BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

GENERATING CODE CHANGE PATTERNS FROM C

GENEROVANÍ ŠABLON ZMĚN KÓDU V JAZYCE C

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

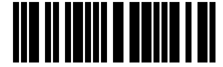
AUTHOR **Bc. TOMÁŠ KUČMA**
AUTOR PRÁCE

SUPERVISOR **Ing. VIKTOR MALÍK**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment



157087

Institut: Department of Intelligent Systems (DITS)
Student: **Kučma Tomáš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Mathematical Methods
Title: **Generating Code Change Patterns from C**
Category: Software analysis and testing
Academic year: 2023/24

Assignment:

1. Get acquainted with DiffKemp, a tool for static analysis of semantic differences between versions of large-scale C projects.
2. Investigate DiffKemp's approach to handling user-defined patterns of changes that should be considered semantically equal (so-called custom change patterns).
3. Propose a way to define custom change patterns by describing the correspodning changes in the C language.
4. Implement the proposed solution in a way such that the defined patterns can be used by DiffKemp.
5. Evaluate the created solution by implementing at least 5 custom change patterns in C and demonstrating that they help DiffKemp eliminate false positive results.

Literature:

- Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 329–339. IEEE (2021)
- Malík, V., Šilling, P., and Vojnar, T.: Applying Custom Patterns in Semantic Equality Analysis. In: The 10th Edition of the International Conference on NETworked sYStems (NETYS). pp. 265-282. Springer (2022)

Requirements for the semestral defence:
The first two points of the assignment.

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

Supervisor: **Malík Viktor, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: 1.11.2023
Submission deadline: 17.5.2024
Approval date: 6.11.2023

Abstract

Ensuring the semantic stability of software projects is often a costly task. DiffKemp is a tool that automatizes this process, with a special emphasis placed on performance and usability in large-scale projects. A trade-off for its efficiency is a greater degree of inaccuracy compared to formal tools. To minimize this issue, DiffKemp allows users to define their own semantics-preserving patterns, describing what kind of changes are to be treated as equal. Currently, this support is restricted to patterns written in LLVM internal representation, which is not a user-friendly language. The purpose of this work is to extend this capability to patterns written in C, significantly simplifying the process of their creation. This includes a proposal of a representation of the patterns, which must be able to encode all necessary meta-information, and subsequent design, implementation, and testing of a DiffKemp extension that allows utilization of patterns encoded in C.

Abstrakt

Zabezpečenie sémantickej stability softvérových projektov je často nákladnou úlohou. Nástroj DiffKemp automatizuje tento proces so špeciálnym dôrazom na výkon a použiteľnosť v rozsiahlych projektoch. Cenou za jeho efektívnosť je väčšia nepresnosť oproti formálnym nástrojom. Na minimalizáciu tohto problému DiffKemp umožňuje používateľom definovať vlastné vzory zachovávajúce sémantiku, opisujúce, aké zmeny majú byť považované za ekvivalentné. V súčasnosti je táto podpora obmedzená na vzory napísané v internej reprezentácii LLVM, ktorá nie je priateľská pre používateľa. Cieľom tejto práce je rozšíriť túto podporu na vzory napísané v jazyku C, čo výrazne zjednoduší proces ich vytvárania. To zahŕňa návrh reprezentácie vzorov, ktorá musí byť schopná zakódovať všetky potrebné metainformácie, a následný návrh, implementáciu a testovanie rozšírenia DiffKemp, ktoré umožní využívať vzory zapísané v jazyku C.

Keywords

code change patterns, DiffKemp, semantic analysis, refactorization

Kľúčové slová

vzory zmien kódu, DiffKemp, sémantická analýza, refaktorizácia

Reference

KUČMA, Tomáš. *Generating Code Change Patterns from C*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík

Rozšírený abstrakt

Pri vývoji softvéru je dôležitým aspektom nezavádzať do softvéru neúmyselné zmeny. Zamedzuje sa tým zbytočným chybám, zaisťuje sa, že rozhrania fungujú tak ako to od nich užívatelia očakávajú, a podobne. Preto je veľmi dôležité, aby aj pri vývoji a aktualizáciách zostávali niektoré časti sémanticky stabilné. Zároveň je ale v niektorých prípadoch náročné túto požiadavku zabezpečovať aj pri vysokom počte úprav a rozsiahlejších refaktorizáciách. Z toho dôvodu boli na túto úlohu vyvinuté automatizované nástroje, ktoré zachovanie sémantickej stability overujú pomocou statickej analýzy zdrojového kódu. Fungujú tak, že pre dve verzie softvérového projektu porovnávajú sémantiku programov, napríklad jednotlivých funkcií, a o výsledku následne informujú užívateľa. Jedným z takýchto nástrojov je otvorený nástroj DiffKemp, vyvíjaný primárne firmou Red Hat.

DiffKemp sa od ostatných nástrojov v tejto oblasti líši tým, že je zameraný na vysokú škálovateľnosť. Bežný prístup pozostáva z používania formálnych metód, vďaka čomu býva dosiahnutá vysoká spoľahlivosť, ale časové náklady rastú prudko s veľkosťou kódu, a teda v praxi je ho väčšinou možné použiť iba na malé izolované funkcie. Aby DiffKemp bolo možné používať aj na projekty veľkosti napríklad Linuxového jadra, bolo potrebné zvoliť iný prístup. Skúmaný kód sa prevedie do internej reprezentácie LLVM, nízkoúrovňového jazyka podobného strojovému kódu, vykonajú sa určité analýzy a zjednodušujúce transformácie, a potom samotné porovnávanie prebieha zväčša na báze jednotlivých dvojíc inštrukcií. Takto je dosiahnuté omnoho lepšie škálovanie, avšak za cenu väčšej nepresnosti. DiffKemp je navrhnutý tak, aby sémantickú ekvivalenciu porovnávaných verzií hlásil iba vtedy, ak je skutočne potvrdená. Nepresnosti vo vyhodnocovaní sa preto (okrem výnimiek) týkajú toho, že pri zložitejšej refaktorizácií DiffKemp hlási rozdiel v sémantike, aj keď sú obe verzie ekvivalentné. Napriek tomu má DiffKemp veľký prínos pre šetrenie času a zdrojov, aj zvyšovanie kvality softvéru, keďže výraznú časť potenciálnych zmien v sémantike vie skontrolovať a spracovať úspešne.

Aby sa nedostatky tohto nástroja ešte viac limitovali, pre užívateľov existuje možnosť zadefinovať si vlastné vzory zmien v kóde. Užívatelia nimi špecifikujú, že určitá zmena má byť považovaná za sémantiku-zachovávajúcu zmenu. Okrem toho, že užívateľ má možnosť takto pokryť konkrétne prípady, o ktorých vie, že DiffKemp sám o sebe ich nevie správne spracovať, môže navyše takéto vzory vytvoriť aj pre zmeny, ktoré sémantiku menia, ale úmyselne, lebo sa napríklad týkajú bezpečnostných opráv.

Problém s týmto prístupom bol v tom, že tieto vzory zmien bolo nutné písať v LLVM internej reprezentácií, ktorú využíva interne aj DiffKemp. Nie je to užívateľsky prívetivý jazyk, keďže je primárne navrhnutý na automatické analýzy a transformácie kódu, a tiež nie je tak bežný, aby bol s ním a jeho špecifikami štandardný užívateľ dobre oboznámený. To vytvára výraznú bariéru pri používaní tejto funkcionality nástroja.

Preto vznikla táto práca, ktorej cieľom bolo rozšíriť tento systém o možnosť používať vzory zmien napísané v jazyku C. V prvom rade bolo teda potrebné navrhnúť konkrétny formát a spôsob zápisu vzorov zmien v jazyku C. Ďalej bolo treba navrhnúť rozšírenie nástroja DiffKemp aby takéto vzory podporoval. Keďže už obsahuje celú funkcionalitu na spracovanie vzorov v internej reprezentácií LLVM a keďže C je možné previesť do tohto jazyka pomocou prekladača Clang, prirodzene bol zvolený prístup, kedy vzory v C sú prevedené na formát už podporovaný. Vďaka tomu možno znovu využiť celú už existujúcu časť nástroja zodpovednú za správne využívanie poskytnutých užívateľských zdrojov pri porovnávaniach verzií projektov.

Navrhnuté rozšírenie bolo implementované. Pre zápis samotných vzorov bol vytvorený hlavičkový súbor, ktorý poskytuje rozhranie pre definovanie vzorov zmien pomocou makier

jazyka C. Ďalej bol vytvorený spracovávajúci prechod — takzvaný *pass* — pre generované vzory, keďže samotný preklad z C do LLVM internej reprezentácie nevedie na validný vzor zmien. Tento prechod preložený vzor spracuje tak, aby bol valídny. Tieto veci boli zakomponované do nástroja, aj s úpravami rozhrania a iných relevantných aspektov, aby vzory v jazyku C boli plne podporované nástrojom DiffKemp. Vďaka tomu pribudla možnosť používanie týchto vzorov pri porovnávaní, aj možnosť vykonať len jednoduchý preklad vzoru z jazyka C do valídnej reprezentácie LLVM, ktorý potom môže užívateľ prípadne ďalej upraviť ručne. Táto implementácia je už pridaná do voľne dostupného repozitáru projektu DiffKemp.

Rozšírenie bolo otestované vo viacerých ohľadoch. Prvý sa týkal základnej funkcionality a tieto testy skúmali správne správanie pri jednoduchých konštrukciách jazyka C. Druhá sada testov replikovala už existujúce testy nad reálnymi príkladmi z praxe, tento krát ale využívajúc novo podporované kódovanie vzorov v jazyku C, miesto tých pôvodných. Obe množiny testov dopadli úspešne. Výstupom tejto práce teda je rozšírenie, ktoré výrazne zjednodušuje prácu s nástrojom DiffKemp, konkrétne v ohľade písania a využívania užívateľsky definovaných vzorov zmien.

Generating Code Change Patterns from C

Declaration

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Viktora Malíka. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Tomáš Kučma
May 15, 2024

Acknowledgements

Rád by som sa poďakoval vedúcemu Ing. Viktorovi Malíkovi za podporu, vedenie a rady v súvislosti s touto diplomovou prácou.

Contents

1 Introduction **4**

2 Semantic Equivalence Analysis with DiffKemp **6**
2.1 Semantic Equivalence Analysis State of the Art 6
2.2 LLVM Internal Representation 7
2.3 DiffKemp Architecture 9

3 Code Change Patterns **12**
3.1 Built-in Code Change Patterns 14
3.2 Custom Code Change Patterns 15

4 Design of DiffKemp extension for code change pattern encoded in C **21**
4.1 Proposed form of code change patterns encoded in C 21
4.2 Generating code change patterns from patterns encoded in C in DiffKemp 23

5 Implementation of DiffKemp extension for custom C patterns **27**
5.1 Header for pattern definition 27
5.2 Pass for LLVM IR patterns generated from C 30
5.3 Interface for patterns encoded in C 33

6 Evaluation of patterns for DiffKemp written in C **35**
6.1 Basic C pattern tests 35
6.2 Replicating existing custom LLVM IR pattern tests with C-encoded patterns 36

7 Conclusion **41**

Bibliography **43**

A Contents of the included storage media **46**

B Header file for defining custom patterns **47**

List of Figures

2.1 An example of a function written in C and its LLVM IR version compiled with Clang, optimization level 1. This function calculates the sum of the elements of an integer array. The ϕ (`phi`) instruction characteristic for the Static Single-Assignment form is used multiple times. 8
2.2 Control Flow Graph representation of the LLVM function in Figure 2.1. . . 9

3.1 Example of refactoring *while into for* written in the language C. 13
3.2 Value pattern representing change to a |1UL « |`NR_PAGEFLAGS` kernel value, due to change to macro value. 16
3.3 Instruction pattern representing change of kernel function call from `cleanup-_srcu_struct` to `cleanup_srcu_struct_quiesced`, requiring specific preceding context. Auxiliary structure definitions and function declarations are not included. 17

4.1 Example of an LLVM IR instruction pattern without an output mapping and its encoding in C using the proposed form. 22
4.2 Example of an LLVM IR instruction pattern with output mapping and its encoding in C using the proposed form. Macro definitions in C excluded for length. 23
4.3 Example of an LLVM IR value pattern and its enconding in C using the proposed form. 24
4.4 Diagram showing the architecture of DiffKemp and its behavior with the proposed extension when compiling patterns in C. What is not explicitly shown is that compilation and related processing passes are not directly performed by the Python module. Instead, the module uses installed `clang` and `opt` tools. 25

5.1 Example of a basic instruction pattern encoded in C using the implemented header file. This pattern describes both a change in parameters used to call a function and an addition of another function call. For more examples, see Chapter 6. 28
5.2 The workflow of the C pattern pass. Green and red boxes mark changes. . . 30
5.3 Example of a YAML pattern configuration file, loading five C encoded patterns, with one specific pattern being compiled with additional `-O0` Clang compiler option, to compile it at optimization level 0 instead of the default level 1. Failure to parse pattern is specified to lead to an error. 34

6.1 Pattern adding a call. This specific pattern does not currently work properly, because of how DiffKemp internally handles mapping of values between compared modules and patterns. 36
6.2 User-defined pattern written in C specifying semantic equivalence between two versions of used kernel masking macro. 38
6.3 User-defined pattern written in C specifying semantic equivalence between two versions of inlined assembly code. 40
6.4 User-defined pattern written in C for a value, where an older version of used macro mask had to manually defined as it is not possible to include both the original and updated version. 40

Chapter 1

Introduction

In our daily lives, software plays an ever-expanding role, influencing everything from commerce to medicine. However, this dependence on software also means that if something goes wrong, the consequences might be severe. Potential security vulnerabilities may lead to unauthorized access or leakage of private data. Minor bugs can cost organizations substantial amounts of dollars in both financial losses and reputational damage. A mistake in one system can trigger a cascade of failures in other connected systems, creating a domino effect that amplifies the impact of the initial error.

