BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Applying Code Change Patterns during Analysis of Program Equivalence

Petr Šilling*

Abstract
For some software projects, it might be crucial to ensure semantic stability of their core components between multiple release versions. Nowadays, semantic stability may be checked automatically even on large real-world projects using scalable tools, e.g., DIFFKEMP, which focuses on checking semantic equivalence of different versions of the Linux kernel. However, while these tools are highly scalable, they do tend to report some false-positives. Therefore, in this paper, we propose a static analysis method for matching patterns of recurrent changes between different versions of code. The proposed solution introduces a novel pattern matching algorithm based on gradual comparison of instructions according to their control flow and a method for encoding code change patterns into the LLVM intermediate representation. The proposed analysis has been implemented as an extension of DIFFKEMP, where we demonstrate how it may eliminate a substantial amount of non-equivalence results, which would generally require manual inspection.

Keywords: DIFFKEMP — LLVM — GNU/Linux kernel — Pattern Matching — Semantic Equivalence

Supplementary Material: DIFFKEMP GitHub Repository

*xsilli01@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

When modifying code that should ideally remain stable and consistent for extended periods of time (e.g., for the lifetime of a major software release version), it might be vital to know which parts of the program will get impacted, or, perhaps even more importantly, how will the changes affect semantics. To ease the process of finding unintentional side-effects, developers may want to utilize automated static analyzers of semantic equivalence to compare different program versions. Unfortunately, this is typically not possible since current techniques for sound checking of semantic equivalence generally depend on computationally intensive formal methods. Consequently, the applicability of such analyzers to large-scale projects is fairly limited, forcing developers to rely almost entirely on an especially time-consuming and error-prone manual analysis instead.

Nevertheless, analyzers that concentrate on scalability and usability on large projects are slowly emerging as well. One such analyzer is DIFFKEMP [1], a tool for checking semantic equivalence of different versions of large C projects, which—due to a particular interest of Red Hat—focuses primarily on the Linux kernel. DIFFKEMP tries to find the middle ground between formal methods, which are sound but heavy-weight, and simplified light-weight methods (often based on plain text similarity)—it introduces a highly scalable technique that usually produces only a small number of false non-equivalence results.

To further reduce the number of false-positives reported by DIFFKEMP, and to introduce the option to selectively hide already reviewed semantic changes (e.g., security fixes), we propose to extend DIFFKEMP with support for custom patterns of code modifications. These patterns enable users to specify which kinds of changes should get ignored during the semantic comparison.

In particular, we summarize the contributions of this paper as follows:

- First, we introduce the idea of so-called *code change patterns (CCPs)* and prepare a repre-

sentation of CCPs that can be easily parsed by DIFFKEMP.

- Second, we propose a method for detecting CCPs in compared programs. The proposed *matching* method is based on the *subgraph isomorphism problem* and leverages the LLVM infrastructure—in particular, the fact that LLVM functions are represented as *control-flow graphs (CFGs)*.
- Finally, we evaluate the extension on multiple past versions of the Linux kernel, demonstrating how it can help eliminate false-positive or potentially undesirable non-equivalence reports.

Before describing the contributions in detail, we first present the basic concepts behind DIFFKEMP.

2. Comparing Programs with DIFFKEMP

In recent years, several projects on static analysis of semantic equivalence have emerged, creating a widely studied field of program analysis. The tools implemented based on these projects, such as LLREVE [2], typically rely on costly formal methods, which—while completely eliminating false-positives common for more relaxed techniques—suffer greatly from scalability issues. Application on large enterprise projects, such as the Linux kernel, is therefore not feasible. More scalable alternatives (e.g., based on abstract syntax tree comparison [3]) exist as well. However, such tools can generally process only the most simple semantics-preserving changes, which might also be insufficient for larger projects. A more complete overview of analyzers of semantic differences can be seen in [1].

DIFFKEMP can be categorized as a sophisticated light-weight tool for checking semantic equality between two C programs. It aims to provide an accurate overview of semantic differences between the compared programs while retaining the high scalability typical for light-weight analyzers. To give a concrete example, DIFFKEMP is able to automatically compare two versions of the Linux kernel in the order of minutes. And while [1] does show that the results may contain a small number of false-positives, their vast majority tends to be correct. Using DIFFKEMP drastically lowers the amount of manual work required when reviewing changes, which is something that would not be achievable with formal approaches, at least not on projects with the size of the Linux kernel. DIFFKEMP achieves this using three major concepts: (1) the translation of compared programs into the LLVM intermediate representation [4], (2) a specialized definition of function equality based on so-called *synchronisation points*, and (3) an analysis that tries to compare programs *instruction-to-instruction*.

