Automatically Checking Semantic Equivalence between Versions of Large-Scale C Projects

Viktor Malík
Red Hat and Brno University of Technology,
Faculty of Information Technology

Tomáš Vojnar
Brno University of Technology,
Faculty of Information Technology

***Abstract*—Motivated by a need of some software projects to ensure semantic stability of some of their core parts, the paper proposes a highly-scalable approach for automatically checking semantic equivalence of different versions of large C projects, with a particular focus on the Linux kernel. The proposed method uses a novel combination of pattern matching with light-weight static analysis and control-flow transformations. Although the method cannot prove equivalence on heavily refactored code, it can compare thousands of functions in minutes while producing a low number of false non-equality verdicts as our experiments show. We implemented our approach in a tool called DIFFKEMP and we show that DIFFKEMP, unlike other existing tools, gives practically useful results even on projects of the size of the Linux kernel.**

I. INTRODUCTION

The problem of automatically checking semantic equivalence of programs is nowadays a widely studied field of program analysis. Unfortunately, despite a lot of progress, existing approaches for sound equivalence checking often rely on heavy-weight formal methods and consequently have problems with scalability. This limits their usage in the industry despite the fact that there exist a lot of their potential applications.

One of such applications is for systems that require some sort of stability and backwards compatibility. These can be, e.g., various standard libraries or kernels of operating systems. One of the best known commercial operating systems is Red Hat Enterprise Linux (RHEL), whose kernel includes a list of functions, a so-called *Kernel Application Binary Interface* (KABI), whose semantics should be stable across the lifetime of a single major release (unless changes are dictated, e.g., by security issues). Ensuring this stability at presence of constant refactoring changes is rather difficult, and any (even partial) automation of the backwards compatibility checking has huge potential to save a lot of human effort and costs.

In this paper, we propose a novel automated method for verifying whether two versions of a program that should have the same semantics—as one of them is expected to be a refactoring of the other—do indeed have the same semantics (and hence that no error has been done during the refactoring). We aim at applicability to large-scale industrial code. Applicability of our approach to the Linux kernel is particularly important to us, which is motivated by a concrete interest of Red Hat. Our approach is, however, more general and applicable on other C projects too, which we demonstrate by experiments with one of the existing implementations of the C standard library.

To achieve the needed scalability to hundreds of thousands of lines of code, which—to the best of our knowledge—is beyond capabilities of current tools and approaches, we propose to compare the different versions primarily *per-instruction* on the level of their LLVM intermediate code representation. Of course, checking semantic equivalence on the level of single instructions would normally lead to many false non-equivalence results. In order to resolve this problem while retaining scalability, we *pre-process* the code to be compared using various static analyses and code transformations to bring the code to a form that can be compared instruction-by-instruction as often as possible.

Beyond checking per-instruction equivalence, we allow program versions under comparison to differ in ways described by so-called *semantics-preserving change patterns* (SPCPs). Our method is generic in the set of SPCPs to be applied provided they are described in a particular way that we propose. We call such SPCPs as *effective*.

We show that a number of refactoring patterns known in the literature—in particular, we consider those presented in [9]—can be handled by our approach even without using SPCPs, simply due to the use of the considered code transformations. Moreover, we show that many further patterns from [9] can be handled using the proposed notion of effective SPCPs. This applies notably to those patterns that appeared in multiple past versions of the Linux kernel as our extensive experimental study in this area shows. Through this study, we also identify several further change patterns not covered by [9] that commonly appear and can be handled via effective SPCPs too. Finally, to further broaden the scope of our approach, we extend it by a way to handle *code relocations* that goes beyond the notion of effective SPCPs.

We implemented our approach in a tool called DIFFKEMP. We have applied DIFFKEMP on the Linux kernel—both on the RHEL version and the upstream version—as well as the *musl* C standard library (https://musl.libc.org). Our experiments confirm that DIFFKEMP scales far better than other approaches for checking semantic equality and that it can indeed safely verify quite many refactorings and be quite useful in practice.

Overall, we summarize our contributions as follows: (1) We propose a light-weight approach for checking semantic equivalence of program versions obtained by refactoring that is—to the best of our knowledge—much more scalable than other existing approaches for checking semantic equivalence. (2) We have implemented the proposed method in a new open-source tool DIFFKEMP that is capable of checking preservation of semantics of refactored code compiled into the LLVM inter-

mediate representation. (3) We demonstrate the capabilities of the approach on several practical applications on large-scale real-life projects including the Linux kernel (which has the size in millions of LOC) and the *musl* C standard library.