One of the ways such problems may occur is when unintended changes are introduced into a program. Even a simple refactorization can actually change semantics in a nuanced and unexpected way. That is why ensuring semantic stability is a must in some projects. However, carefully reviewing all changes is labor intensive and often requires significant expertise or knowledge of the modified software, while still leaving the possibility of having some changes slip through the cracks.

Recognizing the potential consequences, there is a growing demand for automated tools to analyze and prevent such software-related problems. One of such tools is the framework DiffKemp. This program allows users to semantically compare different versions of programs written in the C language, on a function-by-function basis. DiffKemp employs static code analysis, an aspect shared by many other programs with similar purposes. However, what sets DiffKemp apart from most tools is its chosen approach. Traditionally used formal methods are very accurate, but time-wise they struggle even with moderately sized functions. This means that they are practically unusable for large-scale projects. DiffKemp on the other hand, was designed for projects of enormous sizes, such as the Linux kernel.

To handle projects of this scale, a different approach was needed. Since most of the codebase, after one or few refactorizations are performed, will likely remain the same, it is possible to save a lot of time by performing simple one-to-one instruction comparisons. However, the C code is not suitable for such comparisons. Therefore, DiffKemp first converts the code into an intermediate representation used by the LLVM project and the Clang compiler, which is better suited for analysis. Afterward, various transformations and simplifications are applied to the code, which has the effect of converting some semantics-preserving changes into identical representations. However, that is still not enough to handle all of the possible refactorizations. During the comparison itself, DiffKemp might additionally explicitly try to seek relocated instructions, inline function calls, or handle inverted conditions with switched `if-else` branches.

Due to this, DiffKemp is capable of comparing two versions of a Linux kernel in ten minutes with significant accuracy. Although the possibility of false positives exists –— indicating incorrectly identified inequalities —– DiffKemp effectively minimizes false negatives, except in the case of implementation bugs. Although complete accuracy is not achieved, limiting the potential semantic changes that need to be reviewed to only a fraction of the original changes still significantly increases the efficiency and quality of the code.

One of the reasons for false positives is that it is not practically feasible for DiffKemp to account for all possible semantics-preserving changes. To address this issue, DiffKemp was extended to support user-defined patterns. Users can provide code patterns that describe changes to the code that are to be treated as semantically equal by DiffKemp. This finds use not only in handling false positives but also when a change that modifies semantics is intended. For example, a security fix will by definition change the program semantics, but it is not interesting for the developer to see these changes when trying to avoid unintended changes.

Currently, these patterns have to be provided in a representation using the aforementioned intermediate code. This representation is great for compilers and similar tools because its properties are fit for analyses and transformations, but it is not user-friendly. It is lower level compared to the C language, contains various atypical instructions, the single assignment rule is used, functions can often have hundreds of lines, and so on. A person without expertise in working with this language will not be able to write a complicated pattern without investing a lot of time.

The purpose of this work is to extend DiffKemp to support user-written patterns in the C language. Users will be able to directly take the code of the refactorization performed and with only small modifications rewrite it into a DiffKemp compatible C pattern. This will greatly streamline the process.

This work is divided into several chapters. Chapter 2 serves as a detailed introduction to DiffKemp and its inner workings, along with a basic introduction to semantic analysis and comparisons with other similar tools. The next chapter, number 3, dives further into the subject DiffKemp's pattern system, and code changes in general. The subsequent chapter, 4, proposes a pattern encoding in C and how DiffKemp can be extended to support these patterns. It is followed by Chapter 5 that contains details on the implementation of the design. The final chapter, 6, presents examples of written patterns and evaluates the implemented extension on both artificial and real world examples.

Chapter 2

Semantic Equivalence Analysis with DiffKemp

DiffKemp [18] is a tool used to semantically compare functions written in the C language. It was originally developed by Red Hat for their Red Hat Enterprise Linux kernel [26] (RHEL). RHEL provides a set of functions and symbols called *Kernel Application Binary Interface* [23] (kABI for short). These functions and symbols are used by drivers and other kernel modules to interface with the kernel. Because of this, it is important that the functions behave as their users expect them to behave, and therefore Red Hat guarantees that the semantics of these functions will not change across minor versions, with the exception of security fixes. Using DiffKemp to compare updated versions of functions with original versions, many of the refactorizations performed can be easily handled and only a fraction of them need a further manual review. However, DiffKemp has not been developed purely for the Red Hat Linux kernel. It is open source software and can be used for any C project in general using the `make` build system.

The first section of this chapter — 2.1 — provides an introduction to static analysis of semantic equivalence, lists other similar tools, and describes the differences in the approach utilized. Next, in Section 2.2, the LLVM intermediate representation is introduced, as understanding it is instrumental in understanding how DiffKemp itself works. Section 2.3 describes the implementation and architecture of DiffKemp. Information is taken primarily from [18], unless otherwise noted.

2.1 Semantic Equivalence Analysis State of the Art

Program analysis is the study of the properties of computer programs [12]. It is usually divided into two disciplines. *Dynamic analysis* is performed by executing the program and observing its behavior. *Static analysis* is performed without execution. Dynamic analysis is the simpler of the two — it is only necessary to provide the inputs and then compare the result with the expected outputs, or monitor a certain aspect of the behavior (e.g. memory access). However, it is difficult to use it to examine the behavior of the program in a comprehensive way. For example, to determine whether two functions are semantically equivalent, one might need to test every possible input and compare the results. On the other hand, static analysis requires an explicit analysis of the semantics of the instructions but can be used to provide general answers about the system as a whole.

The field of semantic comparisons of programs has been researched for years, and thus there are multiple tools that provide the ability to compare semantics of functions. One of them is a tool called LLReve [5]. It was designed to automate *regression verification* in C projects. The goal is to determine whether two versions of the same program behave identically or to determine the specific differences in their behavior. To this end, LLReve tries to infer predicates that mathematically describe the relationship between these two versions. *Horn clauses* are used to represent the verification conditions, which are then solved using *model checking* [12] techniques. In practice, Horn clauses are solved with a *Satisfiability Module Theory* (SMT for short) solver such as Z3 [19] developed by Microsoft.

SymDiff [13] (**Sym**bolic **Diff**) is another tool developed for this purpose. In this case, the tool is language-agnostic. Specifically, it operates on an intermediate verification language *Boogie* [3], for which translators from various languages, such as C, C# or x86, exist. SymDiff operates on two loop-free programs (loops can be unrolled to a specific depth or transformed into tail-recursive functions). In the intermediate language, a new function is created, which executes both compared functions sequentially, each from an identical state, while storing the results and the global state. The function ends with the assertion of equality of the results and global state. The generated code is then transformed using *Boogie Modular Verifier* [3] into a verification condition, and as with LLReve, the solver Z3 is used.

Most of such tools work in a similar way. They rely on formal methods that often guarantee soundness and completeness. However, the problem is with their performance. Publication [18] showed that when comparing LLReve and DiffKemp, both of which use LLVM intermediate representation, on thirty selected functions with a 30 second timeout per each, DiffKemp processed all functions within the time limit and all except two were correct. LLReve timed out on every function except one, where the result was identical to the one determined by DiffKemp. These results show that the use of formal methods for large-scale projects is not feasible.

2.2 LLVM Internal Representation

The LLVM Compiler Infrastructure [15] is a project whose purpose is to provide a collection of various modular, reusable compiler and toolchain technologies. It was originally developed as a research project of the University of Illinois and is available under *Apache-2.0 with LLVM-exception license* [27]. Today, LLVM consists of a large number of subprojects, some of the most notable being *LLVM Core*, libraries that provide language-agnostic optimization and code generation support for various CPUs, the *Clang* compiler for C / C++ languages, the *LLDB* debugger, and the implementation of the standard C++ library *libc++* . LLVM finds use in many different commercial and open source projects.

The fundamental aspect of LLVM is its intermediate representation code, also called the LLVM IR. It is a low-level representation with only slightly more abstraction compared to standard assembly languages. It can be used to represent various high-level languages. A very important attribute of LLVM IR is the use of the *Static Single-Assignment* (SSA for short) form [1]. This means that every assignment is into a variable with a distinct name. However, we might want to use the same variable name on two different control flow paths. To account for this situation while following the SSA rule, each of these variables is given a different name. The value of these variables after the paths rejoin is represented by a new variable defined by an assignment with a ϕ instruction. This instruction can return

```c
int foo(int *data, int len) {
    int sum = 0;
    for (int i = 0; i < len; i++) {
        sum += data[i];
    }
    return sum;
}
```

```llvm
define i32 @foo(ptr nocapture noundef readonly %0, i32 noundef %1) {
  %3 = icmp sgt i32 %1, 0
  br i1 %3, label %4, label %6
4:                                                ; preds = %2
  %5 = zext i32 %1 to i64
  br label %8
6:                                                ; preds = %8, %2
  %7 = phi i32 [ 0, %2 ], [ %13, %8 ]
  ret i32 %7
8:                                                ; preds = %4, %8
  %9 = phi i64 [ 0, %4 ], [ %14, %8 ]
  %10 = phi i32 [ 0, %4 ], [ %13, %8 ]
  %11 = getelementptr i32, ptr %0, i64 %9
  %12 = load i32, ptr %11, align 4, !tbaa !4
  %13 = add i32 %12, %10
  %14 = add nuw nsw i64 %9, 1
  %15 = icmp eq i64 %14, %5
  br i1 %15, label %6, label %8, !llvm.loop !8
}
```

Figure 2.1: An example of a function written in C and its LLVM IR version compiled with Clang, optimization level 1. This function calculates the sum of the elements of an integer array. The ϕ (`phi`) instruction characteristic for the Static Single-Assignment form is used multiple times.

different values based on the path previously taken, solving the problem of branching in the SSA form. LLVM IR also contains type system that is language-independent.

These aspects, along with the tools provided by the LLVM project for the purposes of working with the LLVM IR, make it suitable for static analysis and various code transformations. This can be seen by looking at some examples that use it — for example, BIN2LLVM [11] converts binary code to LLVM IR for the purpose of detecting and identifying cryptographic routines, and NUMBA [14] is an LLVM-based Just-in-Time compiler for Python. DIFFKEMP uses LLVM-IR for similar reasons.

LLVM IR represents functions in the form of *Control Flow Graphs* [2]. Control Flow Graph is a directed graph consisting of nodes called *basic blocks*. Basic block consists of a sequence of instructions that are always executed sequentially. Jumping is possible only from the end of a basic block and only to the beginning of a basic block. An example of

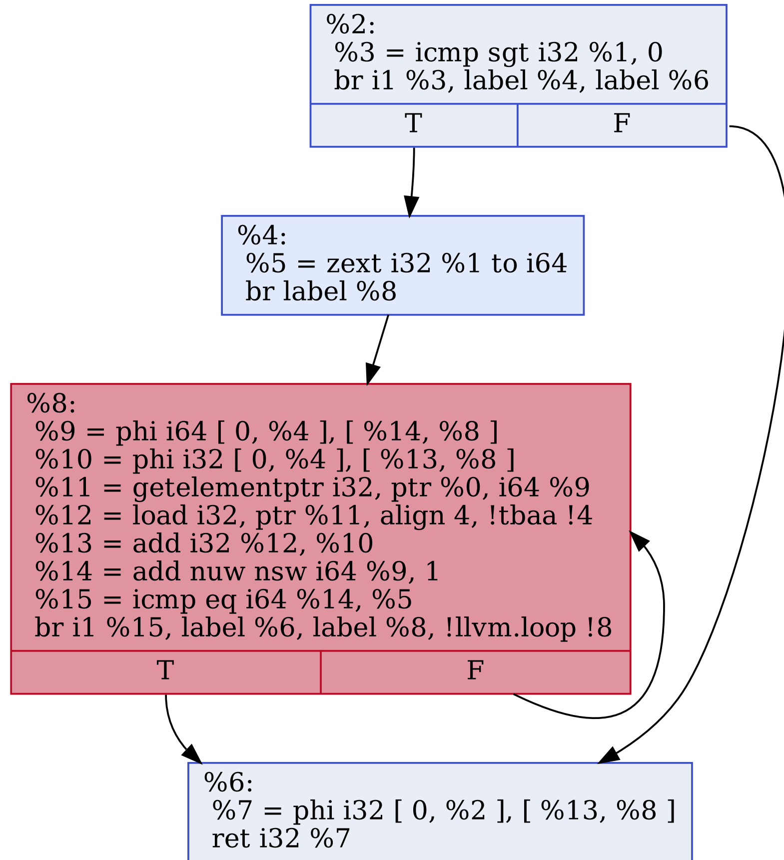


Figure 2.2: Control Flow Graph representation of the LLVM function in Figure 2.1.

a C function and its LLVM IR version can be seen in Figure 2.1. The Control Flow Graph representation of this function can be seen in Figure 2.2.

2.3 DiffKemp Architecture

DiffKemp works in two separate stages. First, a *snapshot* must be created for each version of the compared project. Afterwards, it is possible to compare functions of any two snapshots for semantic equivalence.

The purpose of the snapshot stage is to prepare the chosen version of the project for further analysis. Parts of the source code that are necessary for the comparison are compiled into LLVM IR code. During this, several standard compiler optimizations are performed. This alone has the capacity to resolving some of the refactorizations by transforming different code snippets with the same semantics into identical LLVM IR code.

The two generated snapshots can be then compared. First, all LLVM source files used for the specified comparison are loaded by DiffKemp, so that they can be processed using the API of the official LLVM library. Preprocessing is performed using so-called LLVM

passes [25]. They can be written for functions, modules, loops, regions, or call graphs. These passes are standardly used for compiler optimization, but the LLVM library provides developers with the option to define their own passes. DiffKemp uses this functionality to perform various analyses that provide information for later use — such as analysis of all potentially called functions (transitively) by a given function — and transformations that help unify different, but semantically equivalent code snippets — for example, removing unused return values.