2.1 Representation of Compared Programs

During its analysis, DIFFKEMP utilizes control-flow graphs (CFGs). In particular, DIFFKEMP translates the compared C programs into a low-level code representation called LLVM IR—the intermediate representation of Clang/LLVM. In LLVM IR, each function corresponds to a single CFG and may be perceived as one. The following definition of CFGs is inspired by the notion of CFGs presented in LLVM IR and by [1].

A CFG is a directed graph in which nodes are basic blocks and edges represent program branches. Each *basic block* is composed of a sequence of instructions. An *instruction* performs an *operation* over a (possibly empty) list of operands and may store its result into a local variable. Each *operand* is either a variable (global or local), a constant, or a function. All variables and constants are typed. The type system and the operations associated with different kinds of LLVM instructions are defined by [5]. Additionally, each CFG must satisfy the *static single assignment (SSA)* property, meaning that each variable must be assigned to at most once.

Each internal instruction of a basic block has exactly one successor (the instruction immediately following it). A basic block ends by a branch instruction or by a terminator instruction. Each *branch* instruction has either one or two successors, all of which must be initial instructions of basic blocks. Branch instructions with one and two successors are called *unconditional* and *conditional* branches, respectively. A *terminator* instruction has no direct successors as it terminates the function, possibly returning a result.

2.2 Definition of Function Equality

DIFFKEMP analyses programs that are translated into LLVM IR by their CFGs (i.e., by LLVM functions). Since checking semantic equality of entire functions at once would be complicated, DIFFKEMP achieves it by splitting each function into the same number of blocks that can be compared separately. These blocks are delimited by so-called *synchronisation points*. For each block of code located between a pair of succeeding synchronisation points in the first compared function, the block of code between the pair of corresponding synchronisation points in the second compared function must be semantically equal.

Intuitively, two blocks of code may be considered semantically equal if they both terminate and produce the same output for the same input, or if they both do

Function f_1

```
1  define i32 @f1(i32 %x) {
2      %1 = add i32 %x, 42
3      %2 = add i32 %1, %1
4      %3 = add i32 %2, %1
5      ret i32 %3
6  }
```

Function f_2

```
1  define i32 @f2(i32 %x) {
2      %1 = add i32 %x, 42
3      %2 = mul i32 %1, 3
4      ret i32 %2
5  }
```

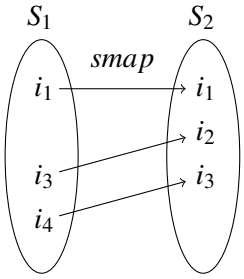

Figure 1. Two compared functions f_1 and f_2 with the associated *smap* and *varmap* mapping functions. For $n \in \{1,2,3,4\}$, instructions from both functions are represented in order by i_n. For a parameter x, f_1 and f_2 are semantically equal, as they both calculate the result of $3 \times (x+42)$. Instructions that should be synchronized are connected by arrows, and variables that should be mapped share the same colour in both functions.

not terminate. Input is defined by the values of the input variables and the initial state of memory (i.e., both the stack and the heap), and output by the output variables and the final state of the memory. A more formal definition of semantic equality is beyond the scope of this paper. However, the general idea of the definition presented in [1] is based around two mapping functions *smap* and *varmap*, where *smap* represents a bijective mapping between sets of synchronisation points from both compared functions, and *varmap* represents a similar mapping between variables. A graphical example of both mapping functions can be seen in Figure 1.

2.3 Algorithm for Checking Function Equality

The top-level comparison algorithm builds on the idea introduced above. For two compared functions f_1 and f_2, it tries to find appropriate sets S_1 and S_2 of synchronisation points and the mapping functions *smap* and *varmap* such that all blocks of code between corresponding pairs of synchronisation points are semantically equal. A simplified version of the algorithm is presented in Algorithm 1. The full version of the algorithm can be found in [1].

Algorithm 1 starts by taking two functions f_1 and f_2 and running traditional semantics-preserving code transformations on them (e.g., dead code elimination). This is important because individual instructions are generally quite simple to compare (they should perform the same operations on the same operands or on operands that can be mapped to each other using *varmap*). Applying these transformations makes it much more likely for synchronisation points to get placed per instruction, which greatly improves the performance of the algorithm on functions that are not syntactically the same.

