patterns, adding support for certain kinds of patterns, e.g., those describing changes in control flow, would also involve fairly complex modifications of the top-level comparison algorithm of DiffKemp itself. Chapter 6 further elaborates on this problem.

Nonetheless, an encoding based on specialized segments of LLVM IR code has been chosen as the most practical option available, mainly because of two reasons:

1. While it would be possible to, e.g., design a custom, text-based and more user-friendly encoding, doing so would likely be much less efficient as a sophisticated parser would have to get developed as well.

2. Manual creation of CCPs is not the primary focus of the proposed extension. On the contrary, since LLVM IR can be easily generated via the LLVM infrastructure, the whole pattern generation process could get automated, e.g., by potential future extensions of DiffKemp.

The rest of this chapter introduces the LLVM IR pattern representation. In particular, Section 4.1 proposes two kinds of pattern representations, while Section 4.2 provides details about custom LLVM metadata nodes, which are crucial for pattern parameterization.

4.1 Encoding Code Change Patterns

This section builds on the understanding of CCPs presented in Section 3.1 and proposes representations of CCPs based on LLVM IR (introduced in Section 2.2).

Since patterns can be, theoretically, created for any code modification, they may have different levels of complexity and may describe completely unrelated kinds of changes. Additionally, the study of patterns presented in Sections 3.2 and 3.3 suggests that a substantial amount of CCPs describe modifications of individual values, e.g., changes in macro constants or buffer allocations, or—on the contrary—modifications possibly spanning over multiple program branches, e.g., changes in conditions or code relocation. Therefore, we propose the following pattern representations:

1) *Instruction patterns*, capable of encoding all kinds of CCPs, including those that affect multiple program branches (i.e., multiple basic blocks).

2) *Value patterns*, which can efficiently encode CCPs that describe single-value changes.

Before describing both representations in detail, it should be noted that instruction patterns are generic. Therefore, they can also encode CCPs that are more suitable for value patterns (although doing so would result in a much less compact representation). To illustrate the differences between instruction patterns and value patterns more clearly, the following explanation will use the same CCP—describing a substitution of an integer macro for a global constant—for both representations. However, such CCPs would generally never be represented by an instruction pattern, although the decision ultimately depends on exactly how restrictive the given representation should be and is, therefore, strictly in the hands of the end user.

As stated in Section 3.1, each CCP can be characterized by two fragments of code, each from a different program version—the original code and the modified code, i.e., the *old side* and the *new side* of the pattern, respectively, and by the functions *imap* and *omap*, which define mappings between input and output, respectively. Therefore, to encode both sides of a pattern into LLVM IR, two independent blocks of code have to be present in the representation. This can either be done using separate LLVM modules or separate functions. Since having both pattern sides in the same module greatly increases clarity and eliminates the need to parse multiple modules, we propose to use two functions—one for each pattern side, i.e., each code fragment. To determine which function corresponds to which pattern side, we use special prefixes of their identifiers. In particular, functions with the `diffkemp.old` prefix belong to the old side of the pattern and functions with the `diffkemp.new` prefix belong to the new side of the pattern. This general idea is used by both the instruction and the value patterns, described below.

Figure 4.1 shows an example of the general structure of instruction patterns. As presented above, the representation is composed of two functions, each describing one side of the encoded CCP. Each function can be split into the following sections, each corresponding to its respective counterpart from the definition of code fragments provided in Section 3.1.

- *Input*, which is defined by function parameters.
- *Output*, which is denoted by the arguments of calls to the special `@diffkemp.mapping` function, which handles output identification and output mapping (explained below).
- *Main body*, generally consisting of all instructions and basic blocks present in the function. To increase readability and efficiency, custom `!diffkemp.pattern` metadata

nodes can get attached to instructions to limit the size of the main body. The concept of `!diffkemp.pattern` metadata gets discussed in further detail in Section 4.2.

Instruction pattern representation

```
1  ; For RHEL 8.1 (older version)
2  define void @diffkemp.old.side(i32) {
3    %2 = icmp sle i32 %0, 30
4    call void @diffkemp.mapping(i1 %2)
5    ret void
6  }
7
8  ; For RHEL 8.2 (newer version)
9  define void @diffkemp.new.side(i32) {
10   %2 = load i32, i32* @node
11   %3 = icmp sle i32 %0, %2
12   call void @diffkemp.mapping(i1 %3)
13   ret void
14 }
```

Figure 4.1: Instruction-based representation of a code change pattern extracted from differences reported by DiffKemp during the comparison of two versions of the RHEL kernel. The pattern describes a substitution of an integer macro (old side) for a global constant `@node` (new side). The example has been simplified for brevity.

Finally, in instruction patterns, the mapping functions *imap* and *omap* are encoded in the following way:

- *imap* gets created by mapping function parameters from both functions in order (both functions must have the same number of parameters).
- *omap* gets specified by calling the `@diffkemp.mapping` function before returning from either function. All instructions that get used as arguments of this call get mapped together (pairwise in order, since the call should be present in both functions). For single values, the return instruction can be used in the same manner as the `@diffkemp.mapping` function call.

Combining all parts of instruction patterns creates a universal CCP representation that can be easily parsed by tools from the LLVM infrastructure. However, while instruction patterns are universal, they are also quite sizeable—even when encoding a relatively simple CCP, as can be seen in Figure 4.1. Moreover, they might be rather limiting for certain use cases, since they need to directly specify all instructions that should be present in the compared programs. For example, in Figure 4.1, the representation should encode a macro substitution pattern. While it certainly does so, it also has to include specific information about the instructions present in the main body of the pattern—for example, it has to specify that the macro substitution has to be tied to a value comparison performed by an `icmp` instruction. This, however, does not necessarily have to be the case for many places where the pattern occurs. To address this issue, we propose to use so-called *value patterns*, which can encode patterns describing single-value changes much more efficiently.

Again, the structure of value patterns consists of two separate functions (one for each pattern side), as can be seen in Figure 4.2. However, compared to instruction patterns, both functions only contain a single return instruction, which returns the value prescribed by the encoded CCP (global variables, such as the variable `@node`, are returned as a pointer). That is possible since value patterns are no longer generic—they are restricted to single-value changes. Therefore, they do not need to directly encode code fragments with respect to the definition from Section 3.1. Instead, value patterns only encode a single pair of values that—even though they might be different—should always be compared as equal when, e.g., used as instruction operands. The complete structure of the pattern (i.e., of its code fragments and mapping functions) can be then determined lazily during pattern matching depending on the instructions present in the compared program versions. This enables users to describe single-value changes without tying patterns to specific instructions.

For RHEL 8.1

```
1 define i32 @diffkemp.old.side() {
2   ret i32 30
3 }
```

For RHEL 8.2

```
1 define i32* @diffkemp.new.side() {
2   ret i32* @node
3 }
```

Figure 4.2: Value-based representation of the pattern presented in Figure 4.1. The pattern has been simplified using only return instructions referencing the values required by the pattern. Contrary to the instruction-based representation, this representation does not enforce the existence of specific instructions, such as an `icmp` value comparison instruction.

The proposed pattern representations also support several other features, e.g., prefixes for preventing symbol name collisions. However, these are beyond the scope of this thesis and have been therefore omitted for brevity.

4.2 Pattern-Specific LLVM Metadata Nodes

Since instruction patterns are robust and universal, it may be necessary to specify additional details that further define the structure of the encoded CCP. Doing so may not be strictly necessary in some cases. However, in others, it might optimize pattern matching. Therefore, this section introduces a method for encoding auxiliary information about CCPs into the generic structure of instruction patterns proposed in Section 4.1.

Since instruction patterns are built around LLVM IR instructions, the encoding of supplementary information should reflect that. For example, it might require additional instructions—e.g., calls to specialized functions—to be inserted into the main body of instruction patterns. However, while doing so would be possible, it would increase the complexity of pattern matching since the LLVM toolchain would parse the additional instructions as a direct part of pattern control-flow graphs. Therefore, it would be best to utilize features of LLVM IR that have been designed with auxiliary information in mind—in particular, LLVM metadata (introduced in Section 2.2).