After the preprocessing is done, it is possible to proceed directly to the comparison itself. The basic structure of the comparison is very simple and occurs in a hierarchical manner. Comparisons of functions lead to comparisons of basic blocks; those are performed by comparing individual instructions, which can lead to operation and operand comparisons, and so on. Ultimately, this can lead to a new comparison of different functions in the case of function calls. The algorithm describing this behavior can be seen in Algorithm 2.3.1. In the algorithm, *synchronization points* primarily represent individual instructions. The synchronization points of the compared versions are mapped to each other using *smap*. The algorithm also utilizes a mapping between variables called *varmap*. The algorithm tries to sequentially compare instructions, alternatively applying a pattern if possible, while updating and checking the described mappings. This approach is computationally very cheap and effective.

Some of the possible refactorizations cannot be effectively handled in practice using only preprocessing transformations. Therefore, DiffKemp has explicit built-in support for various *semantics-preserving change patterns*. This includes changes such as inlining function call (or the opposite), relocating blocks of code (assuming doing so does not break any data), inverting condition and switching `if-else` branches, and so on. Furthermore, DiffKemp includes support for user-defined patterns. These patterns have to be written in the LLVM IR code, and they use special debugging annotations. It is not necessary for these patterns to describe semantically equivalent changes, which can be useful in cases such as fixing security vulnerabilities. A deeper dive into the inner workings of the built-in and user patterns is provided in Chapter 3.

From an implementation point of view, DiffKemp mainly uses the C++ and Python languages. The core of the tool is a C++ library called `SimpLL`. This library is responsible for preprocessing and comparisons. The main comparison functionality is handled by `DifferentialFunctionComparator`, which inherits functionality from LLVM library's `FunctionComparator`, but extends it to work more generally. The preprocessing passes are located in a separate subfolder `Passes`. In addition, the library includes various utilities, such as a debugging logger and support for user-defined patterns. The rest of the DiffKemp's functionality is implemented in Python. A collection of Python scripts, utilizing the Simpll library and various LLVM command line tools (Clang, Opt, etc.), packaged together forms the Python library `diffkemp`. The DiffKemp executable is a simple Python file that uses this library. The project supports LLVM versions from 9 to 16 (LLVM 17 [21] is the newest LLVM version at the time of writing this thesis). Each LLVM version uses slightly different representations, which can lead to slightly different results or minor changes in performance.

Input: Functions f_1, f_2
Result: `true` if f_1 and f_2 are semantically equal, `false` otherwise

1. preprocess f_1 and f_2
2. **if** $|P_1| \neq |P_2|$ **then**
3.   **return** `false`
4. $S_1 = \{i^1_{\text{in}}\}$, $S_2 = \{i^2_{\text{in}}\}$
5. $smap(i^1_{\text{in}}) = i^2_{\text{in}}$
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. $varmap(g_1) = g_2 \in G_2$, such that g_1 has the same name as g_2

`// Main loop`

10. $Q = \{(i^1_{\text{in}}, i^2_{\text{in}})\}$
11. **while** Q *is not empty* **do**
12.   take any (s_1, s_2) from Q
13. $p = detectPattern(s_1, s_2)$
14.   **for** *each pair* $s'_1, s'_2 \in succPair_p(s_1, s_2)$ **do**
15.   **if** $(s'_1 \in S_1 \lor s'_2 \in S_2)$ **then**
16.   **if** $smap(s'_1) \neq s'_2$ **then**
17.   **return** `false`
18.   **else**
19.   **continue**
20.   **if** p *is* `none` **then**
21. $equal = cmpInst(s_1, s_2)$
22.   **else**
23. $equal = compare_p((s_1, s'_1), (s_2, s'_2))$
24.   **if** $\neg equal$ **then**
25.   **return** `false`

        `// Update synchronisation sets and maps`

26. $S_1 = S_1 \cup \{s^1_1\}$, $S_2 = S_2 \cup \{s^1_2\}$, $smap(s'_1) = s'_2$
27.   `update` $varmap$ `according to` p
28.   insert (s'_1, s'_2) to Q
29. **return** `true`

Algorithm 2.3.1: Algorithm used to check a pair of functions f_1 and f_2 for semantic equivalence, adapted from [18]. The algorithm is simplified for illustration purposes. The sets P_i denote the parameters of the compared functions. The sets S_i denote the mapped synchronization points of the function i, initialized to contain their first instruction. *smap* denotes synchronization point mapping and *varmap* variable mapping. Successor synchronization points are generally understood to be the next instruction, in the case of pattern the instruction following the snippet that matched the pattern, and in the case of branches all of the entry points of the successor basic blocks.

Chapter 3

Code Change Patterns

This chapter delves deeper into the issues of code change patterns. In this work, the term *code change patterns* is used to talk about patterns that describe a certain kind of relation between pairs of code, provided that they fit the given pattern. In a narrower sense, we understand them as pairs of parameterized segments of code, each of them having a list of input and output variables, with mapping existing between these variables. A specific focus is placed on *semantics-preserving* code change patterns, meaning that the behavior described by the two segments is semantically identical. Therefore, if each of the mapped pairs of input variables is set to an identical value, the mapped pairs of output variables will also have an identical value.

These kinds of pattern are useful for describing various standard refactorizations in particular. The work [6] proposes a list of 29 refactorings in four categories, specifically for the C language. These categories are as follows:

1. Adding a Program Entity:
   (a) Add a variable
   (b) Add a parameter to a function
   (c) Add a typedef definition encapsulating an existing type
   (d) Add a field to a structure
   (e) Add a pointer to a variable
2. Deleting a Program Entity:
   (a) Delete unused variable
   (b) Delete unused parameter
   (c) Delete a function
3. Changing a Program Entity:
   (a) Rename variable
   (b) Rename constant
   (c) Rename user-defined type
   (d) Rename structure field
   (e) Rename function

```c
//...
int i = 0;
while (i < len) {
    sum += data[i];
    i++;
}
//... (i is not used)
```

```c
//...
for (int i = 0; i < len; i++) {
    sum += data[i]
}
//...
```

Figure 3.1: Example of refactoring *while into for* written in the language C.

   (f) Replace the type of a program entity
   (g) Contract variable scope
   (h) Extend variable scope
   (i) Replace value with constant
   (j) Replace expression with variable
   (k) Convert variable to pointer
   (l) Convert pointer to direct variable access
   (m) Convert global variable into parameter
   (n) Reorder function arguments

4. Complex refactorings:
   (a) Group a set of variables in a new structure.
   (b) Extract function
   (c) Inline function
   (d) Consolidate conditional expression
   (e) For into while
   (f) While into for
   (g) While into do while

An example of one of these refactorings can be seen in Figure 3.1. These refactorings form the basis for the semantics-preserving changes considered.

Section 3.1 describes how code change patterns with built-in support are handled in DiffKemp, as described by [18]. The next section — 3.2 — includes a formal definition for user-defined code change patterns, describes how these user-defined DiffKemp patterns can be written, and how DiffKemp handles user-defined patterns implementation-wise.

3.1 Built-in Code Change Patterns

The approach used by DiffKemp, specifically the performed preprocessing transformations and the way comparisons are handled, gives the tool the capability to handle many of the refactorizations described in the introduction to this chapter. However, that is not the case for all of them. Additionally, other kinds of refactorization were identified that were present in the examined in the Linux kernel repository, which were not included in the provided list, and also require individual approach. These two sources of unhandled refactorings were used as a springboard for the DiffKemp's built-in pattern support.

DiffKemp provides native support for the following kinds of changes:

- *Changes in Structure Data Types*
  - *Changed Offset of a Structure Field*
  - *Different Ways to Access the Same Field*
- *Moving Code into Functions*
- *Changes in Enumeration Values*
- *Changes in Source Code Location*
- *Inverse Branch Conditions*
- *Code Relocations*

Internally, DiffKemp does not use a standardized interface for these kinds of changes. Instead, each of them is implemented individually, directly in the source code used for comparisons. However, handling of these situations is always conditional, based on the used configuration, and so the user can pick and choose which changes get supported and which not.

Changed Offset of a Structure Field is a type of change that occurs when some of the fields in structure change order, type, get removed, or added. Even an unmodified field may in this case move to a different offset within the structure, which leads to a different operands when using LLVM's `GEP` (*get element pointer*) function, which is used to access structure field. To address this type of change, DiffKemp considers both the name of the field contained in the debugging information and the offset value. If both match, the equality holds. If only the name has changed, but the new structure does not contain any field with the original name, DiffKemp assumes that the structure field has been renamed, and equality still holds. If neither match, the compared instructions are determined to be unequal. The most complicated case is when only the name is identical. If no pointer arithmetic is performed on the pointers calculated as a result of the `GEP` instruction, the instructions are considered equal. In the other case, the absolute value of the offsets matters, and thus the instructions cannot be considered equal.

Different Ways to Access the Same Field change applies in situations where, for example, a structure field was nested into a union within the structure. This may lead to two different sequences of `GEP` instructions in the old and the new code, with their input — source pointer used in the first instruction — being identical and having only single output — variable containing the pointer calculated by the last instruction of the sequence. If such a sequence is detected and all index operands are constants, DiffKemp uses this knowledge

to calculate the final access offset. If the offsets match, the sequences are considered equal, and the resulting outputs are mapped to each other.

Moving Code into Functions is a common refactorization that improves the readability of the code. In DiffKemp, this kind of change is handled by inlining called functions into the code. This is not done automatically, but lazily. Specifically, if a comparison of an instruction fails, with at least one of the instructions being a call, the call(s) gets inlined. Afterwards, additional preprocessing is performed to eliminate dead code and propagate constants. This ensures that if, for example, the newly defined function is more general than the original code, but this particular call passes arguments that ensure identical behavior, comparison after inlining will be successful. When this is done, the function comparison process can run again. This may occur multiple times during a single function comparison.

Changes in Enumeration Values occur, when enumeration values are added or removed. The underlying value of enumeration constant changes, but generally the way enumerations are used does not rely on the specific numerical values they are assigned. If enumeration constants are used in both instructions, their name is retrieved using associated debugging information, and comparison is performed by comparing their names, not values.

Changes in Source Code Location concerns Linux kernel specifically. It contains various warning functions, where the specific message is not considered important. Often these functions also contain information about file and line of the source code, where the call is located, which is also not relevant to the semantics of the code. For these specific instructions, operands representing the aforementioned values are ignored during comparisons.

Inverse Branch Conditions are those, where the condition was inverted and the branches were swapped. To handle this, in cases where a compared pair of comparison instructions evaluates opposite value and the result is used only as branching condition, the instructions are considered equal and the order of branch instruction successors is swapped.

Code Relocations — perhaps the most general type of change, but also one that is most difficult to handle. DiffKemp specifically supports relocations anywhere within sequential code, as long as there is no dependency between the relocated code and the code skipped by the relocation. In the cases where instruction comparison fails, *relocation detection* begins. For either of the compared instructions, a matching instruction is searched in the unprocessed sequence of instructions on the other side of the comparison. If this is successful, the skipped code block is marked as *potentially relocated* and the comparison continues from the point where the successful comparison was performed. Later, it is necessary to match the potentially relocated block. This is called *relocation matching* and an attempt to match it happens if another instruction pair is compared as unequal. It is necessary for all potentially relocated blocks to be matched for the function comparison to be resolved as equality. After matching the block, another phase follows — *relocation checking*. It consists of checking, whether there is a data dependency between the relocated code and the skipped code, in other words, whether one block writes to a variable and another reads from the same variable. There being no data dependency between these blocks is another necessary condition for returning a result indicating equality.

3.2 Custom Code Change Patterns

DiffKemp was extended to support user-defined custom code change patterns in 2021, as part of the bachelor's thesis [28]. The thesis introduces two kinds of code change patterns, namely *instruction patterns* and *value patterns*. Instruction patterns are more general and consist of sequences of instructions. However, these patterns can become quite large.

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.

Table 3.1: List of the metadata symbols used for defining custom code change patterns, along with their description. Adapted from [28].

```
define i64 @diffkemp.old.NR_PAGEFLAGS() {
  ret i64 33554431
}

define i64 @diffkemp.new.NR_PAGEFLAGS() {
  ret i64 67108863
}
```

Figure 3.2: Value pattern representing change to a `1UL << NR_PAGEFLAGS` kernel value, due to change to macro value.

Furthermore, when the change concerns only a specific value, to address all possible instructions in which this value can appear, one would need to create multiple instruction patterns. Value pattern is a special pattern variant designed to address this problem. They allow the user to simply define two values that are meant to be treated as semantically equivalent, regardless of context. Internally, they are implemented using instruction patterns.

Formally, these patterns are defined using *parametrized control flow graphs* [17]. The parameterized control flow graph c is a triple:

$$c = (in, cfg, out)$$

Here, cfg is a control flow graph that uses undefined variables and types — parameters. The set of these parameters, labeled in, is the input to the control flow graph. out denotes the set of outputs, that is, the variables that can be used outside of the control flow graph. Using this definition of parametrized control flow graphs, we can define the code change pattern as a quadruple:

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

$c_o = (in_o, cfg_o, out_o)$ and $c_n = (in_n, cfg_n, out_n)$ are parametrized control flow graphs representing the old and the new versions of the code, respectively. $imap : in_o \leftrightarrow in_n$ is a bijective function that maps old inputs to new inputs and vice versa. In the same way,

```
define void @diffkemp.old.free_user(%struct.kref*) {
  %2 = bitcast %struct.kref* %0 to i8*
  %3 = getelementptr i8, i8* %2, i64 -50456
  %4 = bitcast i8* %3 to %struct.ipmi_user*
  %5 = getelementptr inbounds %struct.ipmi_user, %struct.ipmi_user* %4,
  ↪  i32 0, i32 2
  call void @diffkemp.old.cleanup_srcu_struct(%struct.srcu_struct* %5),
  ↪  !diffkemp.pattern !pattern-start
  ret void, !diffkemp.pattern !pattern-end
}

define void @diffkemp.new.free_user(%struct.kref*) {
  %2 = bitcast %struct.kref* %0 to i8*
  %3 = getelementptr i8, i8* %2, i64 -50456
  %4 = bitcast i8* %3 to %struct.ipmi_user*
  %5 = getelementptr inbounds %struct.ipmi_user, %struct.ipmi_user* %4,
  ↪  i32 0, i32 2
  call void
  ↪  @diffkemp.new.cleanup_srcu_struct_quiesced(%struct.srcu_struct* %5),
  ↪  !diffkemp.pattern !pattern-start
  ret void, !diffkemp.pattern !pattern-end
}
```

Figure 3.3: Instruction pattern representing change of kernel function call from `cleanup_srcu_struct` to `cleanup_srcu_struct_quiesced`, requiring specific preceding context. Auxiliary structure definitions and function declarations are not included.