Related Work: There is a number of existing works on static analysis of semantic equivalence—for an overview, see, e.g., [17]. Some of the approaches were implemented in tools applied to real-life code, such as RVT [10], SYMDIFF [16], [14], DISE [21], [2], LLREVE [15], or UC-KLEE [25]. Many of these tools use a similar general approach—namely, equivalence of functions under comparison is encoded using formulae and/or special program constructions, and a suitable decision procedure or program verifier is used to prove equality. In particular, SYMDIFF uses Z3 [6], RVT uses CBMC [5], UC-KLEE uses KLEE [3]. LLREVE represents function equality by a set of Horn clauses and uses a Horn solver, such as Z3, to solve them. DISE is slightly different in that it employs KLEE to generate function summaries and then compares the summaries. These approaches build on heavy-weight formal methods that, despite a lot of advances, do still not scale enough to allow equivalence checking on large code. This applies even to one of the latest works in the area [4], based on semantics-driven alignment of program traces, which scales on benchmarks from vectorizing compilers up to tens of lines.

On the other hand, there exist light-weight and extremly fast tools based on text similarity (such as the Unix `diff` tool) or on simple abstract-syntax-tree matching [19] that are able to compare huge code bases in the order of seconds. These are, typically, able to handle only the simplest semantics-preserving changes (such as, e.g., variable renaming).

Compared with the mentioned works, our approach lies in between the two areas. While our method is not as fast as the simple approaches, it can show equality of much more complicated refactorings in the order of minutes for a similarly large code. Also, since we build on light-weight back-end approaches, our method scales far better than those using formal methods, at the possible expense of not being able to show equality of some heavily refactored functions.

There exist still other tools—based, e.g., on comparing dependence relations [13], abstract semantic trees [23], or (similarly to our work) control-flow graphs [1]—that also aim at practical usability on large projects. However, these tools primarily aim at finding differences between programs and at describing the differences in the best way possible, typically not being able to ignore semantics-preserving changes.

Another field of works in this area aims at identification of refactorings in software: cf., e.g., [22]. These works often introduce or make use of a pre-defined list of refactoring patterns. The best known list is probably the Fowler's catalogue [8], which describes mostly structural refactorings occurring in object-oriented languages. Refactoring lists for low-level procedural languages (such as C) are less common—the most exhaustive that we are aware of is [9]. In our work, we concentrate on supporting a number of patterns from [9], extended by several additional types of semantics-preserving changes occurring in the Linux kernel that are discovered by our own in-depth study of a number of Linux versions.

Principles of semantics-preserving code transformations and variable mapping were successfully used in other works too, e.g., in [7] for so-called on-stack replacement. We, however, combine these basic ideas with multiple further techniques (e.g., advanced pattern matching), allowing us to show equivalence of real-life programs with more complex refactorings.

Apart from comparing the semantics of programs, equivalence checking was successfully applied in hardware (see [12] for an overview), and there exist several industry-level works on translation validation of compilers [20], [26], [11] too. These are, however, far from the problems considered here.

II. CHECKING SEMANTIC EQUIVALENCE

Our goal is to develop an as-precise-as-possible but still highly-scalable method to automatically compare two versions of a function, typically obtained through refactoring, and determine whether the semantics of the function is preserved. The main complexity lies in deciding whether a syntactic change in the function causes a change in the semantics. For high scalability, we concentrate on changes that can be handled on the level of particular instructions or that are instances of several generic types of changes, which we denote as *semantics-preserving change patterns (SCPCs)*.

In particular, our method generically supports so-called *effective* SPCPs, each of which is specified through defining four functions that provide: (1) a test indicating potential applicability of the SPCP at given starting lines of the compared program versions, (2) a way to compute code locations succeeding the given instance of the SPCP, (3) a condition under which the potential SPCP does indeed preserve semantics, and (4) a way to compute which program variables of the two program versions correspond to each other after the SPCP. The idea is that the initial test (Point 1) should be done by a quick and efficient analysis of the compared program functions (which may be quite large), while the method for determining the actual semantic equality (Point 3) may use a more complex algorithm since it is used on a substantially smaller code bounded by the point of detection and the location succeeding the detected pattern instance (determined in Point 2).

A. Program Representation

We represent functions under comparison using *control flow graphs (CFGs)*. In particular, since our tool builds on the LLVM infrastructure, we translate the compared programs into the LLVM intermediate representation (LLVM IR), in which each function can be viewed as a single CFG. The following definition of CFGs is therefore heavily based on the notion of CFGs defined by LLVM IR.