LLVM uses multiple kinds of built-in metadata. In addition, it also enables users to create custom metadata that can then, for example, be attached to instructions through metadata nodes, which would be ideal for instruction patterns. Therefore, we propose to support the metadata presented in Table 4.1. All metadata nodes that include the pattern-specific metadata should be attached to instructions through the `!diffkemp.pattern` identifier.

Table 4.1: Overview of custom kinds of metadata available for instruction patterns.

Metadata kind	Semantics
`pattern-start`	Marks the first pair of differing instructions (used for pattern matching optimization).
`pattern-end`	Labels the end of the main body of a code fragment. After this kind of metadata, only the code fragment output and its mapping may get specified.
`group-start`	Denotes the start of an instruction group. Grouped instructions have to be matched as a single block (no additional instructions are allowed between them).
`group-end`	Indicates that the active instruction group has ended.
`disable-name-comparison`	Disables name-based comparison of structures, replacing it with a complete type equality verification.

The specific kinds of metadata get specified by metadata strings (i.e., strings that have the `!` metadata prefix), which should be placed inside the attached metadata nodes and match the identifier of the desired metadata kind. Additionally, since metadata nodes are similar to structured types, it is possible to define multiple kinds of metadata in a single metadata node. Figure 4.3 demonstrates how to add two different kinds of metadata, `pattern-start` and `group-start`, to an instruction via a single metadata node.

```
1 !0 = !{ !"pattern-start", !"group-start" }
2 call void @example(), !diffkemp.pattern !0
```

Figure 4.3: Simplified example of a `call` instruction with attached `!diffkemp.pattern` metadata. The metadata node `!0` appends two kinds of metadata: `pattern-start` and `group-start`.

With the proposed kinds of metadata, instruction patterns should be well-prepared to encode any of the patterns presented in Sections 3.2 and 3.3. Additionally, even if some other helpful kinds of metadata were to be discovered in the future, adding support for them should be straightforward due to the flexibility of LLVM metadata.

Chapter 5

Design of the DiffKemp Extension

This chapter builds on the preliminary ideas behind DiffKemp and code change patterns (CCPs) described in previous chapters, focusing on the goal of this thesis, the design of the pattern matching DiffKemp extension. Considering the fact that both instruction patterns and value patterns need to be supported, the extension can be divided into the following parts.

First, since the two code fragments of each CCP are independent of each other, they need to be matched to the code in the corresponding programs separately. Therefore, a top-level algorithm that provides a simple interface for the matching process of instruction patterns must exist. The top-level algorithm can then execute the matching process for each pattern code fragment and analyse the resulting mapping of pattern input and output. The top-level matching algorithm for instruction patterns gets introduced in Section 5.1.

Second, the matching algorithm for code fragments of instruction patterns has to be created. This algorithm is presented in Section 5.2 and is the most sophisticated part of the extension since instruction patterns are (in contrast to value patterns) fully defined with respect to the CCP definition from Section 3.1. The algorithm must be executed twice for each instruction pattern—once for each side of the pattern (i.e., pattern code fragment and the associated program version). During each execution, the algorithm gradually compares instructions from the selected pattern side and its corresponding program. If a match for all pattern instructions is found on both sides, and if all input constraints defined by the pattern are satisfied, the top-level matching algorithm detects a match.

Finally, value patterns have to be processed. However, since value patterns only specify a pair of values that should be present in the compared programs, they need to be initialized based on the code around the instructions to which the pattern should be matched. Doing so lazily generates fully-defined patterns, which can then be analysed using the matching algorithm for instruction patterns. In other words, for each value pattern, its internal instruction pattern representation is created according to the given comparison context. The generation of instruction patterns from value patterns is described in Section 5.3.

5.1 Top-Level Matching Algorithm

This section explains the top-level instruction pattern matching algorithm, which controls the selection and execution of the lower-level control-flow-based pattern code fragment matching algorithm (described in Section 5.2). All of the presented algorithms expect that DiffKemp is comparing two versions of the same program.

Since pattern matching should serve as the final validation step before declaring two functions as semantically different, pattern matching should be run after the blocks of code (s_1, s_1') and (s_2, s_2'), compared on Line 17 of Algorithm 2.1, get determined as not equal. Then, instead of immediately interrupting the ongoing comparison of functions f_1 and f_2 because the blocks (s_1, s_1') and (s_2, s_2') differ, the pattern matching procedure should check whether the blocks can be matched to one of the loaded LLVM IR patterns. If a match is found, the comparison should disregard the difference associated with the pattern. Otherwise, the comparison should keep its original result and return. Additionally, it should be noted that the matching of dynamically loaded patterns could also theoretically be incorporated directly into the SPCP detection process from Algorithm 2.1. However, we have decided not to do so, mainly because of two things:

1) Since the number of dynamically loaded pattern may potentially be quite large, we want to only start the matching process after a difference between two compared instructions gets detected to achieve higher efficiency. Therefore, it is better to begin matching after the difference first gets detected by the main function comparison algorithm.

2) The matching procedure for dynamically loaded patterns can skip instructions unrelated to the pattern that is being matched. However, the skipped instructions still need to be compared by the primary function comparison algorithm afterwards.

The top-level pattern matching procedure is described by Algorithm 5.1. The algorithm expects to receive the first pair of instructions (i_o, i_n) that has been compared as semantically different by Algorithm 2.1, where i_o belongs to the block (s_1, s_1') from the older version of the compared program and i_n belongs to the block (s_2, s_2') from the newer version. Additionally, the algorithm needs access to the original mapping functions *smap* and *varmap* from Algorithm 2.1, and requires a set P_i of instruction patterns that should be used as the basis for pattern matching. Value patterns are not given to the algorithm, since they should be converted to instruction patterns based on context (the conversion process is explained in detail in Section 5.3).

Algorithm 5.1 starts by iterating over patterns $(c_o, c_n, imap, omap)$ until a match is found or all patterns are exhausted (in which case, the algorithm returns an empty set since no matching pattern has been identified). At the beginning of each iteration, the algorithm calls the function $matchCFG$, which is responsible for single-side instruction pattern matching (presented in Section 5.2). In particular, $matchCFG$ is called twice—the first time with i_o and the pattern code fragment c_o corresponding to the same program version (which, in the case of i_o, is the older version) and the second time with i_n and c_n (i.e., for the newer program version). The calls to $matchCFG$ return two tuples $(r_o, imatch_o, omatch_o, M_o)$ and $(r_n, imatch_n, omatch_n, M_n)$—for the older and newer program version, respectively—where, for $x \in \{o, n\}$, the following holds:

- r_x is the primary result of the single-side matching process (*true* if it successfully finds a match, $false$ otherwise).

- $imatch_x$ is the mapping of input matches, which provides information about how to map the input of the code fragment c_x to variables from the corresponding program version x. In other words, it maps the input variables of c_x to the matching variables from x.

Algorithm 5.1: Top-level instruction pattern matching

Input: Pair of differing instructions (i_o, i_n)
smap and *varmap* from Algorithm 2.1
Set of available instruction patterns P_i

Result: A set of matched instructions, which is empty if no pattern is matched

```
1:  foreach (c_o, c_n, imap, omap) ∈ P_i do
2:      (r_o, imatch_o, omatch_o, M_o) := matchCFG(i_o, c_o)
3:      (r_n, imatch_n, omatch_n, M_n) := matchCFG(i_n, c_n)
4:      if r_o ∧ r_n then
            // Check the mapping of inputs
5:          valid := true
6:          foreach (i_c^o, i_m^o) ∈ imatch_o do
7:              i_c^n := imap(i_c^o)
8:              if varmap(i_m^o) ≠ imatch_n(i_c^n) then
9:                  valid := false
10:                 break
11:         if ¬valid then continue
            // Synchronize outputs
12:         foreach (o_c^o, o_m^o) ∈ omatch_o do
13:             o_c^n := omap(o_c^o)
14:             smap(o_m^o) := omatch_n(o_c^n)
15:             varmap(o_m^o) := omatch_n(o_c^n)
16:         return M_o ∪ M_n
17: return ∅
```

- $omatch_o$ is the mapping of output matches, which contains a similar mapping of the output of c_x. In particular, it maps the instructions (or, more specifically, the variables created by them) that got marked as the output of c_x to the matching instructions from x (i.e., to the corresponding variables).