$omap : out_o \leftrightarrow out_n$ is a bijective function that describes the mapping between the old outputs to the new outputs and vice versa.

3.2.1 Pattern Representation

Custom patterns, as they are currently implemented, must be written in LLVM IR. This representation was a natural choice for this purpose, because it matches the actual representation of programs used by DiffKemp for comparison. Each pattern control flow graph is described by a specially named function, with its parameters denoting the inputs. These function pairs are prefixed with `diffkemp.old` and `diffkemp.new`, respectively. The outputs are denoted using a special `diffkemp.mapping` function, called just before the function exit. It is also possible to use parametrized types by prefixing them with `diffkemp.type`. The input and output mappings are inferred automatically, based on the order of the input/output variables provided. Furthermore, to ensure compatibility with the chosen pattern matching algorithm and to provide additional information, it is necessary to use special metadata symbols. These symbols are listed in Table 3.1. To represent a value pattern, the two functions have to contain a single instruction, returning the old/new value.

An example of a value pattern can be seen in Figure 3.2. An example of an instruction pattern can be seen in Figure 3.3.

3.2.2 Pattern Matching

Because the number of user-loaded patterns is not directly limited, trying to apply the pattern on every instruction would be ineffective. For this reason, attempts to apply dynamically loaded user-defined patterns occur only after a comparison of two instructions fails, but before actually ending the function comparison with `false` result. For this reason, it is necessary to use the metadata symbol `pattern-start` specified in the previous subsection when defining patterns to mark the first pair of differing instructions.

The top-level instruction matching algorithm can be seen in Algorithm 3.2.1. For each user-defined pattern, DiffKemp attempts to match the two parameterized control flow graphs with the old and new versions of the code, starting from the failed instruction pair. If the pattern matches successfully, a mapping is created between the program variables and the input/output pattern variables. Then, it is checked whether each pair of input variables, as defined by *imap*, is also mapped in the program according to *varmap* (as defined in Algorithm 2.3.1). Finally, for each output, *varmap* and *snmap* are updated to map the pairs of program variables used as output together. All instructions that were resolved using this pattern and in this way determined to be equal are returned as a result of this comparison.

Input: (i_o, i_n): pair of differing instructions
smap and *varmap*: as defined in Algorithm 2.3.1
P_i: set of available instruction patterns

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

```
1  for (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 for (i^o_c, i^o_m) ∈ imatch_o do
6              i^n_c = imap(i^o_c) if varmap(i^o_m) ≠ imatch_n(i^n_c) then
7                  valid = false
8                  break
9          if ¬valid then
10             continue
           // Synchronize outputs
11         for (o^o_c, o^o_m) ∈ omatch_o do
12             o^n_c = omap(o^o_c)
13             smap(o^o_m) = omatch_n(o^n_c)
14             varmap(o^o_m) = omatch_n(o^n_c)
15         return M_o ∪ M_n
16 return ∅
```

Algorithm 3.2.1: Top-level algorithm for matching instruction patterns, adapted from [28]. For $i \in \{o, n\}$, r_i indicates the result of CFG matching, $imatch_i/omatch_i$ mapping between pattern inputs/outputs (respectively) and the code variables, and M_i is the set of instructions matched by the pattern. Code fragment matching performed by $matchCFG$ is further described by Algorithm 3.2.2.

The matching of parametrized control flow graphs of a certain pattern with a code fragment of a compared program can be seen in Algorithm 3.2.2. It proceeds somewhat similarly to a standard DiffKemp comparison, however, with certain differences. A local mapping $varmap_c$ between the pattern and program variables is initialized with shared global variables. Next, a working set Q, initially containing the first differing instruction of the program and the first differing instruction of the pattern, labeled with `pattern-start`, is continually processed, until it is empty. Processing consists of attempting to match the instruction pair, then on success updating the mapping between the program variables and input/output variables of the pattern based on the operands, updating the local $varmap_c$ mapping with the used instruction, and finally marking the matched instruction as processed. Whether the instruction match was successful or not, its successors are added to the working set Q. If all pattern instructions are matched, one more check must be performed.

Input: $c = (in, cfg, out)$: pattern code fragment, where i_c^b is the instruction tagged with `pattern-start`
i_p^d: differing instruction from one of the programs

Result: A tuple of (r_x, $imatch_x$, $omatch_x$, M_x), where
r_x: `true` if pattern matched, `false` otherwise
$imatch_x$: mapping between pattern inputs and code variables
$omatch_x$: mapping between pattern outputs and code variables
M_x: set of matched instructions

matchCFG (i_p^d, c)*:*

1 $M = \emptyset$
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 **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.`
6   **for** $k \in \{1, 2, ..., n\}$ **do**
7   **if** $o_k^c \in in(c)$ **then**
8                 $imatch(o_k^c) = o_k^p$
9   **if** $i_c \in out(c)$ **then**
10             $omatch(i_c) = i_p$
11         $M = M \cup \{i_p\}$
12         $varmap_c(i_c) = i_p$
    `// Queue up the following instruction pair`
13   **for** $(i_c', i_p') \in succInstPair(i_c, i_p)$ **do**
14   insert (i_c', i_p') into Q
15 **if** *all instructions in c have been matched* $\wedge$ $checkContext(match_x, ctx_x, in_x)$ **then**
16   **return** $(\texttt{true}, imatch, omatch, M)$
17 **return** $(\texttt{false}, imatch, omatch, M)$

Algorithm 3.2.2: Algorithm for matching custom pattern control flow graphs to code fragments, adapted from [17] and [28]. Note that $succInstPair(i_c, i_p)$ may return a successor pair $(i_c, succ(i_p))$. This is called *instruction skipping* and is performed when the current pair did not match successfully.

That is, whether the so-called *context* of the pattern, meaning unchanged instructions at the beginning of the pattern, before the instruction tagged with `pattern-start`, is correct. Only then is the match resolved as successful.

Value patterns are handled by converting them to instruction patterns. This is done lazily, before each top-level matching algorithm evaluation. For each value pattern, an instruction pattern is generated for the particular pair of unequal program instructions, if possible.

Chapter 4

Design of DiffKemp extension for code change pattern encoded in C

This chapter discusses the design of the DiffKemp extension using code change patterns generated from patterns written in C. Section 4.1 specifies the proposed form of the patterns written in C, such that they are compatible with the representation of code change patterns used natively by DiffKemp. The subsequent chapter — 4.2 — contains information on how support for such patterns can be incorporated into DiffKemp.

DiffKemp currently utilizes LLVM IR for representation of both the compared projects and the custom patterns. Compilation from C to LLVM IR is natively provided by the Clang compiler (which is a part of LLVM project) since it utilizes LLVM IR as its own intermediate representation. Therefore, it is natural choice to exploit these facts, by choosing LLVM IR as the target representation for code change patterns generated from C, and by performing the compilation of C code into LLVM IR using the Clang compiler. However, because of the specific requirements placed on the encoding of code change patterns in LLVM, this alone is not sufficient, and additional processing is be required.

4.1 Proposed form of code change patterns encoded in C

This section concerns the form of code change patterns written in C. It is necessary for the proposal to allow the user to explicitly encode all the necessary information that is required from code change patterns encoded in LLVM IR, unless that information can be automatically inferred and generated from the provided pattern. As described previously in Section 3.2, each pattern must have a name, two bodies in the form of control flow graphs representing the old and the new code snippet, a list of inputs for each body and mapping between them, a list of outputs for each body and mapping between them, and metadata about the location of the start and end of the pattern for each body. Since the LLVM IR encoding uses functions for the representation, it is natural to use functions in C as well. This representation already includes the name and list of inputs by default. Mapping between inputs can be defined by their order, analogously to how it is done with LLVM IR patterns. For the list of outputs and mapping between them, a special mapping function can be defined, called at the end of the pattern function, again akin to the LLVM IR encoding. The function bodies in C are not explicitly encoded as control flow graphs, however, they are directly converted into these graphs when compiled to LLVM IR. The only significant problem is with encoding the metadata. One statement in C may be compiled into multiple

```
define void @diffkemp.old.free_user(%struct.kref*) {
  %2 = bitcast %struct.kref* %0 to i8*
  %3 = getelementptr i8, i8* %2, i64 -50456
  %4 = bitcast i8* %3 to %struct.ipmi_user*
  %5 = getelementptr inbounds %struct.ipmi_user, %struct.ipmi_user* %4,
  ↪  i32 0, i32 2
  call void @diffkemp.old.cleanup_srcu_struct(%struct.srcu_struct* %5),
  ↪  !diffkemp.pattern !pattern-start
  ret void, !diffkemp.pattern !pattern-end
}
define void @diffkemp.new.free_user(%struct.kref*) {
  %2 = bitcast %struct.kref* %0 to i8*
  %3 = getelementptr i8, i8* %2, i64 -50456
  %4 = bitcast i8* %3 to %struct.ipmi_user*
  %5 = getelementptr inbounds %struct.ipmi_user, %struct.ipmi_user* %4,
  ↪  i32 0, i32 2
  call void
  ↪  @diffkemp.new.cleanup_srcu_struct_quiesced(%struct.srcu_struct* %5),
  ↪  !diffkemp.pattern !pattern-start
  ret void, !diffkemp.pattern !pattern-end
}
```

```
void __pattern_old_free_user(struct kref *ref) {
    cleanup_srcu_struct(&user->release_barrier);
}
void __pattern_new_free_user(struct kref *ref) {
    cleanup_srcu_struct_quiesced(&user->release_barrier);
}
```

Figure 4.1: Example of an LLVM IR instruction pattern without an output mapping and its encoding in C using the proposed form.

instructions in LLVM IR. This is not transparent to the user, and so even if we consider ways of encoding metadata in C, such as pragmas (special compiler directives [7]), it is not possible to interact directly with a specific LLVM IR instruction. However, because of how the pattern start and pattern end are defined, we can add this metadata automatically using static analysis after the patterns have been compiled into the LLVM IR encoding. For value patterns, we can simply use pairs of functions that return the values that are meant to be evaluated as equal, again analogous to the LLVM IR value patterns.

One more thing that needs to be noted is that it is necessary to be able to differentiate between functions representing patterns and ordinary functions, as the former may utilize calls to the latter and therefore they may appear in the same context. Furthermore, mapping function must be clearly differentiable form either of those. It is also necessary to be able to clearly determine the existing relationship between a function representing the old body of a pattern and a function representing the new body of the same pattern. For these reasons, a special naming schema must be used. The plan is to allow users to define

```
define i64 @diffkemp.old.SWP_OFFSET_MASK(i64) {
  %2 = and i64 %0, 144115188075855871
  ret i64 %2
}

define i64 @diffkemp.new.SWP_OFFSET_MASK(i64) {
  %2 = and i64 %0, 288230376151711743
  ret i64 %2
}
```

```
void __pattern_old_SWP_OFFSET_MASK (int i) {
    __output_mapping(i & SWP_OFFSET_MASK_old);
}
void __pattern_new_SWP_OFFSET_MASK (int i) {
    __output_mapping(i & SWP_OFFSET_MASK_new);
}
```

Figure 4.2: Example of an LLVM IR instruction pattern with output mapping and its encoding in C using the proposed form. Macro definitions in C excluded for length.

these patterns in a simple standardized way through a macro-based interface, which will ensure that the naming schema is followed. However, the specifics of these aspects depend on implementation and are thus described in more detail in Chapter 5.

You can see examples of patterns written in C using the proposed form of encoding, along with their counterparts encoded in LLVM, in Figure 4.1, Figure 4.2, and Figure 4.3. The specific naming schema used in these examples is only for demonstration purposes.

4.2 Generating code change patterns from patterns encoded in C in DiffKemp

It is necessary to be able to generate equivalent code change patterns encoded in LLVM IR from those encoded in C in the proposed form from previous section. This section describes the approach chosen to perform this generation.

DiffKemp, as written in Section 2.3, consists of an internal SimpLL C++ library, that is utilized by a Python module `diffkemp` providing interface between the library, its users, and other external tools. As was mentioned previously, for the compilation to LLVM IR itself, the Clang compiler can be used. It is a self contained tool that requires no further development work. Clang is one of the external tools already used by the Python module, and so naturally, from architecture perspective, compilation of patterns to LLVM IR can also be handled by the Python module. Clang is currently used specifically for compilation of the compared projects into snapshots. Clang offers various levels of optimization and optimization passes, and since the instructions from patterns have to successfully map onto the instructions of compared snapshots, the optimization level and passes used when compiling patterns should be same as when compiling projects into snapshots.

```
define i64 @diffkemp.old.NR_PAGEFLAGS() {
  ret i64 33554431
}

define i64 @diffkemp.new.NR_PAGEFLAGS() {
  ret i64 67108863
}
```

```c
unsigned long __old_pattern_NR_PAGEFLAGS() {
    return (1UL << 25) - 1;
}

unsigned long __new_pattern_NR_PAGEFLAGS() {
    return (1UL << 26) - 1;
}
```

Figure 4.3: Example of an LLVM IR value pattern and its enconding in C using the proposed form.