Then, the initialisation of synchronisation sets and maps takes place. At first, only the first instructions in the entry basic blocks of functions f_1 and f_2 are considered synchronized (Lines 2–3). These are denoted

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

1: perform *code transformations* of f_1 and f_2
   // Initialise synchronisation sets and maps
2: $S_1 := \{i^1_{in}\}$, $S_2 := \{i^2_{in}\}$
3: $smap(i^1_{in}) := i^2_{in}$ *// Map the entry instruction pair*
4: add parameter and global variable mappings in *varmap*
   // Primary comparison loop
5: $Q := \{(i^1_{in}, i^2_{in})\}$ *// Initialise the main queue*
6: **while** $Q \neq \emptyset$ **do**
7: take any pair (s_1, s_2) from Q
8: $p := detectPattern(s_1, s_2)$
9: **foreach** $(s'_1, s'_2) \in succPair_p(s_1, s_2)$ **do**
10: **if** (s_1, s'_1) *is semantically different to* (s_2, s'_2) **then**
11: **return** *false*
   // Update synchronisation sets and maps
12: $S_1 := S_1 \cup \{s'_1\}$, $S_2 := S_2 \cup \{s'_2\}$
13: $smap(s'_1) := s'_2$
14: update *varmap* according to p
15: insert (s'_1, s'_2) into Q
16: **return** *true*

Algorithm 1: Semantic comparison of functions

by i^1_{in} and i^2_{in} for f_1 and f_2, respectively. Furthermore, a mapping between pairs of parameters and pairs of global variables is created as well (Line 4). Parameters of f_1 and f_2 are mapped based on their order, while global variables that get used in f_1 and f_2 are mapped according to their names.

Afterwards, the main comparison loop begins, operating until the queue Q of pairs of synchronisation points is empty. Initially, only the first synchronized pair of instructions (i^1_{in}, i^2_{in}) is queued up for analysis. At the start of each iteration, a single pair (s_1, s_2) is taken from the queue. Then, each pair (s_1, s_2) gets analysed by the function $detectPattern$, which checks whether some supported *semantics-preserving change pattern (SPCP)* could be applied to it. SPCPs are statically defined patterns of recurrent code modifications, which—if detected in a block of code—are known to preserve program semantics (e.g., shifts in enumeration values).

Subsequently, the function $succPair_p$ retrieves all possible successor pairs following (s_1, s_2). These are typically placed at the instructions immediately following s_1 and s_2. However, if a SPCP gets detected, pattern-specific successors might get selected instead. For all possible successor pairs (s'_1, s'_2), the algorithm utilizes instruction comparison and synchronisation maps to check whether the blocks of code between the corresponding pairs of synchronisation points are semantically equal (Line 10). If so, the algorithm continues, updating the synchronisation sets and maps, and queueing up (s'_1, s'_2) for analysis (Lines 12–15).

Finally, if the queue Q is successfully emptied, the functions f_1 and f_2 are proclaimed semantically equal (Line 16), and the comparison of the next pair of functions may begin.

3. Code Change Patterns

Code change patterns (CCPs) are descriptions of *recurrent* software modifications [6]. They are especially important for DIFFKEMP since not all functions can be compared instruction-to-instruction, which is why DIFFKEMP already supports many CCPs in the form of built-in SPCPs. However, since the total number of existing CCPs is theoretically unbounded, using statically predefined SPCPs for all of them is not feasible.

Additionally, a lot of commonly repeating patterns of changes alter the semantics but are known to be safe (e.g., security fixes and safety assertions). Therefore, it might be desirable to filter out such changes from comparison results. With respect to that and to our own experiences with the Linux kernel, we analyse two kinds of CCPs:

a) *Semantics-preserving patterns*, also known as *refactoring patterns*, i.e., patterns that modify code in a way that preserves its observable behaviour [7]. Some of these, e.g., the addition of a new value into an enumeration type, are already handled by built-in SPCPs already present in DIFFKEMP [1].
b) *Semantics-altering patterns*, corresponding to changes that cause semantic differences. For example, they might add, remove, or rewrite a conditional expression in an attempt to fix programming mistakes [8].

By carefully examining the differences reported by DIFFKEMP when comparing multiple versions of the Red Hat Enterprise Linux (RHEL), we were able to confirm the occurrence of both kinds of CCPs within the kernel of RHEL. Since built-in SPCPs might be impractical in this scenario (especially when semantics-altering patterns are concerned), we propose to reduce the number of pattern-related differences shown in the results using CCPs encoded into LLVM IR. Encoded CCPs may then get dynamically loaded into DIFFKEMP.