- M_x is the set that contains instructions from x that have been matched to instructions from the pattern code fragment c_x. In other words, the set identifies which instructions exactly are affected by the matching pattern (the previous mappings only contain information related to the input and the output of c_x and not its main body).

Concrete details about the creation of the tuples returned from calls to *matchCFG* are presented in Section 5.2.

After getting the results of both evaluations of *matchCFG*, the top-level algorithm checks whether both pattern code fragments have been matched successfully (Line 4). If so, it continues by validating the correctness of the mappings $imatch_o$ and $imatch_n$ with respect to the input mapping function *imap* of the current pattern. The validation phase uses a helper variable *valid* (which holds the input validation result) and iterates over all mapped input match pairs (i_c^o, i_c^m) that are given by $imatch_o$. For each pair, the following actions are performed:

- The corresponding input variable i_c^n from the newer program version is retrieved from *imap* using i_c^o. This is possible since *imap* maps the input of the code fragment c_o to the input of c_n. Therefore, it must also contain the mapping between i_c^o and i_c^n, since both belong to the input of c_o and c_n, respectively. Considering the fact that each code fragment input must be matched to exactly one variable from the corresponding program version, *imap* effectively creates a bijective mapping between $imatch_o$ and $imatch_n$.

- The variables that got matched to i_c^o and i_c^n (i.e., $i_m^o = imatch_o(i_c^o)$ and $imatch_n(i_c^n)$, respectively) are validated with respect to *varmap* (Lines 8–10). This ensures that the matched variables are either the same or can be mapped together, i.e., that the pattern matches in both c_o and c_n use the same (or mapped) input variables. Doing so is necessary since the matching of input is done separately for each pattern code fragment. Therefore, the variables matched to code fragment input could be potentially completely unrelated to each other.

If the input validation fails (i.e., if $false$ gets assigned to *valid*), the pattern match is discarded, and the algorithm continues with the following pattern, if available (Line 11).

Afterwards, the algorithm analogically iterates over the mapped output match pairs (o_c^o, o_m^o) taken from $omatch_o$. At the start of each iteration, it retrieves the corresponding output variable o_c^n using the pattern output mapping function *omap*. Then, the matched output variables $o_m^o = omatch_o(o_c^o)$ and $omatch_n(o_c^n)$ are mapped together via both *smap* and *varmap* (Lines 14–15). This is in direct contrast with the input validation process since—contrary to the variables matched to code fragment input, which should be created by instructions placed before the first differing pair of instructions (i_o, i_n) and, hence, that should have already been processed and mapped to each other accordingly—instructions associated with the output variables have not yet been analysed by the main function comparison algorithm (i.e., cannot be mapped together before Line 14 of Algorithm 5.1).

Finally, after the pattern input is successfully validated and the output mappings are created, the algorithm returns $M_o \cup M_n$ (Line 16), i.e., the set of all instructions that have been matched to the detected pattern, combined for both of its code fragments c_o and c_n. It should be noted that the algorithm could be extended to support multiple pattern matches originating from the same instruction pair. The extension would be rather straightforward—instead of returning immediately after the first pattern match is found, the algorithm could continue analysing other patterns, keeping the previous results (i.e., the unions of M_o and M_n), and merging them into a single set, which would be then returned after all patterns are analysed.

Additionally, the processing of a pattern match continues even after Algorithm 5.1 returns. In particular, Algorithm 2.1 has to retain the results of all successful pattern matches for the duration of the whole function comparison and use them to identify which instructions (from either of the compared functions) have been matched to a pattern. All matched instructions should then be skipped since instructions associated with a code change pattern should always be considered semantically equal, regardless of potential semantic differences. This skipping of instructions should be performed immediately after the main function comparison algorithm takes the pair (s_1, s_2) of synchronisation points from its queue Q (i.e., after Line 12 of Algorithm 2.1). In particular, if any of the instructions represented by s_1 and s_2 have been identified as parts of a dynamic pattern match, they should be skipped, i.e., their successors should be used instead.

5.2 Pattern Code Fragment Matching

This section follows the explanation of the top-level pattern matching algorithm for instruction patterns presented in Section 5.1. In particular, it describes the matching procedure for code fragments, which has to be evaluated twice for each analysed pattern, i.e., it explains the semantics of the function *matchCFG* used within Algorithm 5.1. Similarly to the primary function comparison algorithm, the single-side matching procedure utilizes the LLVM infrastructure to analyse pattern code fragments by their control-flow graphs (CFGs; defined in Section 2.2). However, compared to Algorithm 2.1, it does not aim at finding instructions that are semantically different. Instead, it searches for instructions that can be matched to those present in the main body of the given code fragment.

Algorithm 5.2: Control-flow-based pattern code fragment matching

Input: Differing instruction i_p^d from one of the compared programs
The corresponding pattern code fragment c

Result: r, the result status, which is *true* if a match gets found, *false* otherwise
imatch, the mapping of input matches
omatch, the mapping of output matches
M, the set of all matched program instructions

matchCFG (i_p^d, c):

1: $M := \{\}$
2: initialize $varmap_c$ with shared global variables
3: $Q := \{(i_c^b, i_p^d)\}$
4: **while** $Q \neq \emptyset$ **do**
5: take any pair (i_c, i_p) from Q
6: **if** i_c *can be matched to* i_p **then**
// Let $(o_1^c, \ldots, o_n^c)$ *and* $(o_1^p, \ldots, o_n^p)$ *be the operands of* i_c *and* i_p*, respectively.*
7: **foreach** $1 \leq k \leq n$ **do**
8: **if** $o_k^c \in in(c)$ **then**
9: $imatch(o_k^c) := o_k^p$
10: **if** $i_c \in out(c)$ **then**
11: $omatch(i_c) := i_p$
12: $M := M \cup \{i_p\}$
13: $varmap_c(i_c) := i_p$
// Queue up the following instruction pair
14: **foreach** $(i_c', i_p') \in succInstPair(i_c, i_p)$ **do**
15: insert (i_c', i_p') into Q
16: **if** *all instructions in c have been matched* **then**
17: **return** $(true, imatch, omatch, M)$
18: **else**
19: **return** $(false, imatch, omatch, M)$

The implementation of the function *matchCFG* is shown in Algorithm 5.2. The algorithm expects to receive the first instruction i_p^d from one of the analysed program versions that has been compared as semantically different and the pattern code fragment c cor-

responding to the same program version as i_p^d. In other words, using the notation from Algorithm 5.1, it requires i_o and c_o for the older program version, and i_n and c_n for the newer program version. The selected program version is denoted by p.

Algorithm 5.2 starts by creating the set M of instructions from p that have been matched to instructions from the main body of c. The set M, as well as the mapping of input matches *imatch* and the mapping of output matches *omatch*, which are also used within Algorithm 5.2, are all initially empty (their semantics are described in detail in Section 5.1). Additionally, a mapping between global variables that are used within both c and p and that share the same name is established (Line 2).

Then, the primary matching loop begins. The loop works similarly to Algorithm 2.1—it relies on the queue Q, operating until Q is emptied. However, since pattern matching is always done instruction-to-instruction and never on larger blocks of code, instructions are at all times queued up in the same sequence as they are present in the underlying LLVM IR code. Therefore, synchronisation points and the corresponding mapping function *smap* are not necessary. Initially, only the instruction pair (i_c^b, i_p^d) is queued up for matching, where i_c^b denotes the first instruction in the main body of c.

At the beginning of each iteration, a single pair of instructions (i_c, i_p) is taken from Q, and the algorithm checks whether i_c can be matched to i_p. Similarly to Algorithm 2.1, the matching of individual instructions is based on simple instruction-to-instruction comparison. The variable mapping function $varmap_c$ is used during the instruction matching process to check the correspondence between already matched variables (and the instructions that created them).