Patterns compiled to LLVM IR still need to be processed into a correct form. First of all, it is necessary to modify the function names from the naming scheme used in C encoding to those used in LLVM IR encoding. That is because C does not allow the usage of structured naming of functions or variables using ".", so following the naming scheme used by the LLVM IR patterns is not possible. Therefore, the naming scheme used in C must be different. Renaming itself can be done quite easily, as it consists of simply replacing the chosen prefixes and names with different prefixes and names. Furthermore, the SimpLL library contains additional passes for preprocessing compiled projects, and just like before, these passes need to be used for patterns too, so that their instructions can successfully map to the instructions of the compared snapshots. Finally, metadata signifying pattern start and pattern end must be added. Pattern end can be determined quite simply, as it is generally the last or next to last instruction, either return, or call to the mapping function, if it is used. However, determining the start of the pattern is more complicated. It is defined as the instruction of the first non-matching instruction pair, where the match is determined by DiffKemp, specifically the SimpLL library comparison algorithm. Quite naturally, using this library to compare the old and new code segments of the pattern will yield the necessary information.

These actions require utilization of the LLVM API and the SimpLL library itself, so in this work the additional processing of the patterns is designed to be performed by the C++ library itself. This processing is to happen after loading the LLVM IR pattern generated from C, before proceeding with the comparison itself. Based on the specified requirements, the pattern processing of LLVM IR patterns compiled from C consists of these specific steps:

1. Determine all pattern pairs in the loaded pattern LLVM module. This can be done simply by matching function names with based on the chosen C pattern naming scheme.

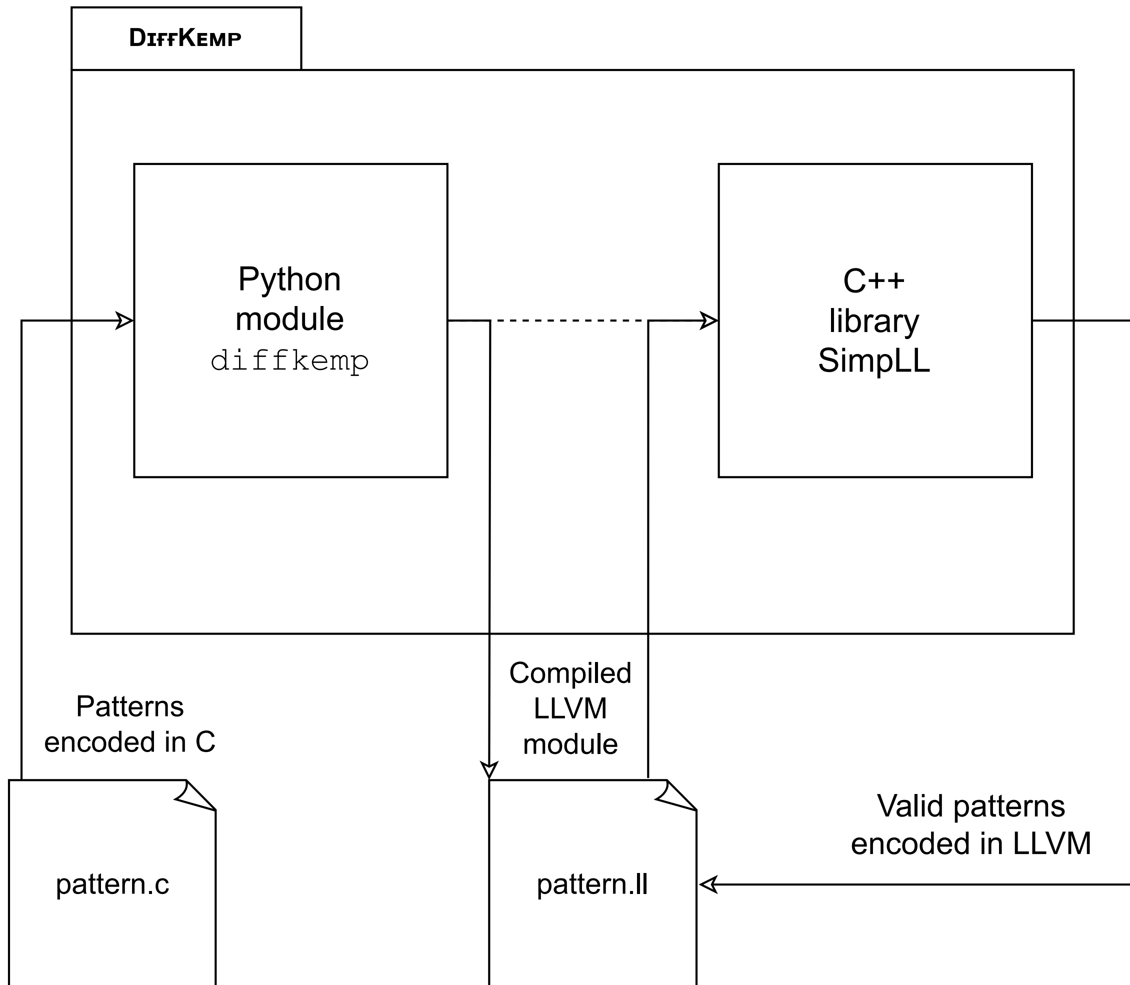


Figure 4.4: Diagram showing the architecture of DiffKemp and its behavior with the proposed extension when compiling patterns in C. What is not explicitly shown is that compilation and related processing passes are not directly performed by the Python module. Instead, the module uses installed `clang` and `opt` tools.

2. Rename functions to proper naming used in the LLVM IR encoding. Analogous to finding patterns, it consists simply of replacing pattern prefixes used in C with prefixes used in LLVM IR, and renaming the mapping function to required name.

3. Adding metadata to pattern instructions. The end of the pattern can be determined individually for each function. If we define *pattern body* of a function as every instruction except return and call to the mapping function, each instruction that is not part of the *pattern body*, but follows an instruction that is, can be marked as pattern end. This approach accounts for even more complicated cases of pattern control flow graphs. Function starts need to be determined for the pattern pairs as a whole. For each pattern pair, it is necessary to perform an entire comparison of the old and new functions of the code change pattern. For any reasonable pattern, this comparison should be resolved as unequal; otherwise, the pattern is useless. After the result is evaluated to inequality, the pair of instructions that could not be matched and caused the unequal result are the first differing instructions. These can be labeled as the pat-

tern start. Furthermore, as part of the comparison, multiple preprocessing passes that are used for compared projects are automatically applied to these compared pattern functions as well, and thus it is not necessary to run them manually beforehand.

This approach ensures that users will have the ability to utilize patterns seamlessly encoded in C with DiffKemp, identically to how they can use patterns written in LLVM IR. However, as mentioned previously, the compilation from C to LLVM IR is not entirely predictable and controllable by the user. There are multiple ways in which a given C construction may be represented using LLVM IR instructions. Sometimes, it is desirable for the user to perform a minor manual modification to the generated LLVM IR pattern module. In this way, a significant amount of work on the LLVM IR pattern is handled automatically by the Clang compiler and processing pass, while giving the user the precision of writing patterns that LLVM IR encoding provides. For this purpose, the design of the DiffKemp extension also includes a proposal of a new top-level command. In addition to building and comparing snapshots, compilation for non-LLVM IR patterns will be added. As considered in this thesis, this includes only patterns written in C, but it may be extended for other encodings in the future. Users will be able to directly compile and process patterns encoded in C, generating a DiffKemp compatible LLVM IR pattern with equivalent semantics. The way this fits within the current DiffKemp architecture can be seen in Figure 4.4.

Chapter 5

Implementation of DiffKemp extension for custom C patterns

This chapter provides information about the implementation of an extension for DiffKemp that utilizes the generation of code change patterns from C to allow users to use patterns with C encoding. The first section of the chapter covers the implementation of C header file, providing everything necessary for defining patterns through a macro-based interface. The following section contains information on a pass for processing patterns generated from C, implemented as part of the SimpLL library. The last section describes how DiffKemp as a whole was modified, utilizing everything from the previous sections, to natively support the patterns encoded in C and their compilation to LLVM IR. The basic idea is to compile the patterns to LLVM IR standardly using the Clang compiler, then these patterns can be processed with the SimpLL C pattern pass.

An important aspect of DiffKemp is its backwards compatibility with the older versions of the LLVM libraries that it uses. Specifically, at the moment, DiffKemp supports LLVM versions from 9 to 16. Because different major versions may not be backwards compatible, when developing DiffKemp, it is necessary to take care to explicitly ensure compatibility is maintained. Some of the LLVM library functions that were used in this implementation had been renamed, or the arguments they take have been changed, across the various LLVM library versions. This was addressed using conditional compilation in the DiffKemp source code, where chosen lines have multiple variants that get used based on the currently installed LLVM version. An additional significant change that occurred between the major LLVM versions was the change from explicitly typed pointers, where the type of the referred data is part of the pointer type, to the so-called *opaque pointers*, where all pointers use the same type `ptr` [22]. However, because C patterns are compiled into LLVM using Clang, which generally uses the same installed LLVM version as DiffKemp, the compiled patterns use the same kind of pointers as DiffKemp. In regards to this aspect, compatibility is thus automatically ensured without requiring any further effort, which is another advantage of writing patterns in C.

5.1 Header for pattern definition

To simplify and standardize the writing of patterns in C, the implementation includes a header file `diffkemp_patterns.h`, containing information, macros, and definitions for defining the patterns. The entire header can be seen in Appendix B.

```c
void console_flush_on_panic();
void panic_print_sys_info();

#define PATTERN_NAME panic_first
#define PATTERN_ARGS

PATTERN_OLD {
    console_flush_on_panic();
}

PATTERN_NEW {
    console_flush_on_panic(0);
    panic_print_sys_info();
}
```

Figure 5.1: Example of a basic instruction pattern encoded in C using the implemented header file. This pattern describes both a change in parameters used to call a function and an addition of another function call. For more examples, see Chapter 6.

Internal definitions

First, the header file contains internal definitions that determine the naming schema used, specifying the prefix for old and new pattern functions and the name of the mapping function. The used prefixes are `__diffkemp_old_` and `__diffkemp_new_`, the mapping function is named `__diffkemp_output_mapping`. Additionally, it contains a declaration of the mapping function and a definition of global variable `__diffkemp_is_cpattern` that can be used to determine whether an LLVM module is a compiled, unprocessed C pattern. Since the mapping function is expected to be called with various amounts and types of arguments, depending on the specific pattern, it is declared with an empty parameter list, which means that it can be called with any arguments [8]. In the new C23 standard, this is not supported. However, it allows declaring functions with variadic arguments — using "`...`" as the last argument to specify that arbitrary amounts and types of arguments may be used — without having to specify at least one mandatory parameter before the variadic parameters. In this use case, this serves an equivalent purpose [10], and so the declaration used for the output mapping function is chosen at the compile time, depending on the currently used C standard.

Interface for processing patterns

The next part of the header defines the interface for pattern processing, using a stringification macro to define string versions of the naming prefixes, the mapping name, and the name of the global indicator variable. This part of the interface is used by the SimpLL library itself, specifically the parts that are responsible for processing the loaded patterns originally written in C.

Interface for defining patterns

The final part of the header contains a macro-based user interface used directly to define patterns. For standard instruction patterns, `PATTERN_OLD` and `PATTERN_NEW` are used, with body of the pattern immediately following the macro. To allow the user to define the pattern name and the argument list for both versions in a single place, this information must be specified before defining the patterns by defining macro `PATTERN_NAME` and `PATTERN_ARGS` to the desired value. The patterns defined this way are functions with void return type, and the mapping is performed using `MAPPING(...)` macro call. An example of a basic instruction pattern encoded in C using this header can be seen in Figure 5.1

Sometimes it is desirable to specify pattern up to the point of resolution of a branching condition, but not include the branching itself. Defining an `if` statement while leaving the branches empty will lead to compiler optimizing the statement away. Using mapping on a logical statement would automatically resolve this statement as an integer, not a boolean value, due to the so-called *default argument promotions* in C [9]. For this reason, the header also specifies `CONDITION_PATTERN_OLD` and `CONDITION_PATTERN_NEW` macros, which work analogously, but have return type `_Bool` (native C boolean type). In this macro, the user can return the condition that is to be resolved, since returning a value is also a valid way of defining output mapping in the LLVM IR encoding. This solves the aforementioned case for patterns related to branching condition resolutions. Note that the promotion behavior in the mapping function may cause problems in general. For instance, floats will always be converted to doubles before being mapped, which may be undesired. Currently, the only workaround is to modify the header file used for the particular pattern by defining the mapping function with an explicit parameter list.

The header also contains a special macro for value patterns `VALUE_PATTERN(name, old_value, new_value)`. Internally, this defines a pattern function for both the old and the new value, each simply returning the given value, the return type of the pattern function being inferred from the type of the provided value using the compiler operator `__typeof__`.

Interface for defining pattern functions

In some cases, these macros are too restrictive. Additionally, if a function called from the old and new code segments has the same name, but different type signature, it is not possible to define such a function standard way. Both of these issues are addressed by `FUNCTION_OLD(name, ...)` and `FUNCTION_NEW(NAME, ...)` macros. These macros resolve to function name and list of arguments or parameters, with the name being modified to fit the used naming scheme. In this way, the user can declare, define, or call such a function while having the freedom to specify the return type or other details. If used for a function definition, since their names comply with the required names for patterns, they will be treated as patterns. Thanks to this, the user can for example define a pattern with different names of arguments (but the types still have to match), which can be useful in avoiding unnecessary renaming when copying code segment from the code, if the old and new code segments use differently named variables. Additionally, using this macro to declare and call functions solves the problem with different type signatures of identically named functions in old and new code patterns, as the actual names of the functions will be different. The macro has one more possible use case. With some patterns, it is desirable to import a header for various functions, types, and macros, to avoid having to define them manually. However, it may be undesirable for one of the included function definitions to be inlined by the compiler, replacing the call to the function with its body. Here, one

```
@__diffkemp_is_cpattern = global i32 1
void @__diffkemp_output_mapping(...)
@__diffkemp_old_swp_offset(i64) {
  %2 = and i64 %0, 144115188075855871
  call @__diffkemp_output_mapping(%2)
  ret void
}
@__diffkemp_new_swp_offset(i64) {
  %2 = and i64 %0, 288230376151711743
  call @__diffkemp_output_mapping(%2)
  ret void
}
```

(a) C pattern compiled into LLVM IR using Clang.

```
@__diffkemp_is_cpattern = global i32 1
void @diffkemp.output_mapping(...)
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 144115188075855871
  call @diffkemp.output_mapping(%2)
  ret void
}
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 288230376151711743
  call @diffkemp.output_mapping(%2)
  ret void
}
```

(b) Functions renamed to proper LLVM IR names.

```
@__diffkemp_is_cpattern = global i32 1
void @diffkemp.output_mapping(...)
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 144115188075855871,
  ↪ !diffkemp.pattern !0
  call @diffkemp.output_mapping(%2),
  ↪ !diffkemp.pattern !1
  ret void
}
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 288230376151711743,
  ↪ !diffkemp.pattern !0
  call @diffkemp.output_mapping(%2),
  ↪ !diffkemp.pattern !1
  ret void
}
!0 = !{!"pattern-start"}
!1 = !{!"pattern-end"}
```

(c) Pattern starts and ends tagged.

```
void @diffkemp.output_mapping(...)
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 144115188075855871,
  ↪ !diffkemp.pattern !0
  call @diffkemp.output_mapping(%2),
  ↪ !diffkemp.pattern !1
  ret void
}
@diffkemp.old.swp_offset(i64) {
  %2 = and i64 %0, 288230376151711743,
  ↪ !diffkemp.pattern !0
  call @diffkemp.output_mapping(%2),
  ↪ !diffkemp.pattern !1
  ret void
}
!0 = !{!"pattern-start"}
!1 = !{!"pattern-end"}
```

(d) C pattern indicator removed.