3.1 Code Change Pattern Definition

Before presenting the pattern encoding method, we formally define CCPs. The definition, as well as the rest of this paper, assumes that DIFFKEMP is used to compare two versions of the same program. Then, a code change pattern p can be understood as a tuple

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

where

- c_o and c_n are the *code fragments* associated with the older and newer versions of the compared programs, respectively, and
- *imap* and *omap* are mapping functions that map the input of c_o to the input of c_n, and the output of c_o to the output of c_n, respectively.

Each code fragment c is composed of input, output, and its main body, which describes how to transform input into output. To simplify the presentation, we introduce functions $in(c)$ and $out(c)$ to obtain the input and the output of c, respectively.

3.2 Code Change Pattern Representation

Since DIFFKEMP utilizes the LLVM infrastructure, we propose to encode CCPs using LLVM IR, as doing so does not require any new libraries or parsing tools. On the other hand, larger LLVM IR patterns may be rather hard to create manually (LLVM IR is a very low-level language). However, that is not our primary concern since, in the future, LLVM IR patterns might get produced automatically.

Because LLVM IR directly describes blocks of code, CCPs can get encoded in a very straightforward way. In particular, we represent c_o and c_n of each code change pattern p by two LLVM functions, prefixed by `diffkemp.old` and `diffkemp.new`, respectively. The functions (i.e., code fragments) have the following properties:

- Their inputs are encoded by function parameters. The corresponding *imap* mapping is generated based on the order of the parameters.
- The outputs get specified by calls to the function `@diffkemp.mapping`. All arguments of such calls get mapped in order, creating *omap*.

For RHEL 8.1

```
1 define void @diffkemp.old.fragment(i32) {
2   %2 = icmp sle i32 %0, 30
3   call void @diffkemp.mapping(i1 %2)
4   ret void
5 }
```

For RHEL 8.2

```
1 define void @diffkemp.new.fragment(i32) {
2   %2 = load i32, i32* @node
3   %3 = icmp sle i32 %0, %2
4   call void @diffkemp.mapping(i1 %3)
5   ret void
6 }
```

Figure 2. Representation of an LLVM IR pattern extracted from differences reported by DIFFKEMP during the comparison of two versions of the RHEL kernel. The pattern describes a substitution of an integer macro (older version) for a global constant `@node` (newer version).

- The instructions inside each function represent the corresponding code fragment body and need to get matched in the compared programs in order to detect the pattern.

An example of a CCP that has been encoded into LLVM IR can be seen in Figure 2.

4. Pattern Matching Extension

The extension proposed in this work builds on the notions introduced in Algorithm 1. In particular, it also uses the LLVM infrastructure to analyse programs by their CFGs, and it builds its own mapping of variables. However, compared to the top-level algorithm, the extension does not aim at finding instructions that are semantically different. Instead, it searches for instructions that can be *matched* to those present in code fragments of the selected patterns.

Since the instruction matching is performed on CFGs, the process can be reduced to the subgraph isomorphism problem. We note that there are many well-known algorithms focusing on this problem, such as Ullmann's algorithm [9]. However, these approaches would be rather hard to integrate into the robust LLVM architecture since they typically aim at general graphs and not specifically at CFGs. Therefore, we have decided to implement our own matching algorithm. However, we do use certain heuristics that are mentioned in previous works (e.g., the pruning of graph nodes based on the number of neighbours), although these have been omitted for brevity.

The pattern matching process is shown in Algorithm 2. Since pattern matching should serve as the final validation step before declaring two functions as semantically different, the algorithm should be run after the blocks of code (s_1, s_1') and (s_2, s_2'), compared on Line 10 of Algorithm 1, get determined as not equal. In particular, the algorithm needs to be evaluated *twice* for each loaded pattern until a match is found or all patterns are exhausted. In the first evaluation, the algorithm expects to receive the first instruction i_p^d from

Input: Instruction i_p^d from one of the compared programs p
Code fragment c from one of the loaded patterns
Result: r, which is *true* if matched, *false* otherwise
I, the set of input match pairs
O, the set of output match pairs