Then, let $(o_1^c, \ldots, o_n^c)$ and $(o_1^p, \ldots, o_n^p)$ be the operands of i_c and i_p, respectively. If the matching of i_c to i_p succeeds, the algorithm first processes the input variables (Lines 7–9). In particular, it maps each operand o_k^c that is also the input of c to the corresponding operand o_k^p by the the mapping of input matches *imatch*, $1 \leq k \leq n$. In other words, if o_k^c is the input of c, a mapping between o_k^c and the matching operand of i_p is created.

Additionally, Lines 10–11 perform a similar analysis with the output of c. However, it is the instructions themselves (or, more specifically, the variables created by them) that may be used as part of the output of c (i.e., not their operands). Therefore, if c specifies i_c as its output, i_p gets directly mapped to i_c by the mapping of output matches *omatch*. After the input and the output are processed, the program instruction i_p (i.e., the instruction that has been matched to i_c) is placed into the set of matched instructions M (Line 12). Moreover, a variable mapping between i_c and i_p is generated (Line 13).

Afterwards, the function *succInstPair*—a specialized variant of *succPair* from Algorithm 2.1—retrieves all instruction pairs (i'_c, i'_p) that should be queued up after (i_c, i_p). The implementation of *succInstPair* is displayed in Algorithm 5.3, which operates on the currently analysed instructions i_c and i_p from Algorithm 5.2.

If a match between i_c and i_p has been established, *succInstPair* behaves analogously to *succPair* (Line 2), i.e., it returns the instructions immediately following i_c and i_p, if available. Otherwise, if i_p has a single successor (i.e., if it is an internal instruction of a basic block or an unconditional branch instruction), *succInstPair* indicates that the matching algorithm should keep i_c and try to match it to the instruction that immediately follows i_p. In other words, the algorithm allows to skip instructions of p when searching for suitable instructions matching those in the main body of c. The function *succ* retrieves the single successor of the given instruction. If the number of immediate successors of i_p is different, *succInstPair* yields an error, failing the pattern matching process, as it either

Algorithm 5.3: Calculating successor instruction pairs

Input: Currently analysed instructions i_c and i_p
Result: Tuple of successor instruction pairs

succInstPair (i_c, i_p):
1: **if** i_c *can be matched to* i_p **then**
    // Use the default successor calculation
2: **return** $succPair(i_c, i_p)$
3: **else if** i_p *has a single successor* **then**
    // Try to match i_c *to the next program instruction*
4: **return** $(i_c, succ(i_p))$
5: **else**
6: yield error

cannot continue (i_p is a terminator instruction and, therefore, has no successors) or would branch out (i_p is a conditional branch instruction, i.e., has two successors).

It should be noted that since *succInstPair* yields an error even when i_p has two successors, the pattern matching procedure might fail to match patterns that describe changes in control flow (e.g., additions of new conditional statements)—it may abort the matching process even when suitable instructions in p exist in all execution paths following i_p (which, in this case, must be a conditional branch instruction). However, although the algorithm could potentially be extended to support this use case, e.g., with a backtracking procedure, patterns describing changes in control flow would still not work properly. This is because Algorithm 2.1 expects the compared functions f_1 and f_2 to have the same control-flow graphs (at least if they should be compared as semantically equal) and would have to be modified significantly to be able to process matches of the patterns in question. Such modifications of the core parts of DiffKemp would be beyond the scope of this thesis.

Finally, after the queue Q is emptied, Algorithm 5.2 returns the resulting mappings *imatch* and *omatch*, and the set of matched instructions M. Additionally, if all of the instructions in the main body of c have been matched, the algorithm returns *true* to indicate a successful match. Otherwise, it also returns *false* since the matching of c to p failed. The results are then further analysed by the top-level instruction pattern matching algorithm, as described in Section 5.1.

5.3 Generating Instruction Patterns from Value Patterns

So far, the algorithms presented in this chapter have only dealt with instruction patterns. However, Chapter 4 also introduced so-called value patterns—specialized variants of patterns that are only composed of a single pair of values, i.e., do not properly represent code change patterns with respect to the definition provided in Section 3.1. Therefore, this section presents a conversion process, which can generate instruction patterns based on the descriptions of value modifications given by value patterns loaded into DiffKemp and the context of the code to which the patterns should be matched.

An overview of the instruction pattern generation process is shown in Algorithm 5.4. The algorithm takes the first pair of instructions (i_o, i_n) that have been compared as se-

Algorithm 5.4: Generating instruction patterns from value patterns

Input: Pair of differing instructions (i_o, i_n)
Set of available value patterns P_v
Result: Set P_i^v of instruction patterns generated from value patterns

1: $P_i^v := \{\}$
2: **foreach** $p_v = (v_o, v_n) \in P_v$ **do**
3: **if** *an instruction pattern can be created from* p_v, i_o *and* i_n **then**
4: $p_i^v := createInstPattern(p_v, i_o, i_n)$
5: $P_i^v := P_i^v \cup \{p_i^v\}$
6: **return** P_i^v

mantically different (i.e., the same pair as the one used in Algorithm 5.1) and the set P_v of all value patterns loaded into DiffKemp, and tries to generate a new instruction pattern for each pattern in P_v.

In particular, the algorithm starts by creating an (initially empty) set P_i^v of generated instruction patterns, and iterating over all value patterns $p_v = (v_o, v_n)$ present in P_v. During each iteration, unless p_v cannot be converted, it executes the pattern conversion procedure *createInstPattern* which—based on the value pattern and the pair of differing instructions given to it—creates an internal-only instruction pattern p_i^v, which is inserted into P_i^v.

The function *createInstPattern* generates instruction patterns according to the values (and their types) present in the given value pattern p_v and the instructions that the newly created instruction pattern should be matched to (i.e., the context of the first differing instructions i_o and i_n). In particular, the code fragment for the older program version, which is denoted by p_o, of each of the generated instruction patterns is constructed using one of the methods described below. For v_n and i_n, the code fragment for the newer version is created analogously.

- If v_o, i.e., the value corresponding to p_o, is a constant (for example, an integer) and i_o uses v_o as one of its operands, the code fragment is constructed as follows:
  - The main body will contain the instruction i_o (since it has v_o as its operand).
  - The input will consist of all of the operands of i_o that are not constant (i.e., that reference variables that should be present in the code before i_o).
  - The output will be empty if i_o does not return a value (and, hence, does not create any variables that need to be mapped) or will be composed of the variable produced by i_o.
- If v_o is a pointer to a global variable (pointers to local variables cannot be used in value patterns since only return instructions are allowed) and i_o is a load instruction that loads from the global variable referenced by v_o (an example of this case can be seen in Figure 4.1), the process is similar to the one for constant values. However, since i_o is a load instruction, it is the instruction i'_o, denoting the instruction that uses the value loaded by i_o, that needs to be placed inside the main body of the code fragment and analysed with regards to the input and the output of the code fragment. Additionally, the load instruction i_o must be placed into the main body as

well (before i'_o, since i'_o depends on the result of i_o). Hence, the operand of i'_o that references the value loaded by i_o should not be regarded as a part of the input of the code fragment.

If neither of the above cases can be applied (e.g., since v_o is a pointer to a global variable but i_o is not a load instruction), the construction of the instruction pattern gets aborted. The same applies to the case when both fragments could be created, but the instructions that use the values prescribed by the value pattern do not perform the same operation in both program versions (e.g., the older version adds the value, while the newer version subtracts it), as such a change cannot be attributed to a value pattern alone.

When both code fragments are constructed successfully, the instruction pattern p_i^v is finalized by creating the mapping functions *imap* and *omap*, where *imap* is built according to the order of the instruction operands used as the input of the code fragments, and *omap* is generated analogically with respect to the output of the code fragments (which, in this case, can contain at most a single output variable in each code fragment).

Finally, after analysing all available value patterns, Algorithm 5.4 returns the set P_i^v of newly generated instruction patterns (which might possibly still be empty). This set should be constructed lazily before each evaluation of Algorithm 5.1, to which it should be passed on as a part of the set P_i of available instruction patterns.