Figure 5.2: The workflow of the C pattern pass. Green and red boxes mark changes.

can use the `FUNCTION_OLD(name, ...)` macro to declare the function without providing the body. Then, if called from either pattern body using the macro, it cannot be inlined. However, it will still be considered identical to the original function when mapping the pattern instruction to the compared snapshot instructions based on its name, because in the LLVM IR encoding, DiffKemp specific prefixes are ignored in comparisons.

5.2 Pass for LLVM IR patterns generated from C

This section is dedicated to the pass responsible for processing LLVM IR encoded patterns generated from the C code. It is written in C++ as part of the SimpLL library. It is implemented as in the form of a class `CPatternPass`. The class contains one public function — `run(module)` — which applies the pass on an LLVM IR module loaded in the C++ LLVM API. Internally, the process has several parts, described later in this chapter. The diagram showing the processing performed by the pass is visible in Figure 5.2.

Originally, the pass was intended to be defined as an LLVM module pass [20], using the interface provided by the library. This would allow to include the compiled pass as a plugin for the LLVM `opt` tool — standardly used for running LLVM passes — making it possible to run the pass from the command line through the `opt` tool, directly from the Python interface. However, because the pass utilizes the SimpLL library, and both the SimpLL library and the opt tool use dynamically LLVM loaded libraries, but some of its functions are overwritten in SimpLL, this led to a problematic behavior and did not work correctly. That is why the described approach, where the pass is part of the SimpLL library, was taken instead.

Pass initialization

The first part, initialization, finds all the patterns in the given module. This is done by going through the list of functions, finding names that match the prefix used in C by old pattern code segments, and checking if a function with same name, but with prefix for the new code segment, exists in the module. If this is successful, the pattern pair is stored in a pattern map, with the pattern name being a key. Functions with declarations, but no definitions, are ignored, since these functions cannot be used as patterns and only serve as standard functions called from the patterns.

Renaming functions

Next, the detected patterns and mapping function are renamed to the naming scheme used by LLVM IR encoded patterns. Renaming is separated from the first pass through the list of functions in the initialization step. Changing the names in the middle of the process of finding patterns would make the process more difficult because the not yet renamed functions would be using a different naming scheme than the renamed ones.

Tagging pattern starts

The most important and complicated part is the pattern tagging. Pattern starts are tagged using the comparison functionality of the SimpLL library. First of all, it is not possible to compare functions within a single module. Doing so leads to memory error. Because of this, the tagged module is cloned using the LLVM IR API. Afterwards, each of the pattern pairs can be processed. The comparison configuration gets initialized with the original and cloned module and the names of the old and new pattern code segments. Then the library function for comparison is called. For this to work, the SimpLL library, specifically the result object provided, had to be modified. Previously, the pair of pointers to the first differing instructions found was stored in a `DifferentialFunctionComparator` class member, which is a class that performs the comparison itself. This class is not accessible to the caller of the library comparison function, and thus neither is the information about the first found differing instruction pair. After modification, the address of these instructions is stored in an new field `DifferingInstructions` of the function comparison result object, solving the problem.

It is important to note that tagging the differing instruction found would mean tagging the cloned version of the new pattern segment because the address of the first differing instruction found in this case belongs to the cloned module. It is possible to find a mapping between the memory addresses of the original and cloned instructions by iterating over the instructions of both simultaneously, since the modules are initially identical. This approach

solves the problem of tagging a cloned function instead of the original. However, there was one more problem. The transformations performed by passes during the preprocessing stage of the comparison are also applied only to the cloned version of the new pattern code segment. Running passes manually on the unmodified function would be both inefficient — such function essentially gets processed twice, once in the original module and once in the cloned one — and functionally problematic. For example, when inline assembly code is changed to an abstract function call, the function representing the assembly code is identical in both compared modules, provided that the assembly code is also considered identical. But that is not necessarily true when performing the passes individually within the same module. The same issues occur when performing multiple comparisons — the original old code segment with the cloned new segment and then vice versa — while being significantly more inefficient.

The way this is solved is by simply tagging the cloned version of a found differing instruction in the newer code change pattern and calling a newly added utility function which can clone the function from the module back to the original module. Cloning body of the function back can be done through LLVM library API, however, a mapping between global variables and arguments of the function must be provided. The global variables of the modules are mapped based on their names, as both modules are originally clones of each other, and the only possible differences stem from modifications that occurred during the mutual comparison of the modules. Arguments of the pattern function and its clone match, so their mapping is determined very easily, by simultaneous sequential iteration. For the tagging itself, an auxiliary function was written. For a given instruction, metadata kind — here specifically `diffkemp.pattern` — and a chosen value — here specifically `pattern-start` or `pattern-end` — the value is placed into the instruction metadata, under the specified kind. Doing so normally with the LLVM library would overwrite the value for the given kind in the given instruction, leading to incorrect behavior, for example if the pattern start and end are in the same place. Thus, within the helper function, the existing metadata values for the given kind are first copied into a buffer vector, then the new value is appended, and the vector is used as the new value, creating a structured metadata node.

Tagging pattern ends

The ends of the patterns are determined for each pattern code segment individually, as they do not depend on each other. However, they are determined using the concept of *pattern body*. To verify whether an instruction is part of the *pattern body*, auxiliary function `isPatternBody` was defined. The function returns true if the instruction is not a return, nor a call to the mapping function. For the tagging algorithm itself, each instruction of the pattern function is checked, and if the instruction is part of *pattern body*, has a successor, and the successor is not part of the pattern body, the successor is tagged as the pattern end. A special check is performed at the beginning — if even the first instruction is not part of the pattern body, this instruction gets tagged with the pattern end metadata tag.

Marking pattern as processed

Finally, after all this, the processing pass is finished. One last thing that remains is deleting the global variable indicator, which is used to signify that the module is an unprocessed LLVM IR pattern file generated from a C pattern. Deleting the indicator means that the pass will not have to be run next time this pattern is used.

5.3 Interface for patterns encoded in C

This section is dedicated to information about the implementation of modifications to the Python and SimpLL interface to enable the support of patterns encoded in C in DiffKemp, utilizing the implemented header for defining patterns and the C pattern pass.

Python module interface

The Python command line interface was extended to include a new command — `compile-pattern`. This command compiles and processes a non-LLVM IR encoded pattern into an LLVM IR encoded pattern. Currently, only C patterns are supported. It additionally provides options for appending options to the Clang compiler used for pattern compilation and for disabling processing (only performing the compilation step without running the C pattern pass). Since kernel source code is one of the main targets for DiffKemp, and because compiling code for kernels requires adding several include paths, included files, and defined macros, an option to automatically perform all these things for a chosen kernel source code path is also available. Finally, an option for disabling foreign function interface of SimpLL dynamic library and using SimpLL binary instead is provided as well — more information will be provided on this later. When not compiling patterns but standardly comparing snapshots, the C patterns can be provided in the same way as the LLVM patterns. Its compilation and processing are performed automatically. Additionally, the Clang append option and the kernel pattern option are both included for this command as well.

SimpLL library interface

The SimpLL library can be interfaced with in two ways. Normally, it is compiled to a dynamic library. This library can then be used from Python using the foreign function interface through `cffi` Python package, which provides the ability to call C and C++ functions from Python code [24]. SimpLL itself contains select functions that are made available through this interface. The second alternative is to use a compiled SimpLL binary. The binary is controlled through command line options. The FFI interface was extended with `preprocessPattern(const char *PatternPath)` function, that runs the C pattern processing pass. This function is called when using the pattern compilation command of DiffKemp. The main function of the binary was modified to include an additional `-preprocess-pattern-only pattern-path` option that serves the same purpose, when used instead of the foreign function interface.

Python module pattern handling

In the Python part of DiffKemp, the class responsible for handling custom patterns selected by the user is `CustomPatternConfig`. Originally, it only supported patterns with LLVM IR patterns with `.ll` extension or more complex pattern configuration in YAML [4] format. However, the pattern configuration only specified the LLVM IR patterns to load, making LLVM IR the only valid way to define patterns. The selected patterns were analyzed by the verification pass[1] through the LLVM `opt` tool, checking whether the format of the LLVM IR pattern module is well formed with respect to standard LLVM IR rules (however, it does not check whether it is a valid DiffKemp pattern). In this master's thesis, the class was extended to support patterns with `.c` extension. Such patterns are

[1] https://llvm.org/docs/Passes.html#verify-module-verifier

```yaml
on_parse_failure: ERROR
patterns:
 - tests/custom_patterns/c/rhel-81-82/__alloc_pages_nodemask.pattern.c
 - tests/custom_patterns/c/rhel-81-82/__put_task_struct.pattern.c
 - tests/custom_patterns/c/rhel-81-82/__stack_chk_fail.pattern.c
 - tests/custom_patterns/c/rhel-81-82/blk_mq_end_request.pattern.c
 - tests/custom_patterns/c/rhel-81-82/down_read.pattern.c
clang_append:
  tests/custom_patterns/c/rhel-81-82/blk_mq_end_request.pattern.c:
    - -O0
```

Figure 5.3: Example of a YAML pattern configuration file, loading five C encoded patterns, with one specific pattern being compiled with additional `-O0` Clang compiler option, to compile it at optimization level 0 instead of the default level 1. Failure to parse pattern is specified to lead to an error.

first compiled with the LLVM `clang` compiler, then processed with the `opt` tool using the same passes as those used to build snapshots. Additional clang options passed by the user and the options related to compiling kernel patterns are automatically appended to the clang call if they are provided.

SimpLL pattern handling

In the C++ SimpLL library itself, custom user-defined patterns are handled by `CustomPatternSet` class. The modifications to this class within this implementation were small. Since the patterns passed to the library are already compiled to LLVM IR, the only additional thing that is necessary is to check if the pattern is a compiled C pattern. This is done by checking the presence of the aforementioned global indicator variable. If so, the C pattern pass is used on the pattern module. Afterwards, the pattern is treated identically to any other LLVM IR pattern.

As mentioned above, the patterns can also be loaded from a YAML pattern configuration file. This file provides a list of patterns, all of which are loaded and subsequently simultaneously used for comparison. The user may also specify whether the failure to parse pattern should lead to an error or a warning. Each pattern file specified in the configuration YAML is individually handled identically to a directly loaded pattern file, so no further modifications to the process are required. However, because C patterns are compiled and thus it is sometimes desirable to specify additional compiler options for a specific pattern(s), the supported YAML format was modified. Now, in addition to patterns and behavior in the case of a parsing failure, the user can specify a new `clang-append` field, in which it is possible to define a list of additional clang options for each of the specified pattern files. An example of a YAML configuration file that shows this new functionality can be seen in Figure 5.3.

Chapter 6

Evaluation of patterns for DiffKemp written in C

This chapter is dedicated to experimentation, evaluation, and testing of the DiffKemp extensions implemented in the previous chapter. Here, one can also see examples of C patterns and how they look in practice. The first section concerns experiments with simple handwritten test cases, to verify basic functionality of the extension. The following section is dedicated to replicating pattern for RHEL kernel, which originally use LLVM IR encoded patterns, to evaluate the usability of C pattern encoding on real-life examples. The descriptions of tested patterns and the ways of testing them contained in this chapter also provide information on how C encoded patterns are written and include various details about the specifics of C patterns.

6.1 Basic C pattern tests

Handwritten test cases in this section represent basic C-code constructions and their modifications. Based on their results, it can be seen how the fundamentals of C-encoded patterns work and what implications it has on their usage in general. These tests represent the lowest-level benchmark for the usability of the extension.

Each test contains an old and a new modified source code example, plus a pattern that is meant to address the modification, leading to semantic equality result. In total, this test suite contains eleven individual tests.

The first group of tests concerns function calls. The patterns for adding or changing a call were simple and worked as expected. When changing the function type, while keeping the name, the `FUNCTION_OLD` and `FUNCTION_NEW` macros had to be used, but the test was successful. A problem occurred when adding a call processing a certain value and returning a value of the same type, while specifying a mapping between the original value and the value returned by the function in the new code segment, as visible in Figure 6.1. The problem was due to how the mapping currently works internally DiffKemp — the mapping is a bijection, a one-to-one relationship, as seen in Section 2.3 and Algorithm 2.3.1. However, the pattern would require mapping between both old `x` and new `x`, and between old `x` and `new_foo(x)` value, which is not currently possible. This is a known issue with DiffKemp and affects the functionality of the tool in other ways as well.