1: $I := \{\}$, $O := \{\}$
2: initialize $varmap_c$ with shared global variables
3: $Q := \{(i_c^b, i_p^d)\}$
4: **while** $Q \neq \emptyset$ **do**
5: take any pair (i_c, i_p) from Q
6: **if** i_c *can be matched to* i_p **then**
// Process input and output
7: **foreach** $o_c \in ops(i_c)$ **do**
8: **if** $o_c \in in(c)$ **then**
9: $o_p := o \in ops(i_p)$ s.t. o_c matches o
10: $I := I \cup \{(o_c, o_p)\}$
11: **if** $i_c \in out(c)$ **then**
12: $O := O \cup \{(i_c, i_p)\}$
13: $varmap_c(i_c) := i_p$
// Queue up the following instruction pair
14: **foreach** $(i_c', i_p') \in succInstPair(i_c, i_p)$ **do**
15: insert (i_c', i_p') into Q
16: **if** *all instructions in c have been matched* **then**
17: **return** $(true, I, O)$
18: **else**
19: **return** $(false, \{\}, \{\})$

Algorithm 2: CFG-based pattern matching

the older program version that has been compared as semantically different and the pattern code fragment c corresponding to the program version p that contains i_p^d and (s_1, s_1')—which, for the first evaluation, would be the older version. For (s_2, s_2') and its program version, the second evaluation is analogical.

Each evaluation starts by creating the set I, which provides information about how to map the input of c to the operands of instructions from p, and the set O, which analogically describes how to map the output of c to the instructions from p. Both I and O are initially empty. Additionally, a mapping between global variables that are used within both c and p and that share the same name is established (Line 2).

Then, the primary pattern matching loop begins. The loop works similarly to Algorithm 1—it relies on

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

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

Algorithm 3: Calculating successor instructions

the queue Q, operating until Q is emptied. However, since pattern matching is always done instruction-to-instruction, instructions are always queued up in the same sequence as they are present in the underlying LLVM IR code. Therefore, synchronisation points and the corresponding mapping function are not necessary. Initially, only the instruction pair (i_c^b, i_p^d) is queued up for matching, where i_c^b denotes the first instruction in the main body of c.

At the beginning of each iteration, a single pair of instructions (i_c, i_p) gets taken from Q, and the algorithm checks whether i_c can be matched to i_p. Similarly to the top-level algorithm, the matching of individual instructions is based on simple instruction comparison.

If the matching succeeds, the algorithm iterates over all operands o_c of i_c (instruction operands are retrieved using the function *ops*). For each such operand that is also the input of c, a pair of operands (o_c, o_p) is placed into I (Line 10), where o_p denotes the operand of i_p that has been matched to o_c. In other words, a pair where the first element is an input of c used as an operand of i_c, and the second element is the matching operand of i_p is inserted into I.

Additionally, Lines 11–12 perform a similar analysis concerning the output of c. However, it is the instructions themselves (or, more specifically, the variables created by them) that may get used as part of the output of c (i.e., not their operands). Therefore, if c specifies i_c as its output, the analysed instruction pair (i_c, i_p) itself gets placed into O. After the input and the output are processed, a variable mapping between i_c and i_p is created (Line 13). This mapping is used to match instructions of c and p within Algorithm 2.

Afterwards, the function *succInstPair*—a specialized variant of *succPair*—retrieves all instruction pairs (i_c', i_p') that should get queued up after (i_c, i_p). The implementation of *succInstPair* is displayed in Algorithm 3, which operates on the currently analysed instructions i_c and i_p. If a match between i_c and i_p has been established, *succInstPair* behaves analogically to *succPair*. Otherwise, if i_p has a single successor, *succInstPair* indicates that the matching algorithm should try to match i_c to the instruction that immediately follows i_p. In other words, the matching algorithm allows to skip instructions of p when searching for suitable instructions matching the pattern code fragment. The function *succ* retrieves the single successor of the given instruction. If the number of immediate successors of i_p is different, *succInstPair* yields an error, failing the pattern matching process, as it either cannot continue (i_p has no successors) or would branch out (i_p has two successors).

Finally, if the queue Q is emptied and all of the instructions in the main body of c have been matched, Algorithm 2 returns *true* and the resulting sets I and O (Line 17). Otherwise, the algorithm returns *false* and two empty sets since the matching failed.

Additionally, if both evaluations of Algorithm 2 are successful for a given pattern, the resulting sets of input match pairs and output match pairs should be validated with respect to the input mapping function *imap* and the output mapping function *omap* of the pattern. However, this validation goes beyond the scope of this work.