A CFG is composed of basic blocks connected by edges representing program branches. Each basic block consists of a list of instructions. An *instruction* performs an *operation* over a (possibly empty) list of operands and stores its result into a local variable (if it produces some result). An *operand* may be a variable (global or local), a constant, or a function (this

is the case, e.g., for `call` instructions, in which the callee is represented as an operand, or for instructions that assign to function-pointer variables). Each CFG satisfies the *static single assignment (SSA)* property requiring that each variable is assigned to at most once. Therefore, each instruction i assigns its result into a fresh local variable that we denote v_i. Assignments into global variables are done using the `store` instruction. The CFGs are typed—each variable and each constant has a type. The type system is defined by [18].

Furthermore, to simplify the following presentation, we introduce the function *op* that assigns an operation to each instruction. The set of all operations consists of the different kinds of instructions in LLVM IR [18].

Each internal instruction of a basic block has exactly one successor: the instruction immediately following it. A block ends by a so-called *branch instruction* or by a *terminator instruction* that terminates the function, possibly returning a value. A branch instruction may have one or two successors, which are always initial instructions of basic blocks. Branch instructions with one and two instructions are called *unconditional* and *conditional* branches, respectively. For non-branching and unconditional branching instructions, we introduce the function $succ$ that, for each such instruction, defines its successor. For conditional branches, the successor to be followed at runtime is chosen by evaluating a boolean condition that is an operand of the instruction. We thus refer to the two successors as to the *true-case* and *false-case successor* and introduce functions $succT$ and $succF$ that define these successors for each conditional instruction.

B. Function Equality

Before describing the main idea of our method for proving semantic equality, we first define what we mean by two functions being semantically equal. The idea is to find so-called *synchronisation points* in both functions and to check that the code between pairs of corresponding synchronisation points is semantically equal. As we shall see, synchronisation points will *typically* but *not always* be at each instruction. Moreover, we will show that multiple syntactical transformations must often be done in order to allow for the typical per-instruction synchronisation points.

Intuitively, we consider two pieces of code to be *semantically equal* if they both terminate and their execution produces the same output for the same inputs, or they both do not terminate. Here, by output, we mean values of the output variables and the final state of the memory; and by input, values of the input variables and the initial state of the memory. The memory incorporates both the stack and the heap. The input and output variables are subsets of all variables used in the given function. To reflect possible concurrency, we check that both pieces of code being compared use the same synchronisation means (locks, shared memory, etc.) in the same way. We assume that the used synchronisation assures thread safety, and so we consider sequential executions only. Similarly, we check that both compared pieces of code use the same library and system calls in the same way.

Formally, let us have two functions f_1 and f_2, and, for $i \in \{1, 2\}$, let I_i and V_i denote the sets of instructions and variables used in f_i, respectively. We view the *problem of checking semantic equality* of f_1 and f_2 as the problem of finding two sets of synchronisation points $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$ and two synchronisation functions $smap : S_1 \leftrightarrow S_2$ and $varmap : V_1 \leftrightarrow V_2$ that represent mappings of synchronisation points and variables, respectively, between f_1 and f_2. We consider f_1 and f_2 to be semantically equal iff, for any $s_1, s_2 \in S_1$ and any $s'_1, s'_2 \in S_2$, all of the following hold:

1) For each variable $v \in V_1$ that is used but not defined between s_1 and s_2 (a so-called *input variable* of s_1), the variable $varmap(v)$ is used but not defined between $smap(s_1)$ and $smap(s_2)$, i.e., it is an input variable of $smap(s_1)$. An analogical requirement must hold for s'_1, s'_2 and every $v' \in V_2$, using $smap^{-1}$ and $varmap^{-1}$. This requirement may seem quite strict since the functions may have some input variables that do not influence the output (and that could thus be left out from the requirement), but we will use CFG transformations to eliminate such variables beforehand.

2) For each variable $v \in V_1$ defined between s_1 and s_2 and used after s_2 (a so-called *output variable* of s_2), the variable $varmap(v)$ is defined between $smap(s_1)$ and $smap(s_2)$ and used after $smap(s_2)$, i.e., it is an output variable of $smap(s_2)$. Same for s'_1, s'_2 and $v' \in V_2$, using $smap^{-1}$ and $varmap^{-1}$.