Chapter 6

Extension Implementation

This chapter describes the most important implementation details of the pattern matching extension of DiffKemp proposed in this thesis. The extension is based on the algorithms presented in Chapter 5.

Since the extension serves as an additional component of the DiffKemp analyser, it is written in the C++ programming language[1]—the same language used by the core parts of DiffKemp—and its source code is accessible through the official public GitHub repository of DiffKemp[2] under the Apache 2.0 license. The integration of the pattern matching extension into the architecture of DiffKemp is described in Section 6.2, and the most notable aspects of the implementation process are presented in Section 6.3. Due to the influence of DiffKemp on the implementation of the pattern matching extension, Section 6.1 first introduces the core components of SimpLL, the function comparison module of DiffKemp.

6.1 Architecture of SimpLL

This section describes the parts of the architecture of DiffKemp that are the most relevant to the proposed pattern matching extension. In particular, it explains the operations performed by the core components of SimpLL, the core C++ module of DiffKemp, responsible for the comparison of two functions translated to LLVM IR (i.e., for the checking of semantic equivalence of functions presented in Section 2.4). SimpLL is typically executed from the front-end command-line interface of DiffKemp, which is written in Python. However, it may operate as a standalone module as well. The essential components of the architecture of SimpLL, shown in Figure 6.1, are described below. The descriptions are inspired by the source files of DiffKemp available in its GitHub repository.

LLVM IR parser A composite component that represents the LLVM IR parsing tools from the LLVM infrastructure. Despite the fact that SimpLL only compares two functions at once, LLVM IR files are always processed in their entirety. Therefore, the whole modules represented by the given LLVM IR files are the result of the parsing process.

[1] A small module related to the extension is also implemented for the front-end of DiffKemp (written in the Python language). However, this module only handles new command-line arguments introduced with the extension and not the matching of patterns. Therefore, it has been omitted to simplify the presentation.

[2] **GitHub repository of DiffKemp**—https://github.com/viktormalik/diffkemp.

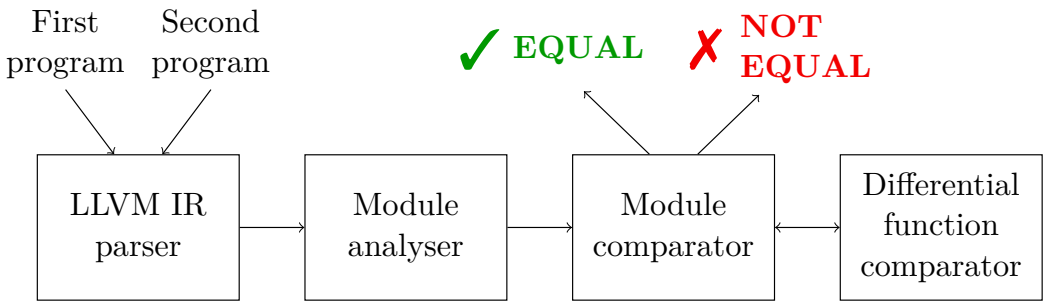


Figure 6.1: The original architecture of SimpLL, the core component of DiffKemp responsible for function comparison.

Module analyser Controls the analysis of the parsed modules. In particular, it applies all available semantics-preserving code transformations (briefly introduced in Section 2.4) and—with the help of the module comparator—begins the comparison of the selected pair of functions.

Module comparator Manages the comparison of the given pair of functions, producing the final result of the analysis. By default, it only passes the functions to the differential function comparator. However, when the default comparison finds semantic differences and the differences correspond to function calls, the module comparator also iteratively tries to repeat the comparison after performing function inlining (with the goal of potentially correcting a false-positive).

Differential function comparator An extension of the LLVM function comparator[3] that allows it to operate on functions from different modules. Compares the given pair of functions using the control-flow-based analysis presented in Section 2.4.

6.2 Integration of the Pattern Matching Extension

The extension proposed in this thesis has been implemented as a part of the architecture of DiffKemp or, more specifically, the architecture of the SimpLL module (introduced in Section 6.1). The pattern matching algorithms from Chapter 5, which the extension builds upon, have been implemented in a straightforward manner—with the exception of the value pattern conversion algorithm presented in Section 5.3, which is partially replaced by a direct value-to-value comparison. The core components of the pattern matching extension, as well as the value-to-value comparison, are described below. The integration of the components into the architecture of SimpLL is displayed in Figure 6.2.

YAML parser Parses pattern configuration files, which are written in YAML[4] and contain the paths to the LLVM IR patterns that should be loaded. The parsing of pattern configuration files is performed by the appropriate tools from the LLVM infrastructure, and the identified pattern files are handed over to the LLVM IR parser.

[3] **LLVM function comparator**—https://llvm.org/doxygen/classllvm_1_1FunctionComparator.html.

[4] **YAML Ain't Markup Language (YAML)**—a human-friendly data serialization language designed to work well with all modern programming languages [3].

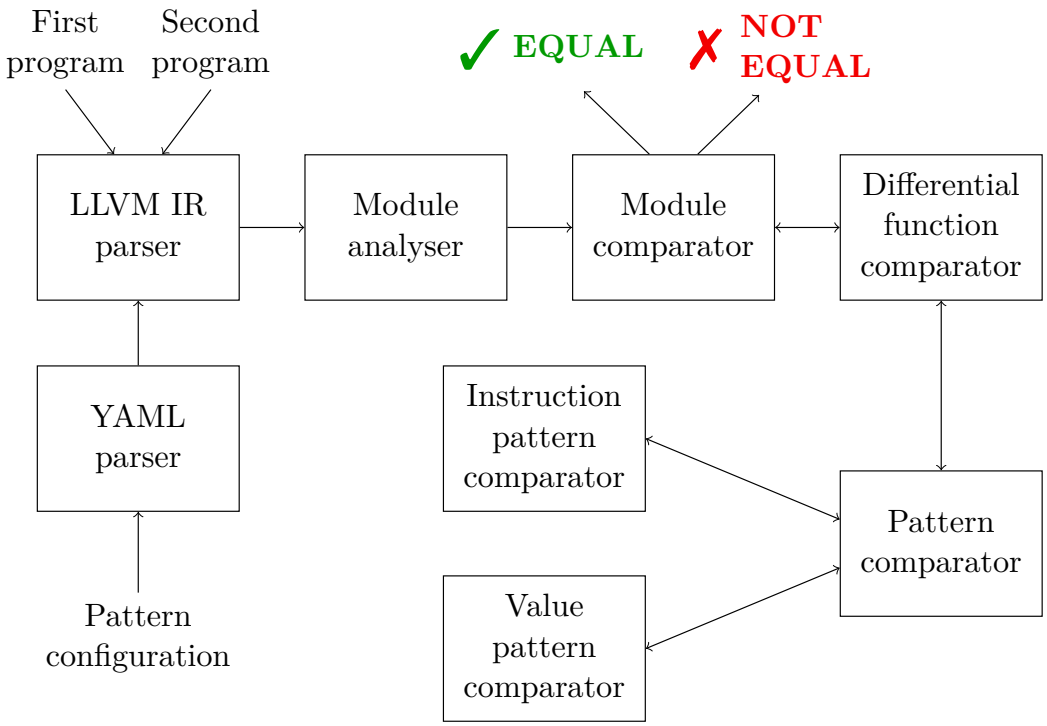


Figure 6.2: Architecture of SimpLL after the integration of the pattern matching extension.

Pattern comparator Controls the pattern matching process, i.e., it implements the top-level matching algorithm presented in Section 5.1. In other words, if a difference is found, it tries to match the code starting with the first pair of instructions compared as semantically different by the differential function comparator to one of the loaded patterns. To achieve this, it utilizes the single-side pattern comparators specialized for the specific kinds of patterns (i.e., the instruction pattern comparator and the value pattern comparator). Additionally, the pattern comparator also analyses the code around the received pair of instructions with respect to the value pattern conversion process described in Section 5.3.