5. Evaluation on the Linux Kernel

In order to verify that our extension is able to eliminate reported differences associated with CCPs, we performed a series of experiments. As the target of our experiments, we chose Red Hat Enterprise Linux (RHEL) due to its popularity and emphasis on stability—its kernel contains a list of functions, a so-called *Kernel Application Binary Interface* (KABI), that should ideally remain semantically stable for the lifetime of each major release of RHEL. During the experiment, we selected three pairs of the most recently released versions of RHEL. For each pair of versions we performed the following sequence of actions:

1) The KABIs of both versions were compared by DIFFKEMP without using patterns.
2) All reported differences were manually examined for the existence of CCPs. In particular, in each pair of versions, we—to the best of our abilities—searched for the five most repetitive CCPs.
3) All of the identified CCPs were encoded into LLVM IR.
4) The selected RHEL versions were compared again, this time with all of the created LLVM IR patterns being loaded into DIFFKEMP.

Table 1. A comparison of pairs of RHEL versions with and without custom LLVM IR patterns

RHEL versions	KABI functions	Not equal results		Runtime (mm:ss)	
		without patterns	with patterns	without patterns	with patterns
8.0/8.1	471	85	75	03:40	03:36
8.1/8.2	521	154	139	03:57	03:55
8.2/8.3	628	177	168	05:38	05:28

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

The results of our experiments can be seen in Table 1. Each experiment was repeated five times, and the runtimes were calculated as averages of the time spent on comparing KABI functions compiled to LLVM IR on a 6 core, 2.80 GHz Intel Core i5 Coffee Lake machine with 16 GB of RAM. The compilation time is not included.

For all pairs of the compared versions of RHEL, the results show that by applying patterns, the total number of KABI functions evaluated as not equal can be lowered, although the amount of eliminated non-equal results varies considerably. However, these fluctuations are caused by the differences in the repetitiveness of the CCPs found in each pair of versions and not by any potential issues with the extension.

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

Based on the findings presented above, we were able to confirm that our extension can help improve the results reported by DIFFKEMP (at least for the evaluated versions of RHEL) since many false-positives can be linked to a particular CCP. Moreover, the results indicate that, generally, the usage of patterns might have a slightly positive impact on runtime performance. Since the proposed pattern representation is generic, these findings also suggest that the extension should be broadly applicable to C projects other than the kernel of RHEL.

6. Conclusion

In this paper, we have proposed a pattern matching extension for DIFFKEMP, an analyzer of semantic differences between C programs. With the extension, we have introduced an encoding of patterns of recurrent software modifications based on LLVM IR, and a pattern matching algorithm that utilizes the LLVM infrastructure in a way similar to DIFFKEMP. Last, we have evaluated the proposed extension on multiple versions of the RHEL kernel, showing that with a proper set of patterns, it can eliminate specific non-equivalence results reported by DIFFKEMP, possibly further limiting the amount of work required to check semantic stability between different versions of the same program.

Future work could improve on these ideas, for example, by (a) lowering the complexity of the matching algorithm, e.g., by adding more heuristics, or (b) supporting more user-friendly (ideally fully automated) ways of generating LLVM IR patterns. Since our experiments do not show any issues regarding time complexity, it might be most optimal if future enhancements focus on, e.g., generating patterns directly from the C source of the compared programs.

Acknowledgements

I would like to thank my supervisor Viktor Malík for his help with understanding DIFFKEMP.

References

[1] Viktor Malík and Tomáš Vojnar. Automatically checking semantic equivalence between versions of large-scale C projects. *Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 329–339, 2021.

[2] Moritz Kiefer, Vladimir Klebanov, and Mattias Ulbrich. Relational program reasoning using compiler ir. *Journal of Automated Reasoning*, 60(3):337–363, September 2017.

[3] Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of*

the 2005 International Workshop on Mining Software Repositories, volume 30 of *MSR '05*, pages 1–5, New York, NY, USA, 5 2005. Association for Computing Machinery.

[4] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–86, Palo Alto, CA, USA, 3 2004. IEEE Computer Society.

[5] LLVM Project. Llvm language reference manual. [online], January 2021. Revised 15. 1. 2021 [cit. 2021-03-24]. Available at: https://releases.llvm.org/11.0.1/docs/LangRef.html.

[6] Matias Martinez, Laurence Duchien, and Martin Monperrus. Automatically extracting instances of code change patterns with ast analysis. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 388–391, Washington, DC, USA, 9 2013. IEEE Computer Society.

[7] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA, 2 edition, November 2018.

[8] Kai Pan, Sunghun Kim, and E. James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.

[9] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.