Testing on arithmetic expressions revealed some shortcomings. There are often many ways of expressing the same arithmetic expression using LLVM IR instructions, and many

```
#define PATTERN_NAME new_call_value
#define PATTERN_ARGS int x
int new_foo(int x);
PATTERN_OLD { MAPPING(x); }
PATTERN_NEW { MAPPING(new_foo(x)); }
```

Figure 6.1: Pattern adding a call. This specific pattern does not currently work properly, because of how DiffKemp internally handles mapping of values between compared modules and patterns.

valid orderings of these the instructions. Therefore, the generated code often varies based on various details of the compiled source code, and the code of patterns does not always match the code of the compared projects. However, simultaneously with this thesis, another thesis is being written on extending DiffKemp with the ability to determine the semantic equivalence of smaller groups of arithmetic and logic instructions based on the rules of distributivity, associativity, and so on. After that extension is implemented, it may be possible to use it for pattern mapping as well, solving the issue.

Testing a value pattern test defined with `VALUE_PATTERN` showed correct behavior. So did a condition-change pattern defined using the `CONDITION_PATTERN_OLD` and `CONDITION-_PATTERN_NEW` macros. A problem occurred with a specific pattern that contains an entire if branch. In this particular example, the compared basic blocks of the control flow graphs did not map correctly, similarly to the aforementioned mapping issue when adding a function call. However, in other examples of patterns that contain branches, the tests were successful, and thus there is not a fundamental problem with defining complex branching patterns.

A small amount of tests were written to cover the refactorizations described in the beginning of Chapter 3, specifically some of those that DiffKemp does not handle natively, nor can it without significant changes, as stated in [18]. Specifically, the tested refactorizations were *global variable into parameter* and *group a set of variables into a new structure*. The former was successful, while the latter was unsuccessful due to the previously described issue with mapping values in DiffKemp. In addition, two tests were written that replicate the functionality of built-in patterns from Section 3.1 — *Inverse Branch Conditions* and *Changes in Structure Data Types*. The tests were performed by disabling the specified built-in patterns and using the replacement custom patterns instead. Both were successful. The tests described in this paragraph show the wide usability of C-encoded patterns. However, note that the patterns for these situations have to be written on a case-by-case basis and cannot be written universally.

6.2 Replicating existing custom LLVM IR pattern tests with C-encoded patterns

As part of the thesis [28] that implements the original extension supporting user-defined custom patterns in LLVM IR, there were defined multiple tests for the functionality of these patterns. These tests were added to DiffKemp's set of regression test suite. Regression tests are tests that are run after code modifications, to verify that no new errors have been introduced [16]. In total, there are 3 groups of tests, one for each pair of Red Hat Enterprise

RHEL	Pattern file name	Pattern name	Type
8.0–8.1	ipmi_set_gets_events	free_user	I
	scnprintf	NR_PAGEFLAGS	V
	set_user_nice	enqueue_task	I
		dequeue_task	I
	vfree	__vunmap	I
	zap_vma_ptes	swp_offset	I
8.1–8.2	__alloc_pages_nodemask	zone_allows_reclaim	V
	__put_task_struct	__put_task_struct	I
	__stack_chk_fail	panic_first	I
		panic_second	I
	blk_mq_end_request	arch_atomic_dec	I
	down_read	__down_read	I
8.2–8.3	__put_page	__update_lru_size	I
	bio_endio	percpu_ref_put_many_first	I
		percpu_ref_put_many_second	I
	blk_execute_rq	blk_execute_rq_nowait	I
	kthread_create_on_node	__kthread_create_on_node	I
	sigprocmask	jobctl_stop_pending	V

Table 6.1: Table overview of all tests written for various versions of Red Hat Enterprise Linux. Pattern name suffix `.pattern.c` not shown to conserve space. In the pattern type column, I stands for the instruction pattern type and V for the value pattern type.

Linux versions from 8.0 to 8.3. Each group contains 5 patterns that are to be tested. The overview of these tests can be seen in Table 6.1.

For each test group, the test is specified using a YAML test specification file. The specification contains functions and symbols that are to be compared and the pattern file to use. The pattern file is always specified as a YAML pattern configuration file for the specific compared Red Hat Enterprise Linux version pair, loading all of the 5 patterns written for the pair. The test specification also describes what kind of results are expected from the comparisons after applying the patterns. In some cases, it is equality; in others, it is inequality, along with the specific semantic difference that is expected to be found.

Since tests are written for multiple versions of LLVM, there are multiple versions of each tested pattern. Older LLVM versions do not support the so-called *opaque pointers* [22], that is, pointers without a specified type of data to which they point. They use explicitly typed pointers. Therefore, each pattern has a version with opaque pointers and a version with explicit pointers, and which one is used depends on the version of LLVM used.

New patterns encoded in C and new pattern configuration files that utilize them were written for the purpose of this thesis. These files were based on the original LLVM IR patterns and configurations, addressing the same cases of detected semantic inequality. All three test specifications for groups testing specific Red Hat Enterprise Linux version pairs were duplicated from the LLVM IR test specification. The only modification to these new files was changing the path to the pattern configuration file, this time to the new pattern configuration files that load the newly written patterns written in C. With C-encoded patterns, it is not necessary to define two versions of each pattern. Each pattern that gets compiled and translated into LLVM IR using Clang utilizes the same installed LLVM

```
#include <linux/mm_types.h>
#include <linux/swap.h>
#include <linux/swapops.h>

#define RADIX_TREE_EXCEPTIONAL_SHIFT 2
#define SWP_TYPE_SHIFT_80                                          \
    ((sizeof(unsigned long) * 8)                                   \
     - (MAX_SWAPFILES_SHIFT + RADIX_TREE_EXCEPTIONAL_SHIFT))
#define SWP_OFFSET_MASK_80 (1UL << SWP_TYPE_SHIFT_80) - 1

#define PATTERN_NAME swp_offset
#define PATTERN_ARGS unsigned long i

PATTERN_OLD { MAPPING(i & SWP_OFFSET_MASK_80); }

PATTERN_NEW { MAPPING(i & SWP_OFFSET_MASK); }
```

Figure 6.2: User-defined pattern written in C specifying semantic equivalence between two versions of used kernel masking macro.

version that DiffKemp does, so their compatibility is ensured. This showcases yet another benefit of writing patterns in C.

To ensure that tests written for kernel patterns that include kernel headers function properly, the testing script was modified to add the option `-c-pattern-kernel-path path` to the used DiffKemp configuration. The provided kernel path is always the newer version of the compared kernel versions. That means that in cases where a used symbol or a called function was redefined between versions, it is necessary to additionally define the older version manually, as it is not possible to include it from the header. If the used version were the older one, a similar but opposite problem would occur in such situations.

Tests for Red Hat Enterprise Linux versions 8.0 and 8.1

The first set of tests, comparing Red Hat Enterprise Linux versions 8.0 and 8.1, contains one value pattern and four instruction pattern files. The value pattern is very simple, defined for two specific constants, and in C consists of only a single line:

```
VALUE_PATTERN(NR_PAGEFLAGS, (1UL << 25) - 1, (1UL << 26) - 1)
```

Three of the pattern files contain simple additions or modifications of a function call. One of these actually contains two separate patterns used for two related tested functions. The last instruction pattern concerns a masking operation, where the macro mask has been modified. The older version of the macro definition had to be copied into the pattern file manually. The pattern utilizes the mapping function. The resulting pattern file can be seen in Figure 6.2.

Tests for Red Hat Enterprise Linux versions 8.1 and 8.2

The second set of tests is written for Red Hat Enterprise Linux versions 8.1 and 8.2. This set also contains a single value pattern and four instruction pattern files. This time, the change in the value pattern concerns replacing a constant with a global variable. The pattern looks like this:

```
extern int node_reclaim_distance;
VALUE_PATTERN(zone_allows_reclaim, 30, &node_reclaim_distance);
```

Note that the equivalence must be defined between the constant and the pointer to the global variable, not the global variable itself. This is specified in the instructions for writing C patterns located in the header file to define the patterns. This is because of how external global variables are handled in LLVM IR, where a global value is defined through a pointer from which the value must be loaded before it is used. The problem with using the global variable itself is that it generates two instructions, one to load the value from the pointer, and the second using the actual value, here specifically the return. That means that the pattern is no longer a value pattern and does not behave as expected. Specifying equivalence with the pointer avoids this issue, while still matching correctly, because of how DiffKemp the matching.

In terms of the instruction pattern files, once again, one file contains two patterns. It is once again a simple pattern that adds new calls or assignments. A second instruction pattern handles a single call to a function that has been modified to have additional parameters. In this case, the functionality of the C pattern header file is utilized to define two variants of the same function. Beyond that, it is a simple call change pattern. Another pattern contains a change to the inlined assembly code. This test is one of the reasons why it is necessary to copy the compared pattern body from the cloned module to the original module, as stated in Section 5.2, and why the other mentioned approaches are not valid. It is necessary for the two inlined assembly code segments to be abstracted by the same global variable so that they are compared semantically equivalently, which happens only with the chosen approach. For demonstration purposes, this pattern can be seen in Figure 6.3.

The last instruction pattern in this set of tests is a complex if statement. Because the code segment contains multiple function calls that are declared and defined only in the implementation part of Linux source code and cannot be included, these functions have to be declared manually. However, because of how the patterns are compiled, it is only necessary to declare the function type and name, with empty parameter list, it is not necessary to determine the types and amount of the parameters of the called function manually, saving work for the developer. This generally applies for any pattern, except for the cases where it may lead to unwanted *default argument promotion* [9], where parameters have to be defined explicitly.

Tests for Red Hat Enterprise Linux versions 8.2 and 8.3

The third and final set is for versions 8.2 and 8.3. Once again, the set contains a single value pattern and four instruction pattern files. The value pattern specifies the equivalence between old and updated versions of a masked value as shown in Figure 6.4.

Once again, it was necessary to manually add an older version of the relevant macro, which is a slight imperfection of the user-defined C patterns system, as it is currently

```
#include <asm/alternative.h>

#define PATTERN_NAME arch_atomic_dec
#define PATTERN_ARGS atomic_t *v

PATTERN_OLD { asm volatile(LOCK_PREFIX "decl %0" : "+m"(v->counter)); }

PATTERN_NEW {
    asm volatile(LOCK_PREFIX "decl %0" : "+m"(v->counter)::"memory");
}
```

Figure 6.3: User-defined pattern written in C specifying semantic equivalence between two versions of inlined assembly code.

```
    #include <linux/sched/jobctl.h>

    #define JOBCTL_PENDING_MASK_82 (JOBCTL_STOP_PENDING |
    ↪   JOBCTL_TRAP_MASK)

    VALUE_PATTERN(jobctl_stop_pending,
                  JOBCTL_PENDING_MASK_82 | JOBCTL_TRAP_FREEZE,
                  JOBCTL_PENDING_MASK | JOBCTL_TRAP_FREEZE);
```

Figure 6.4: User-defined pattern written in C for a value, where an older version of used macro mask had to manually defined as it is not possible to include both the original and updated version.

implemented. Three of the four instruction pattern files once again contain very simple patterns, for example, modifying a single call, with one of the files containing two such patterns. The fourth pattern file was slightly more complex, additionally specifying a mapping for one output variable.

In sum, it was possible to successfully replicate all 16 original LLVM IR custom pattern tests with patterns encoded in C. This shows that despite the fact that encoding in C provides less direct control over the pattern, because of the used abstraction, it is still usable in wide variety of practical cases.

Chapter 7

Conclusion

This master's thesis concerns DiffKemp, an open-source tool used to check for semantic equivalence between pairs of functions written in the language C, using static analysis. The goal was to research this tool, with particular focus on its pattern system, preparing the ground for further work. Chapter 2 is dedicated to the tool itself. It contains a basic introduction to static analysis, comparison with other tools with similar purpose, description of the code representation used by DiffKemp internally, and information about the tool's architecture. The following chapter, number 3, focuses specifically on code change patterns, starting with an introduction to various kinds of refactorizations, followed by a description of DiffKemp's explicit built-in support to account for chosen types of semantics-preserving code changes that are not handled by the DiffKemp's main algorithm alone. The final part of the chapter concerns the existing system for custom user-defined patterns, including how these patterns are formally defined, represented, and algorithmically processed. The subsequent chapter — number 4 — proposes a way of encoding user-defined patterns in C. Then it provides a design for a special processing pass that can be used on C patterns compiled into LLVM IR, so the result is a valid LLVM IR DiffKemp pattern. The design of an extension to the tool to add an interface and support for these patterns is also described. The design chapter is followed by Chapter 5 with a similar structure that focuses on the implementation of the extension. It addresses concrete details of the implementation, which is part of DiffKemp's C++ library SimpLL and Python module `diffkemp`. The last chapter, 6, describes the process of testing and evaluating the C pattern encoding and the related DiffKemp extension. The testing is divided into two categories — synthetic tests testing the basic functionality of the C patterns and replication of existing non-C pattern tests with C patterns.

The result of this thesis is a fully designed and implemented DiffKemp extension, which allows users to define custom code-change patterns that define semantic equivalence for provided code segments, using the C language. The implementation is a contribution to an open-source project and is currently a pull request in the process of being reviewed in the official GitHub repository. The results of the testing show that the encoding used along with the implementation is usable in a wide variety of practical cases, comparable to the previously existing LLVM IR pattern encoding. The patterns in C are significantly easier to define, as C is a more user-friendly language compared to LLVM IR, which is mainly intended as an internal representation for machine analysis and transformations. This significantly simplifies the effort and expertise required from DiffKemp's users.

In the future, there are multiple ways to follow up on this thesis. DiffKemp currently includes an interactive result viewer. It may be possible to extend this viewer with the

ability to select specific differences found in the code to automatically create a C pattern for the selected code differences. Furthermore, the extensions to the DiffKemp's interface and workflow were done in a way that is general enough to make it possible to support new pattern encodings in the future. Finally, improving the pattern matching algorithm by incorporating another extension for DiffKemp, developed in parallel with this, which improves the matching algorithm for complicated arithmetic and logical instruction sequences into the pattern matching algorithm, can increase the expressiveness and usability of C-encoded patterns even more.