3) If the value of each input variable v_{in} at s_1 equals that of $varmap(v_{in})$ at $smap(s_1)$ and the state of the memory at s_1 equals that at $smap(s_1)$, then, if the code between s_1 and s_2 is executed and terminates, an execution of the code between $smap(s_1)$ and $smap(s_2)$ terminates too, and the value of each output variable v_{out} at s_2 equals that of $varmap(v_{out})$ at $smap(s_2)$ and the state of the memory at s_2 equals that at $smap(s_2)$. Likewise for s'_1, s'_2, using $smap^{-1}$ and $varmap^{-1}$.

C. Analysis of Function Equality

We now present the top-level of our algorithm for checking semantic equality of two functions. Using the notions introduced above, our goal is to find the sets S_1 and S_2 of synchronisation points and the mapping functions $smap$ and $varmap$ such that blocks of code between corresponding pairs of synchronisation points are semantically equal. Proving such semantic equality is a rather difficult task, especially for large blocks of code. To cope with this problem, as already indicated, we use the following two main ideas:

1) We *transform* the compared functions so that synchronisation points can be defined as often as possible *per instruction*. Individual instructions are then quite simple to compare—intuitively, they should perform the same operations on operands that are the same or can be mapped to each other.

2) In case a per-instruction synchronisation cannot be achieved for a block of code, we check whether the compared blocks match one of the supported *SPCPs*. If so, we consider the blocks semantically equal too. The check is based on the features of effective SPCPs introduced at the beginning of Section II whose usage is highlighted in the algorithm.

Input: Functions f_1, f_2
Result: *true* if f_1 is semantically equal to f_2, *false* otherwise

1 run transformations of f_1 and f_2
2 **if** $|P_1| \neq |P_2|$ **then return** false
// Initialisation of synchronisation maps
3 $S_1 = \{i^1_{in}\}$, $S_2 = \{i^2_{in}\}$
4 $smap(i^1_{in}) = i^2_{in}$
5 **for** $1 \leq i \leq |P_1|$ **do** $varmap(p^1_i) = p^2_i$
6 **for** $g_1 \in G_1$ **do**
7 $varmap(g_1) = g_2 \in G_2$ s.t. g_1 has the same name as g_2
// Main loop
8 $Q = \{(i^1_{in}, i^2_{in})\}$
9 **while** Q *is not empty* **do**
10 take any (s_1, s_2) from Q
11 $p = detectPattern(s_1, s_2)$
12 **for** *each pair* $(s'_1, s'_2) \in succPair_p(s_1, s_2)$ **do**
13 **if** $(s'_1 \in S_1 \vee s'_2 \in S_2)$ **then**
14 **if** $smap(s'_1) \neq s'_2$ **then return** false
15 **else continue**
16 **if** p *is* none **then** $equal = cmpInst(s_1, s_2)$
17 **else** $equal = compare_p((s_1, s'_1), (s_2, s'_2))$
18 **if** $\neg equal$ **then return** false
// Update synchronisation sets and maps
19 $S_1 = S_1 \cup \{s'_1\}$, $S_2 = S_2 \cup \{s'_2\}$, $smap(s'_1) = s'_2$
20 update $varmap$ according to p
21 insert (s'_1, s'_2) to Q
22 **return** true

Algorithm 1: Checking semantic equivalence of functions

The main workflow of our semantic equivalence checking is shown in Algorithm 1. The algorithm takes two functions f_1 and f_2 as the input. For $i \in \{1, 2\}$, we let P_i denote the list of parameters of f_i, while G_i and C_i denote the sets of all global variables and constants used in f_i, respectively.

First, a number of *code transformations* is applied (Line 1) to the compared functions so that it is easier to define synchronisation points per instruction. These transformations are such that they do not change the semantics of the functions. The most important transformations that we use are constant propagation, redundant instructions elimination, and dead code and dead parameter elimination (since changes in unreachable code do not affect the semantics).

In addition, we run transformations of special calls that occur in LLVM IR, in particular indirect function calls (i.e., calls via function pointers) and calls to assembly code. These calls are replaced by calls to newly generated functions, so-called *abstractions*. *Indirect call abstractions* are function declarations that have the same parameters as the original indirect call and, in addition, a new parameter that represents the called pointer. *Assembly code abstractions* are functions that enclose the called assembly code and promote its parameters into the abstraction function parameters. The purpose of these generated abstractions will be explained later in this section. Finally, some transformations (in particular, function inlining and some related CFG simplifications) are run lazily during SPCP matching (see Section IV-B for more details).

Thanks to the applied transformations, only those parameters that influence the output of the functions are left, and therefore we consider the functions semantically non-equal if they have a different number of parameters (Line 2).