Instruction pattern comparator Implements the control-flow-based matching algorithm for code fragments of instruction patterns from Section 5.2. It does so by extending the LLVM function comparator, which performs a top-down analysis of function differences, starting with basic blocks and continuing by comparing individual instructions, operands, and other values and variables. Program instructions that are compared as equal to those present in the given code fragment are considered to match the corresponding pattern.

Value pattern comparator A specialized variant of the LLVM function comparator that optimizes the matching of value patterns. This is possible since the LLVM function comparator analyses code in a top-down manner. In particular, it also analyses instruction operands individually and independently of the associated instruction. Therefore, the values specified in value patterns can be compared directly, without the need to explicitly generate instruction patterns (although the general idea behind the comparison stays the same). However, any additional load instructions related to value patterns containing pointers to global variables must still be processed separately by the higher-level pattern comparator.

Together, the newly introduced components enable SimpLL or, more specifically, the differential function comparator to receive LLVM IR patterns and use them to validate the discovered differences. In particular, before the differential function comparator declares any two instructions as semantically different, it first passes them to the newly added pattern comparator, which may then match them to one of the loaded patterns (and, in doing so, allow the differential function comparator to disregard the difference). The differential function comparator was extended so that it can properly process the results of the pattern matching analysis. More implementation details regarding the pattern matching process can be found in Section 6.3.

6.3 Extending the LLVM Function Comparison Module

This section gives further details about the implementation of the code fragment matching algorithm introduced in Section 5.2. In particular, it describes the most notable implementation details of the instruction pattern comparator, which tries to match a pattern code fragment (i.e., the LLVM function that represents it) to a function from the corresponding compared program, in which a differing instruction has been identified.

Generally, to implement the algorithm as an extension of the LLVM function comparator, only minor modifications of the original comparison process were necessary. For example, the LLVM function comparator can only compare the given pair of functions as a whole (i.e., from the entry basic block to the terminating basic block or blocks). Therefore, the instruction pattern comparator had to include a mechanism that allows it begin from the instruction where the first known difference between the originally compared programs is located. Nevertheless, some more substantial changes had to be implemented as well in order to increase the efficiency and accuracy of pattern matching. These are listed below.

- The instruction pattern comparator might skip internal instructions of basic blocks and unconditional branch instructions based on the next expected instruction. In particular, if the comparator needs to match a branch instruction (since it is the next unmatched instruction in the pattern code fragment), it may jump to the first available branch instruction that has the same number of immediate successors (if such an instruction exists).

- Since basic blocks connected by unconditional branch instructions may be considered as a single basic block, the instruction pattern comparator additionally allows to skip unconditional branch instructions unless they are directly specified by the given pattern code fragment. This effectively enables the comparator to match a pattern even to a function that has a different control-flow graph (provided the differences are caused by unconditionally connected basic blocks).

Chapter 7

Experiments and Testing

The previous chapters have introduced a pattern matching extension of DiffKemp with the goal of removing potentially undesirable or wrongly reported (i.e., false-positive) differences from the output of DiffKemp. This chapter follows by presenting a series of experiments, which was performed in order to verify that the extension is able to eliminate differences associated with the dynamically loaded LLVM IR patterns. As the target of the experiments, Red Hat Enterprise Linux (RHEL) was chosen due to its popularity and emphasis on stability—its kernel contains a list of functions, a so-called *Kernel Application Binary Interface* (KABI), that should ideally remain semantically stable for the lifetime of each major release of RHEL [17]. The outcome of the experiments is discussed in Section 7.1.

Additionally, to enable faster verification of the extension in the future and to check the correctness of as many execution paths of the matching process as possible, a number of new regression tests were created and executed. Moreover, since the incorporation of the extension also required small modifications of the parts of DiffKemp related to the main program comparison, the previously available regression tests were used to ensure that the core of DiffKemp remained unchanged even when no patterns are utilized. The regression tests also target the kernel of RHEL, and their results are described in Section 7.2.

The output produced by DiffKemp during the experimental evaluation as well as the LLVM IR patterns from both the evaluation and regression testing are available on the attached memory medium (contents of which are described in Appendix A). Additionally, Appendix B presents the steps necessary to execute both the experiments and the tests.

7.1 Experimental Evaluation on the Linux Kernel

This section describes experimental evaluation of the extension proposed in this thesis. The extension was evaluated by performing a series of experiments on the kernel of RHEL. In particular, three pairs of the most recently released versions of RHEL[1] were selected. For each pair of versions, the following sequence of actions was performed:

1) The KABIs of both versions were compared by DiffKemp without using patterns.

2) The reported differences were saved and manually examined for the existence of code change patterns (CCPs; introduced in Chapter 3). In particular, in each pair of versions, we—to the best of our abilities—searched for the five most repetitive CCPs.

[1] At the time of testing, the three pairs of the most recent release candidate versions of RHEL consisted of RHEL 8.0/8.1, RHEL 8.1/8.2, and RHEL 8.2/8.3.

Table 7.1: A comparison of pairs of RHEL versions with and without LLVM IR patterns

RHEL versions	KABI functions	Non-equal results without patterns	Non-equal results with patterns	Runtime (mm:ss) without patterns	Runtime (mm:ss) with patterns
8.0/8.1	471	85	**75**	04:46	**04:43**
8.1/8.2	521	161	**146**	05:15	**05:13**
8.2/8.3	628	178	**169**	07:26	**07:08**

3) All of the identified CCPs were encoded into LLVM IR (using the encoding methods proposed in Chapter 4).

4) The KABIs of the selected pair of versions were compared again, this time with all of the created LLVM IR patterns being loaded into DiffKemp.

5) The results of both comparisons were analysed in terms of execution time and the number of KABI functions proclaimed semantically different (i.e., *not equal*). Additionally, it was manually verified that only the differences related to the identified CCPs were affected by the loaded LLVM IR patterns.

The results of the experiments can be seen in Table 7.1. Each experiment was repeated five times, and the runtimes were calculated as averages of the time spent on comparing KABI functions compiled to LLVM IR on a 4 core, 2.80 GHz Intel Core i7 Kaby Lake machine with 16 GB of RAM. The compilation time is not included in the statistics.

For all pairs of the compared versions of RHEL, the results show that by applying few patterns, the total number of KABI functions evaluated as not equal can be lowered. In particular, on average, each pattern removed approximately 2.27 differences declared as not equal. It should be noted, however, that the actual amount of eliminated non-equal results varied considerably due to differences in repetitiveness of the CCPs discovered among pairs of RHEL versions. For instance, across all pairs of versions, we were able to find only a single pattern that could remove more than five non-equal results by itself. The high number of 15 eliminated non-equal functions between RHEL versions 8.1 and 8.2 can be attributed precisely to this repetitive CCP. On the other hand, the lower repetitiveness of other patterns does not indicate any potential issues with the extension since—according to our manual inspection—all of their occurrences were successfully identified by the pattern matching procedure.

Additionally, the results reveal one rather interesting fact—after applying patterns, the version comparison was consistently faster by a few seconds. That may come as a surprise since the proposed extension only introduces a new pattern matching analysis (i.e., the execution time should generally rise). However, by lowering the number of non-equal functions, DiffKemp does not need to locate the corresponding differences in the original C source code nearly as often as before. Since difference localisation is one of the most demanding operations performed by DiffKemp, the total execution time may be lower even when analysing patterns.

Based on the findings presented above, we were able to confirm that our extension can help to improve results reported by DiffKemp (at least for the evaluated versions of RHEL) since many false-positives can be linked to a particular CCP. Moreover, the results indicate that generally, usage of patterns might have a slightly positive impact on runtime performance (which is one of the most crucial factors related to DiffKemp). Since the

proposed pattern representation is generic, these findings also suggest that the extension should be broadly applicable to C projects other than the kernel of RHEL.

7.2 Regression Testing

To ensure that the changes incorporated into DiffKemp do not affect its output in an undesirable way, DiffKemp contains a set of 122 regression tests, all of which were passing before the development of the extension proposed in this thesis. The tests focus on three pairs of versions of the RHEL 7 kernel as well as a single pair of versions of the upstream Linux kernel and are specified inside configuration files that, for each pair of versions, contain a list of symbols (usually a subset of KABI functions) that should be compared, and the expected comparison results.