Bibliography

[1] Aho, A. V.; Lam, M. S.; Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. 369-370 p. ISBN 0321486811.

[2] Allen, F. E. Control flow analysis. In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: Association for Computing Machinery, 1970, p. 1–19. ISBN 9781450373869.

[3] Barnett, M.; Chang, B.-Y. E.; DeLine, R.; Jacobs, B. and Leino, K. R. M. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: Boer, F. S. de; Bonsangue, M. M.; Graf, S. and Roever, W.-P. de, ed. *Formal Methods for Components and Objects*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, p. 364–387. ISBN 978-3-540-36750-5.

[4] Ben Kiki, O.; Evans, C. and Ingerson, B. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008*, 2009, vol. 5, no. 11.

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

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

[7] Griffith, A. *GCC: The Complete Reference*. McGraw Hill LLC, 2002. 56–57 p. ISBN 9780072228168.

[8] International Organization for Standardization. *Programming languages — C*. ISO/IEC 9899:2011. December 2011. 133–136 p.

[9] International Organization for Standardization. *Programming languages — C*. ISO/IEC 9899:2011. December 2011. 81–82 p.

[10] International Organization for Standardization. *Programming languages — C*. ISO/IEC 9899:2023. April 2023. 128–130 p.

[11] Kirchner, K. and Rosenthaler, S. Bin2llvm: Analysis of Binary Programs Using LLVM Intermediate Representation. In: *Proceedings of the 12th International Conference on Availability, Reliability and Security*. New York, NY, USA: Association for Computing Machinery, 2017. ARES '17. ISBN 9781450352574.

[12] Křena, B. and Vojnar, T. Automated formal analysis and verification: an overview. *International Journal of General Systems*. Taylor & Francis, 2013, vol. 42, no. 4, p. 335–365.

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

[14] Lam, S. K.; Pitrou, A. and Seibert, S. Numba: a LLVM-based Python JIT compiler. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: Association for Computing Machinery, 2015. LLVM '15. ISBN 9781450340052.

[15] Lattner, C. and Adve, V. LLVM: a compilation framework for lifelong program analysis & transformation. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 2004, p. 75–86.

[16] Leung, H. and White, L. Insights into regression testing (software testing). In: *Proceedings. Conference on Software Maintenance - 1989*. 1989, p. 60–69.

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

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

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

[20] Nacke, K. *Learn LLVM 12: A beginner's guide to learning LLVM compiler tools and core libraries with C++*. Packt Publishing, 2021. 202–226 p. ISBN 9781839210037.

[21] Nacke, K. and Kwan, A. *Learn LLVM 17: A beginner's guide to learning LLVM compiler tools and core libraries with C++*. Packt Publishing, 2024. ISBN 9781837634675.

[22] Nacke, K. and Kwan, A. *Learn LLVM 17: A beginner's guide to learning LLVM compiler tools and core libraries with C++*. Packt Publishing, 2024. 40 p. ISBN 9781837634675.

[23] Red Hat. *What is Kernel Application Binary Interface (kABI)?* June 2023. Available at: https://access.redhat.com/solutions/444773. Retrieved on 22th January 2024.

[24] Reitz, K. and Schlusser, T. *The Hitchhiker's Guide to Python: Best Practices for Development*. O'Reilly Media, 2016. 222–223 p. ISBN 9781491933220.

[25] SARDA, S. and PANDEY, M. *LLVM Essentials*. Packt Publishing, 2015. 115-125 p. ISBN 9781783558629.

[26] SMYTH, N. *Red Hat Enterprise Linux 8 Essentials: Learn to Install, Administer and Deploy RHEL 8 Systems*. Payload Media, 2019. ISBN 9781951442040.

[27] THE APACHE SOFTWARE FOUNDATION. *Apache License, Version 2.0*. January 2004. Available at: https://raw.githubusercontent.com/llvm/llvm-project/main/llvm/LICENSE.TXT. Retrieved on 23th January 2024.

[28] ŠILLING, P. *Applying Code Change Patterns during Analysis of Program Equivalence*. Brno, CZ, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/24037/.

Appendix A

Contents of the included storage media

The included storage media has the following structure:

- `REAMDE.md` — basic information about the contents of the included storage media.
- `thesis.pdf` — PDF version of the thesis text.
- `thesis_print.pdf` — PDF version of the thesis text, variant for printing.
- `doc` — folder containing the source files necessary to generate the thesis text, specifically the `thesis.pdf` and `thesis_print.pdf` files.
- `diffkemp` — git repository of the DiffKemp project, set to the `cpatterns` branch, used to develop this thesis project. The work that was done on this thesis consists of the commits added to this branch compared to the `master` branch (try using `git diff master cpatterns` in the project folder to see the changes). Alternatively, the modifications can be seen at GitHub pull request for this extension[1] (note that the contents of the pull request are subject to change).

[1] https://github.com/diffkemp/diffkemp/pull/328

Appendix B

Header file for defining custom patterns

This chapter contains the entire contents of the implemented header file `diffkemp_patterns.h` providing macro-based interface for defining patterns in C (also available at project's GitHub repository[1]):

```
//===------ diffkemp_patterns.h - interface for defining C patterns -------===//
//
//       SimpLL - Program simplifier for analysis of semantic difference      //
//
// This file is published under Apache 2.0 license. See LICENSE for details.
// Author: Tomas Kucma, tomaskucma2@gmail.com
//===----------------------------------------------------------------------===//
///
/// \file
/// This file contains macros and functions used to define custom C patterns.
///
//===----------------------------------------------------------------------===//

/**
 * Usage:
 *
 * This header must be included and DIFFKEMP_CPATTERN macro must be defined when
 * defining patterns. Both of these are done automatically by the pattern
 * compiler.
 *
 * To define a standard instruction pattern, first define PATTERN_NAME macro to
 * the name of the pattern and PATTERN_ARGS to the list of arguments (without
 * brackets). Then use PATTERN_OLD and PATTERN_NEW macros to define the old and
 * new variants of the pattern. To define mapping between the old and the new
 * output variables, use MAPPING macro. Variables are mapped in the order they
 * are passed to the MAPPING macro.
 *
 * Example:
 *
 *  #define PATTERN_NAME sub
 *  #define PATTERN_ARGS int x, int y, int z
 *  PATTERN_OLD {
 *      int f = x - y;
```

[1] https://github.com/diffkemp/diffkemp/blob/c4f8fd468642f09e39a12eb2f693c8c2585b8bf4/diffkemp/simpll/diffkemp_patterns.h

```
 *      MAPPING(f);
 *  }
 *  PATTERN_NEW {
 *      int f = x - z;
 *      MAPPING(f);
 *  }
 *
 * For more examples, see the tests/regression/custom_patterns/c/ folder.
 *
 * Called function can be defined in a standard way. However, if the old and new
 * functions have identical names, but different signatures, use FUNCTION_OLD
 * and FUNCTION_NEW macros to declare and call them, to avoid name colisions.
 * First macro argument is the function name, rest of the macro arguments are
 * the function arguments.
 *
 * Example of a function declaration:
 *  void FUNCTION_OLD(sub, int x, int y, int z);
 *
 * If used for a definition of a function with void return type, it can be also
 * be used to define patterns, which is specifically useful if one wants to use
 * differently named arguments in each version of the pattern. However, it is
 * still necessary that the signatures match.
 *
 * To define a pattern that ends with a resolution of a condition, use
 * CONDITION_PATTERN_OLD and CONDITION_PATTERN_NEW macros. The pattern should
 * return a boolean value, used as the condition. It is not necessary to declare
 * output mapping for the condition variable. See `condition_only.c` pattern in
 * the aforementioned example folder.
 *
 * To define a value pattern, defining a semantic equivalence between two
 * values, use VALUE_PATTERN macro. First macro argument is the function
 * name, the second and the third are the old and the new value, respectively.
 * When using extern global variables, use pointer to the value instead.
 *
 * Examples of a value pattern:
 *  VALUE_PATTERN(value, 0b110UL << 8, 0b101UL << 7);
 *  VALUE_PATTERN(global_value, 30, &extern_var);
 *
 * Patterns defined in this way can then be used by the diffkemp tool using
 * standard -p flag, in the same way as the LLVM patterns. The compiled .ll
 * pattern file will be located in the same location as the .c pattern file from
 * which it was compiled. It is also possible to purely compile the .c pattern
 * file to .ll file without performing comparison, by using the compile-pattern
 * sub-command.
 *
 * When writting patterns for kernel, it is also necessary to provide following
 * definitions and includes at the very beginning of the file, before including
 * other kernel headers:
 *  #define __KERNEL__
 *  #define __BPF_TRACING__
 *  #define __HAVE_BUILTIN_BSWAP16__
 *  #define __HAVE_BUILTIN_BSWAP32__
 *  #define __HAVE_BUILTIN_BSWAP64__
 *  #include <linux/kconfig.h>
 * Then, following include paths in the following order must be provided to the
 * compiler:
 *  -I{linux}/arch/x86/include/
 *  -I{linux}/arch/x86/include/generated/
 *  -I{linux}/include/
```

```c
 *  -I{linux}/arch/x86/include/uapi
 *  -I{linux}/arch/x86/include/generated/uapi
 *  -I{linux}/include/uapi
 *  -I{linux}/include/generated/uapi
 * This can be done automatically by the pattern compiler by providing the path
 * to the kernel source files using --c-pattern-kernel-path option.
 *
 * Patterns written in C can also be loaded from YAML file, in the same way as
 * the LLVM patterns. The YAML file must contain field `patterns` with the list
 * of pattern files. Additionally, it is possible to provide extra clang options
 * for each individual pattern, using field `clang_append`, by providing a map
 * of pattern names to lists of clang options to append to them. For examples,
 * see the tests/regression/custom_patterns/c/
 */

#ifndef DIFFKEMP_SIMPLL_DIFFKEMP_PATTERNS_H
#define DIFFKEMP_SIMPLL_DIFFKEMP_PATTERNS_H

// Internal definitions

#define __DIFFKEMP_STRINGIFY_IMPL(macro) #macro
#define __DIFFKEMP_STRINGIFY(macro) __DIFFKEMP_STRINGIFY_IMPL(macro)
#define __DIFFKEMP_CONCAT_IMPL(arg1, arg2) arg1##arg2
#define __DIFFKEMP_CONCAT(arg1, arg2) __DIFFKEMP_CONCAT_IMPL(arg1, arg2)

#define __DIFFKEMP_PREFIX_OLD __diffkemp_old_
#define __DIFFKEMP_PREFIX_NEW __diffkemp_new_
#define __DIFFKEMP_MAPPING_NAME __diffkemp_output_mapping
#define __DIFFKEMP_CPATTERN_INDICATOR_NAME __diffkemp_is_cpattern

#define __DIFFKEMP_FUNCTION(version, name, ...)                               \
    __DIFFKEMP_CONCAT(__DIFFKEMP_PREFIX_##version, name)(__VA_ARGS__)

#ifdef DIFFKEMP_CPATTERN
int __DIFFKEMP_CPATTERN_INDICATOR_NAME = 1;
#if __STDC_VERSION__ >= 202000L
// In the C2x standard, functions declared with empty argument list no longer
// take any arguments, however, it is possible to use variadic arguments without
// type instead.
void __DIFFKEMP_MAPPING_NAME(...);
#else
void __DIFFKEMP_MAPPING_NAME();
#endif // __STDC_VERSION >= 202000L
#endif // DIFFKEMP_CPATTERN

// Public interface for handling patterns

/// Stringified name of a global variable, the presence of which is used to
/// detect whether a given .ll module is unpreprocessed custom C pattern.
#define CPATTERN_INDICATOR                                                    \
    __DIFFKEMP_STRINGIFY(__DIFFKEMP_CPATTERN_INDICATOR_NAME)

/// String versions of naming schemes.
#define CPATTERN_OLD_PREFIX __DIFFKEMP_STRINGIFY(__DIFFKEMP_PREFIX_OLD)
#define CPATTERN_NEW_PREFIX __DIFFKEMP_STRINGIFY(__DIFFKEMP_PREFIX_NEW)
#define CPATTERN_OUTPUT_MAPPING_NAME                                          \
    __DIFFKEMP_STRINGIFY(__DIFFKEMP_MAPPING_NAME)

// Public interface for defining patterns
```

```
#ifdef DIFFKEMP_CPATTERN

/// Used to define old/new variants of a function with identical name, but
/// different signature, to avoid conflicting definitions. Can be also used to
/// define patterns, if provided definition, for example to use different names
/// for arguments.
#define FUNCTION_OLD(name, ...) __DIFFKEMP_FUNCTION(OLD, name, __VA_ARGS__)
#define FUNCTION_NEW(name, ...) __DIFFKEMP_FUNCTION(NEW, name, __VA_ARGS__)

/// Used to define standard instruction patterns. To use, first define
/// PATTERN_NAME macro to the name of the pattern and PATTERN_ARGS to the list
/// of arguments (without parentheses).
#define PATTERN_OLD void FUNCTION_OLD(PATTERN_NAME, PATTERN_ARGS)
#define PATTERN_NEW void FUNCTION_NEW(PATTERN_NAME, PATTERN_ARGS)

/// Used to define condition pattern. Use identically as standard pattern, only
/// the pattern should return a boolean value, used as the condition.
#define CONDITION_PATTERN_OLD _Bool FUNCTION_OLD(PATTERN_NAME, PATTERN_ARGS)
#define CONDITION_PATTERN_NEW _Bool FUNCTION_NEW(PATTERN_NAME, PATTERN_ARGS)

/// Used to define value patterns. To use, simply provide the old and the new
/// value.
#define VALUE_PATTERN(name, old_value, new_value)                              \
    __typeof__(old_value) FUNCTION_OLD(name, ) { return old_value; }           \
    __typeof__(new_value) FUNCTION_NEW(name, ) { return new_value; }

#define MAPPING(...) __DIFFKEMP_MAPPING_NAME(__VA_ARGS__)

#endif // DIFFKEMP_CPATTERN

#endif // DIFFKEMP_SIMPLL_DIFFKEMP_PATTERNS_H
```