1 **succPair** (s_1, s_2)**:**
2 **if** $op(s_1) = op(s_2) =$ `cond.branch` **then**
3 **return** $(succT(s_1), succT(s_2)), (succF(s_1), succF(s_2))$
4 **else if** $op(s_1) \neq$ `cond.branch` $\wedge\, op(s_2) \neq$ `cond.branch` **then**
5 **return** $(succ(s_1), succ(s_2))$
6 **else** yield error

Algorithm 2: Computing successor synchronisation points

Afterwards, the algorithm starts building the sets of synchronisation points and the mapping functions. Since one of our main goals is high scalability, these are built lazily. Initially, for each function, the synchronisation set only contains the first instruction of the entry basic block (denoted i^1_{in} and i^2_{in} for f_1 and f_2, respectively), and these two instructions are synchronised. The variable mapping is created between pairs of parameters (based on their order—Line 5) and pairs of global variables (based on their name—Lines 6–7).

The main loop of the algorithm works with a queue Q of pairs of synchronisation points. In each iteration, a single pair (s_1, s_2) is taken from the queue. The pair is analysed by the function *detectPattern* that checks whether some pattern p out of the supported SPCPs seems applicable. A special value *none* is returned if no pattern is applicable (forcing a per-instruction comparison). The algorithm can be easily generalised to iterate over multiple patterns possibly applicable at the same time—we have not included this possibility for brevity and also because the applicability of our current patterns is exclusive.

Then, the function $succPair_p(s_1, s_2)$ checks where the next synchronisation points will be placed, i.e., computes the successor synchronisation pairs of (s_1, s_2). We require each pattern to define the behaviour of $succPair_p$. Due to this, the top-level algorithm does not have to search from where to continue the analysis after a successful detection of an instance of a pattern. In our current implementation of the approach, unless a pair of GEP instructions (i.e., LLVM's `getelementptr` instructions—more details in Section IV-A) is encountered, the successor points will simply be at the instructions immediately following (s_1, s_2). The procedure is, however, prepared to easily incorporate dealing with comparisons of other larger code blocks too—should that be needed.

The default implementation of *succPair* is shown in Algorithm 2. It uses the successor functions from Section II-A and returns either one or two pairs of synchronisation points depending on whether a conditional branching follows the current synchronisation points. If some conditional branching appears in one of the functions only, the function ends with an error, causing the comparison to fail as the control flow is different (this case is not included in Algorithm 1 for brevity).

After choosing the successor pair of synchronisation points, we first check whether we have already visited one of the points. If so, we require that the synchronisation points are already mapped to each other, otherwise the synchronisation of the control flow is broken, and the functions are not semantically equal (Lines 13–15).

Subsequently, the blocks of code from the current to the next synchronisation point in each function are checked to be indeed semantically equal. If no pattern is used, each

```
1  cmpInst (i1, i2):
       // Assume (o1^1, ..., o_{n1}^1) and (o1^2, ..., o_{n2}^2) be the
          operand lists of i1 and i2, respectively
2      if op(i1) ≠ op(i2) then return false
3      for 1 ≤ k ≤ n1 do
           // Variables must be mapped
4          if o_k^1 ∈ V1 ∧ varmap(o_k^1) ≠ o_k^2 then return false
           // Constants must be equal
5          else if o_k^1 ∈ C1 ∧ o_k^1 ≠ o_k^2 then return false
           // Functions must be recursively compared
6          else if o_k^1 is a function then
7              if Alg. 1 (o_k^1, o_k^2) = false then return false
8      return true
```

Algorithm 3: Comparing single instructions

of the blocks contains a single instruction, and these are compared using the *cmpInst* function defined in Algorithm 3. The function checks if the instructions perform the same operation on the same or mapped operands.

Moreover, if the compared instructions use functions as operands, we run Algorithm 1 for the functions unless they were compared before. An exception to this are *indirect function calls* and *calls to assembly code*. As already mentioned, we replace such calls by calls to so-called abstraction functions. For an indirect call, only the arguments of the indirect abstraction function call are compared, leading to a comparison of both the original arguments and the function pointers. A comparison of the target functions is started when they are assigned to the function pointers (since that is where the functions appear as operands). As for the assembly abstraction functions, the blocks of assembly code inside them are compared for literal equality instead of using Algorithm 1.

When a potentially applicable SPCP was detected, the comparison of blocks is done using a pattern-specific $compare_p$ function. If the blocks are not compared as equal, the algorithm ends, claiming the functions not to be semantically equal.