Since the regression tests rely on RHEL 7 (i.e., not the currently most recent version of RHEL), we decided not to extend the original tests with pattern-related features. Instead, we created a completely new set of function comparison tests, which are dedicated to the verification of pattern matching and—similarly to the experimental evaluation presented in Section 7.1—target the KABI functions in the three pairs of the most recently released versions of RHEL. The tests, as well as the pattern matching extension itself, support all LLVM versions from 5.0 to 11. Therefore, they can be used directly in the continuous integration of DiffKemp.

The set of regression tests for pattern matching consists of 16 new tests, each of which requires a selected group of LLVM IR patterns to be loaded into the testing instance of DiffKemp. When a pattern matching test gets executed, it first loads the necessary LLVM IR patterns. Afterwards, it compares the two KABI functions that are specified in the configuration file of the test. Finally, the test checks whether the obtained equality verdict is the same as the expected result. The patterns used for regression testing were designed with the goal of analysing as many execution paths of the pattern matching procedure as possible. Repetitive patterns were preferred as well. An overview of the new set of regression tests can be seen in Table 7.2. A single regression test for RHEL versions 8.1 and 8.2 depends on two LLVM IR patterns. All other pattern matching tests only require a single pattern.

Table 7.2: An overview of the new set of regression tests dedicated to pattern matching

RHEL versions	**Number of tests**	**Number of patterns**	**All tests passed**
8.0/8.1	6	6	**yes**
8.1/8.2	5	6	**yes**
8.2/8.3	5	5	**yes**

After fully incorporating the pattern matching extension into DiffKemp, we executed all available regression tests again. All 138 (122 original and 16 new) tests successfully passed, suggesting that the extension works as intended on all of the selected KABI functions and that the previous features of DiffKemp remain unaffected by its addition.

Chapter 8

Conclusion

In this thesis, we have analysed so-called code change patterns (CCPs), i.e., patterns of repetitive software modifications. Due to the presence of CCPs in low-level C code, including the Linux kernel, we have also proposed, designed, and implemented an extension of DiffKemp (an analyser of semantic differences), capable of identifiying CCPs in compared programs. The extension is able to load a selected set of patterns, each of which must be encoded in LLVM IR as a pair of functions. The dynamically loaded patterns can then be matched to the code from the compared programs, potentially eliminating unwanted differences (e.g., false-positives) from the output of DiffKemp. The matching process is based on an instruction-to-instruction comparison, which proceeds in sequence according to control flow.

The extension was evaluated on multiple pairs of the most recent versions of the kernel of Red Hat Enterprise Linux, showing that with a proper set of patterns, it can eliminate a number of non-equivalence results reported by DiffKemp. Doing so should further decrease the amount of work required to check semantic stability between different versions of a program—especially since a substantial number of reported differences related to patterns is composed of false-positives. The extension was also evaluated by multiple regression tests, which showed that its addition did not negatively impact other parts of DiffKemp.

Future work could improve the pattern matching extension and the related aspects of DiffKemp in several ways. For example, a more user-friendly and, ideally, fully automated method of generating LLVM IR patterns might be introduced. Additionally, CCPs that describe modifications of control flow, e.g., introductions of new conditional statements, could get supported (although doing so would require extensive changes in the core parts of DiffKemp). Last, the extension could be improved by lowering the time complexity of the matching algorithm, e.g., by adding more heuristics. However, since the conducted experiments do not reveal any issues regarding time complexity, focusing on other aspects of the extension might be better.

Bibliography

[1] Apiwattanapong, T., Orso, A. and Harrold, M. J. A Differencing Algorithm for Object-Oriented Programs. In: *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society, September 2004, p. 2–13. ASE '04. DOI: 10.1109/ASE.2004.1342719. ISBN 0-7695-2131-2.

[2] Ball, T. The Concept of Dynamic Analysis. In: *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Berlin, Heidelberg: Springer-Verlag, October 1999, vol. 24, no. 6, p. 216–234. ESEC/FSE-7. DOI: 10.1145/318774.318944. ISBN 978-3-540-48166-9.

[3] Ben Kiki, O., Evans, C. and Net, I. döt. YAML Ain't Markup Language (YAML™) Version 1.2. [online]. July 21, 2009. Revised 1. 10. 2009 [cit. 2021-05-02]. 3rd edition. Available at: https://yaml.org/spec/1.2/spec.html.

[4] Carletti, V., Foggia, P., Saggese, A. and Vento, M. Introducing VF3: A New Algorithm for Subgraph Isomorphism. In: Foggia, P., Liu, C.-L. and Vento, M., ed. *Graph-Based Representations in Pattern Recognition*. Cham, Switzerland: Springer International Publishing, May 2017, p. 128–139. LNCS 10310. ISBN 978-3-319-58961-9.

[5] Conradi, T. *Matching of Control- and Data-Flow Constructs in Disassembled Code*. Hamburg, Germany, 2015. Bachelor's thesis. Hamburg University of Technology, Institute for Software Systems.

[6] Conte, D., Foggia, P., Sansone, C. and Vento, M. Thirty Years Of Graph Matching In Pattern Recognition. *International Journal of Pattern Recognition and Artificial Intelligence*. Singapore: World Scientific Publishing Company. May 2004, vol. 18, p. 265–298. DOI: 10.1142/S0218001404003228. ISSN 0218-0014.

[7] Ferrante, J., Ottenstein, K. J. and Warren, J. D. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*. New York, NY, USA: Association for Computing Machinery. July 1987, vol. 9, no. 3, p. 319–349. DOI: 10.1145/24039.24041. ISSN 0164-0925.

[8] Fowler, M. *Refactoring: Improving the Design of Existing Code*. 2nd ed. Boston, MA, USA: Addison-Wesley Professional, November 2018. ISBN 978-0-13-475759-9.

[9] Gao, D., Reiter, M. K. and Song, D. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In: Chen, L., Ryan, M. D. and Wang, G.,

ed. *Information and Communications Security*. Berlin, Heidelberg: Springer-Verlag, October 2008, p. 238–255. ICICS '08. DOI: 10.1007/978-3-540-88625-9_16. ISBN 978-3-540-88624-2.

[10] Garrido, A. *Software Refactoring Applied to C Programming Language*. Urbana-Champaign, USA, 2000. Master's thesis. University of Illinois.

[11] Kiefer, M., Klebanov, V. and Ulbrich, M. Relational Program Reasoning Using Compiler IR. *Journal of Automated Reasoning*. Berlin, Heidelberg: Springer-Verlag. September 2017, vol. 60, no. 3, p. 337–363. DOI: 10.1007/s10817-017-9433-5. ISSN 0168-7433.

[12] Lahiri, S. K., Hawblitzel, C., Kawaguchi, M. and Rebelo, H. SymDiff: A language-agnostic semantic diff tool for imperative programs. In: Parthasarathy, M. and Seshia, S. A., ed. *Computer Aided Verification*. Berlin, Heidelberg: Springer-Verlag, July 2012, p. 712–717. CAV '12. DOI: 10.1007/978-3-642-31424-7_54. ISBN 978-3-642-31423-0.

[13] Lahiri, S. K., Vaswani, K. and Hoare, C. A. R. Differential Static Analysis: Opportunities, Applications, and Challenges. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. New York, NY, USA: Association for Computing Machinery, November 2010, p. 201–204. FoSER '10. DOI: 10.1145/1882362.1882405. ISBN 978-1-4503-0427-6.

[14] Lattner, C. and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. Palo Alto, CA, USA: IEEE Computer Society, March 2004, p. 75–86. CGO '04. DOI: 10.1109/CGO.2004.1281665. ISBN 0-7695-2102-9.

[15] Liu, C., Yang, J., Tan, L. and Hafiz, M. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In: *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. Washington, DC, USA: IEEE Computer Society, March 2013, p. 282–291. ICST '13. DOI: 10.1109/ICST.2013.24. ISBN 978-0-7695-4968-2.