After the semantic comparison, the synchronisation sets and maps are updated on Lines 19–20. The variable mapping is updated wrt the SPCP used. If individual instructions were compared (which is the case when no pattern is used as well as when most of the currently supported patterns are used), the update is quite simple as instructions always have at most one output value represented by the fresh local variable created by the instruction. Hence, when two instructions i_1 and i_2 returning a value are compared, the mapping $varmap(v_{i_1}) = v_{i_2}$ between the new local variables is created.

Finally, if all reachable synchronisation points were visited and no inequality has been found, the functions are considered semantically equal.

III. Supported Semantics-Preserving Changes

The list of SPCPs we concentrate on is inspired by two sources: (1) the list of refactoring patterns from [9] and (2) our own extensive study of frequent change patterns that we have performed on multiple past versions of the Linux kernel.

Concerning the list of [9], we observe that our proposed approach implicitly allows us to handle a number of patterns. This is mainly caused by three facts:

1) We use a CFG-based representation of programs, in which some constructions of high-level languages that look different in source code are represented the same way. This allows us to handle the *consolidate-conditional*, *for-into-while*, *while-into-for*, *add-a-typedef*, and *replace-type* patterns.

2) Our method is insensitive to naming of program entities, which allows it to handle many renaming patterns, in particular, *rename variable/constant/user-defined type/function*.

3) The CFG transformations we use effectively "neutralize" the effect of some change patterns. This is in particular the case for (i) dead-code elimination that allows us to handle the *remove-unused-variable/parameter/function* patterns, (ii) constant propagation that allows us to handle the *replace-value-with-constant* pattern, and (iii) memory-to-register promotion that allows us to handle the *add-variable* and *replace-expression-with-variable* patterns.

Hence, we observe that our algorithm in its basic form (without any explicit patterns) allows us to handle 15 out of 29 refactoring patterns mentioned in [9]. From the rest of the patterns, we concentrate on those that occur the most often in real systems code. We support this claim by a study of past versions of the Linux kernel that we present below.

A. Change Patterns in the Linux Kernel

To derive SPCPs common in Linux, we started by analysing all versions of the RHEL 7 kernel from 2014–18 (RHEL 7 was the major RHEL version until 2019). Newer RHEL versions are then used for our experimental evaluation in Section V.

In particular, for pairs of succeeding releases, we compared the semantics of functions from the KABI list. We performed the comparison using our proposed algorithm without any custom patterns and looked for functions marked as non-equal. Such functions thus differ in ways other than what the effect of the 15 implicitly supported patterns described above can be. Note that the encountered changes need not appear directly in the function code—they may be caused, e.g., by a change in a type declaration too. Subsequently, we manually analysed all the obtained differences and identified the most common remaining kinds of changes not affecting the semantics. This way, we obtained the following list of SPCPs. For each discovered SPCP, we enumerate all patterns from [9] that it covers. In addition, we note that many of the SPCPs identified in this section cover additional refactoring patterns that are not mentioned in [9] (e.g., because they are kernel-specific or because they are rather complex).

Changes in structure data types: These include changes in user-defined structures and unions, which often result in a situation when, e.g., an access to the same structure field yields a different memory-access offset. This pattern covers two patterns from [9]: *add/rename a structure field*. We note that this pattern has two variants, depending on whether the change involves changes in nested structures or not (cf. Section IV-A).

Moving code into functions: The code is refactored by moving parts of it into functions called from where the original

TABLE I: Numbers of SPCPs in KABI functions

RHEL versions	KABI funs	Non-dominated changed functions	Data types	Function splitting	Code loc.	Enum values
7.5/7.6	739	112	10	2	2	0
7.4/7.5	734	218	33	13	1	1
7.3/7.4	678	142	6	3	4	0
7.2/7.3	644	223	9	13	2	0
7.1/7.2	551	111	6	4	3	0
7.0/7.1	395	82	2	5	0	2
	Sum	888	66	40	12	3

code was. This covers multiple patterns from [9]: *extract/inline function*, *add/reorder function parameters*.

Changes in a source code location: In the Linux kernel there are macros and built-in functions that allow one to report the file name and the line number of the current code location. In case such a function/macro is used and the location has changed, the semantics stays the same.

Changes of enumeration values: This situation may happen, e.g., when a new value is added into the middle of an enumeration type. In such a case, the rest of the values are shifted and get different numerical values.

Table I shows numbers of appearances of the mentioned SPCPs in the compared RHEL kernel versions The first column states the versions of the RHEL kernel being compared. The second column contains the number of functions on the KABI list. The third column contains the number of functions, either from the KABI list or (directly or indirectly) called from them, that contain a difference and that are not called solely by some function already containing a difference (in other words, we do not descend into callees of functions that contain a difference). Here, note that changes in macros or data types show up in the code of functions in LLVM IR too. The remaining columns give numbers of those of the discovered differences that are caused by the SPCPs described earlier.

Changes in about 13 % of the changed functions are fully covered by the above described patterns, and the semantics of these functions did not change. (Usually, the change corresponds to a single pattern, but a few functions were changed via multiple patterns—hence the given percentage cannot be obtained directly from Table I; we computed it separately.) We also analysed the other changes and discovered that 99 % of them affect the semantics. Changes in the remaining 1 % typically represent more complicated refactoring.

In addition to the above, we also inspected individual commits in the Git repository of the Linux kernel upstream (https://github.com/torvalds/linux) created in past 2 years and concentrated on the commits that are marked as "refactorings" in their commit message (i.e., those that are expected not to change the semantics). We analysed the changes introduced by these commits and identified two additional frequent SPCPs:

Inverse branching conditions: A branching condition is replaced by an inverse condition with the branches swapped. This also applies to loop conditions, thus covering the *while-into-do-while* pattern of [9].

Relocated code: A piece of code is relocated into a different part of a function (e.g., from the beginning of a loop iteration to before the loop). The relocated code is usually independent from the code skipped by the relocation. From the patterns in [9], it covers the *contract/extend variable scope*.

In total, we have thus identified six SPCPs that we consider as important in the given context. These SPCPs cover 9 patterns from [9] but are more general, covering some semantics-preserving changes not covered by [9], yet showing up in the history of the Linux kernel. In addition, we note that we do not cover 5 patterns from [9] as we have never encountered them in our study of Linux. While it should be easy to handle some of them using effective SPCPs (this is applies to 3 patterns related to converting a variable into a pointer and vice-versa), some (in particular *convert global variable into parameter* and *group a set of variables into a new structure*) would require analysing the global state of the compared programs which our algorithm currently does not do.

In the below section, we show that all the above identified six SPCPs can be formulated as effective SPCPs and hence handled by our algorithm.

IV. Handling the Supported SPCPs

Our method for comparing the semantics of two functions is generic in handling effective SPCPs specified by providing the four functions listed in Section II. We now define these functions for the SPCPs that we identified as frequent in the Linux kernel through our empirical study presented in Section III-A. Some of the patterns use default implementations of some of the functions, which were presented in Section II-C. In such cases, we do not discuss the functions for the given pattern. We also propose a specific treatment for the code-relocation SPCP that goes beyond our notion of effective SPCPs.

A. Changes in Structure Data Types

The most common change that we saw in the Linux kernel and that results in different code produced by the compiler while maintaining the semantics is a change of the layout of a user-defined structure type. In C, a *structure type* (a structure or union) consists of a list of *fields*, each field having its *name* and *data type*. When accessing a particular named field f of a variable v, compilers translate the name of f into a numerical *offset*, which is a number that defines the relative offset of the address of f from the starting address at which v lies in memory. In LLVM IR, this is done by the `getelementptr` (GEP) instruction, which takes a pointer and an index of the field and returns a pointer to the required element.

If the layout of a structure type is changed, usage of the type may be affected in multiple ways: e.g., if a field is added to or removed from the middle of the structure type, the indices of all fields up to the end of the type change. As was outlined earlier, we consider two different variants of this pattern, one for a simple change of a field offset and the other for a more complicated change involving changes in nested structures but leading to accessing the same fields in the end.

1) Changed Offset of a Structure Field: When a new field is added into the middle of a structure type, the fields from the point of addition to the end are shifted. Accessing such fields results in different indices generated by the GEP instruction. To deal with such changes, we provide a special way of comparing two GEP instructions used to access a structure field. In particular, we exploit LLVM *debugging information* that contains a mapping of field names to field indices. We then specify the pattern as follows (using the implicit versions of computing successor pairs and updating maps of variables):

Detection condition: $op(s_1) = op(s_2) =$ `gep` and both GEP instructions access a structure field.