[16] LLVM Project. LLVM Language Reference Manual. *LLVM 11 Documentation* [online]. January 14, 2021. Revised 15. 1. 2021 [cit. 2021-01-21]. Available at: https://releases.llvm.org/11.0.1/docs/LangRef.html. Path: LLVM Home; Documentation; Reference; LLVM Language Reference Manual.

[17] Malík, V. and Vojnar, T. Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects. In: *Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation*. Porto de Galinhas, Brazil: IEEE Computer Society, April 2021, p. 329–339. ICST '21.

[18] Martinez, M., Duchien, L. and Monperrus, M. Automatically Extracting Instances of Code Change Patterns with AST Analysis. In: *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, September 2013, p. 388–391. ICSM '13. DOI: 10.1109/ICSM.2013.54. ISBN 978-0-7695-4981-1.

[19] Møller, A. and Schwartzbach, M. I. *Static Program Analysis* [online]. Aarhus: Department of Computer Science, Aarhus University, October 2018, revised 30. 11. 2020 [cit. 2021-16-01]. Available at: http://cs.au.dk/~amoeller/spa/.

[20] Neamtiu, I., Foster, J. S. and Hicks, M. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. In: *Proceedings of the 2005 International Workshop on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, May 2005, vol. 30, no. 4, p. 1–5. MSR '05. DOI: 10.1145/1083142.1083143. ISBN 1-59593-123-6.

[21] Pan, K., Kim, S. and Whitehead, E. J. Toward an Understanding of Bug Fix Patterns. *Empirical Software Engineering*. Norwell, MA, USA: Kluwer Academic Publishers. June 2009, vol. 14, no. 3, p. 286–315. DOI: 10.1007/s10664-008-9077-5. ISSN 1382-3256.

[22] Person, S., Dwyer, M. B., Elbaum, S. and Păsăreanu, C. S. Differential Symbolic Execution. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, November 2008, p. 226–237. SIGSOFT '08/FSE-16. DOI: 10.1145/1453101.1453131. ISBN 978-1-59593-995-1.

[23] Ullmann, J. R. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*. New York, NY, USA: Association for Computing Machinery. January 1976, vol. 23, no. 1, p. 31–42. DOI: 10.1145/321921.321925. ISSN 0004-5411.

Appendix A

Contents of the Attached Medium

The most notable directories on the attached memory medium are the following:

- `/diffkemp/`
  - `/diffkemp/` – Source codes of DiffKemp.
  - `/tests/` – Automated tests used by DiffKemp.
- `/experiments/` – Results of experiments and the patterns used to generate them.
- `/tex/` – LaTeX source codes of this thesis.
- `/init.sh` – Project initialisation script.
- `/README.txt` – Readme file containing the manual for compilation and execution.
- `/xsilli01_bp.pdf` – This thesis in PDF.

The source codes of the pattern matching extension of DiffKemp proposed in this thesis can be found in the `/diffkemp/diffkemp/simpll` directory. The modules that contain the extension are listed below.

- `InstPatternComparator` – Matching of instruction patterns.
- `PatternComparator` – Top-level pattern matching controller.
- `PatternSet` – Set of loaded LLVM IR patterns.
- `ValuePatternComparator` – Matching of value patterns.

The regression tests created for the extension (as well as the original regression tests) are located in the `/diffkemp/tests/regression` directory. The relevant files and subdirectories are the following:

- `patterns/` – Required LLVM IR patterns and pattern configuration files.
- `rhel-80-81-patterns.yaml` – Specification of tests for RHEL versions 8.0 and 8.1.
- `rhel-81-82-patterns.yaml` – Specification of tests for RHEL versions 8.1 and 8.2.
- `rhel-82-83-patterns.yaml` – Specification of tests for RHEL versions 8.2 and 8.3.

Appendix B

Compilation and Execution

This appendix describes how to compile the project and execute both the evaluation experiments and pattern matching regression tests. It assumes that the working directory contains all files and directories from the attached memory medium (contents of which are outlined in Appendix A). It should also be noted that since multiple kernels of Red Hat Enterprise Linux (RHEL) have to be downloaded, prepared, compiled and compared, the whole process may be rather time-consuming and requires a large amount of disk space, as well as a stable internet connection.

Compilation and Kernel Preparation

First, the dependencies of DiffKemp and `rhel-kernel-get`[1] have to be installed. These are presented in the `README.md` files placed in their respective public GitHub repositories. To summarize, the following dependencies and packages are required:

- Kernel build dependencies: `gcc`, `make`, `bison`, `flex`, `libelf-dev`, `libssl-dev`.
- Archivation utilities: `cpio`, `tar`, `xz`, `bzip2`.
- Clang and LLVM for development (to run the experiments, versions 9, 10 or 11 should be used; otherwise, versions 5, 6, 7 and 8 are also supported).
- CScope, CMake and the Ninja build system.
- Python 3 for development and with CFFI.
- Python packages from `diffkemp/requirements.txt`.
- The `progressbar` Python package (can be installed automatically by the included initialisation script as well).

Second, the included `init.sh` initialisation script can be used to download and prepare the necessary versions of the kernel of RHEL 8 or, if not on an internal Red Hat network, the equivalent versions of the CentOS kernel. The script might require superuser privileges since it needs to use `pip`. The kernel preparation can be performed by the following command:

```
./init.sh kernels
```

[1] `rhel-kernel-get`—an open-source tool for automatic downloading and preparing of Linux kernels—https://github.com/viktormalik/rhel-kernel-get.

After the necessary versions of the kernel of RHEL 8 are placed in the `diffkemp/kernel` directory, DiffKemp should be compiled into the `diffkemp/bin` directory using the same initialisation script:

```
./init.sh compile
```

Finally, DiffKemp has to translate the KABIs of the prepared kernel versions into LLVM IR (i.e., it has to generate so-called kernel snapshots). The following command should generate the required snapshots into the `diffkemp/snapshots` directory:

```
./init.sh snapshots
```

Since the compilation process is quite complicated and requires many dependencies, DiffKemp also provides a Docker container image that contains all of the necessary prerequisites. The image can be initialised by running the `run-container.sh` script from the `diffkemp/docker/diffkemp-devel` directory.

Experiments and Tests

Contrary to the initialisation phase, the commands for evaluation and regression testing expect to be executed from the top-most `diffkemp` directory.

The evaluation experiments have to be run for each pair of the prepared kernel snapshots separately. During each experiment, the snapshots should be first compared without patterns. Then, the comparison should be repeated with the corresponding pattern configuration file. The following commands execute the experiments:

```
# RHEL 8.0 vs 8.1 without patterns
bin/diffkemp compare snapshots/linux-4.18.0-80.el8 \
snapshots/linux-4.18.0-147.el8 --report-stat --stdout

# RHEL 8.0 vs 8.1 with patterns
bin/diffkemp compare snapshots/linux-4.18.0-80.el8 \
snapshots/linux-4.18.0-147.el8 --report-stat --stdout \
--pattern ../experiments/rhel-80-81-config.yaml

# RHEL 8.1 vs 8.2 without patterns
bin/diffkemp compare snapshots/linux-4.18.0-147.el8 \
snapshots/linux-4.18.0-193.el8 --report-stat --stdout

# RHEL 8.1 vs 8.2 with patterns
bin/diffkemp compare snapshots/linux-4.18.0-147.el8 \
snapshots/linux-4.18.0-193.el8 --report-stat --stdout \
--pattern ../experiments/rhel-81-82-config.yaml

# RHEL 8.2 vs 8.3 without patterns
bin/diffkemp compare snapshots/linux-4.18.0-193.el8 \
snapshots/linux-4.18.0-240.el8 --report-stat --stdout

# RHEL 8.2 vs 8.3 with patterns
bin/diffkemp compare snapshots/linux-4.18.0-193.el8 \
snapshots/linux-4.18.0-240.el8 --report-stat --stdout \
--pattern ../experiments/rhel-82-83-config.yaml
```

The pattern matching regression tests can be executed using the following command:

```
pytest tests -k patterns
```