Definition of $compare_p$: The comparison is done using a slightly modified version of *cmpInst* from Algorithm 3. When comparing operands that are GEP indices on Line 5, instead of comparing the numerical offsets o_k^1, o_k^2 for equality, we first retrieve the corresponding field names n_k^1, n_k^2 from debugging information. Then, we distinguish four possible situations:

- $o_k^1 = o_k^2, n_k^1 = n_k^2$—the operands are equal.
- $o_k^1 = o_k^2, n_k^1 \neq n_k^2$—check if n_k^2 occurs in the structure type that contains n_k^1. If it does not, then the operands are equal (the field has very probably been renamed), otherwise we treat the operands as not equal.
- $o_k^1 \neq o_k^2, n_k^1 = n_k^2$—the offset has been shifted, but the programmer still accessed the same name. We check whether there is some pointer arithmetic performed on the pointers computed as the results of the instructions s_1 and s_2. If so, the operands are not equal (as the absolute value of o_k^1 or o_k^2 matters), otherwise they are equal.
- $o_k^1 \neq o_k^2, n_k^1 \neq n_k^2$—the operands are not equal.

2) Different Ways to Access the Same Field: There may occur situations when the layout of a structure type is changed in a more complicated way, and the same fields are accessed in a different manner. An example that often happens in the Linux kernel is replacement of a field f by a field u of a `union` type that contains the original field f and some other field g. When accessing the field f through u, the final generated offset is (usually) exactly the same (since f is stored at the beginning of u), but the access is done using one more field (and one more GEP instruction in LLVM IR).

In this case, the same semantics is achieved by a different number of instructions in each of the compared functions. Thus, this pattern must compare larger blocks of instructions.

Detection condition: $op(s_1) = op(s_2) =$ `gep` and there is a sequence of instructions $i_1, \ldots, i_n$ in the first version of the code where:

- $i_1 = s_1$, i.e., the sequence starts from s_1,
- for all $1 \leq k < n$, $succ(i_k) = i_{k+1} \wedge op(i_k) =$ `gep`,
- the sequence has a single input variable—the source pointer accessed by i_1,
- the sequence has a single output variable—the variable v_{i_n} that is the final pointer computed by the sequence, and
- for all $1 \leq k \leq n$, all index operands of i_k are constant.

Moreover, an analogous sequence (possibly of another length) of GEP instructions starts from s_2. We denote by s_1'/s_2' the last instructions of the sequences starting from s_1/s_2, respectively. At least one of the sequences has more than one instruction.

```
1 compare_p (s1, s2):
2   if ¬cmpInst(s1, s2) then
3     if op(s1) = call then inline s1 and simplify
4     if op(s2) = call then inline s2 and simplify
5     insert (s1, s2) to Q // Yield a new comparison of s1, s2
6   return true // The comparison will always continue
```

Algorithm 4: Handling function refactoring in Algorithm 1

Definition of $succPair_p$**:** it returns the single pair of synchronisation points $(succ(s_1'), succ(s_2'))$.

Definition of $compare_p$: As all indices are constant, we can compute the exact memory offset that each instruction would produce. Thus, $compare_p$ returns true iff (1) $varmap$ maps the input variable of s_1 to the input variable of s_2 and (2) the sum of all offsets of all instructions is equal for both sequences.

Method for updating $varmap$**:** $varmap(v_{s_1'}) = v_{s_2'}$.

B. Moving Code into Functions

Another frequent change that preserves semantics is splitting a block of code into pieces and moving (some of them) into some new (or existing) functions. Such functions are then called with appropriate parameters from the place where the original code was. This is a common refactoring process that usually improves readability and simplifies the code.

As our experiments show, when some code is moved into a function, the function usually executes exactly the same operations as the original code. Thus, to handle this kind of changes, it suffices to find a correct synchronisation between instructions of the original code and those of the called function. In order to achieve this, we make use of multiple CFG transformations with the most important being *function inlining*:

Detection condition: $op(s_1) =$ `call` $\vee\, op(s_2) =$ `call`.

Definition of $compare_p$: The implementation of $compare_p$ is shown in Algorithm 4. If the compared instructions are not equal and at least one of them is a call, the call is inlined and a new comparison of the current synchronisation pair is scheduled since the call instruction is replaced by new code.

The above approach is, however, not always sufficient. Sometimes, the code is moved into a function containing more behaviour than the original code, but that behaviour is not executed for the particular call (e.g., by setting some parameter to `false`). The semantics is preserved, but the code produced by inlining contains more instructions than the original code, and so a per-instruction synchronisation cannot be achieved.

Therefore, we perform additional semantics-preserving CFG transformations after the inlining, namely *constant propagation* and *dead code elimination*. Constant propagation may evaluate some conditions to false, and dead code elimination will then remove unreachable code, leaving only the code that can possibly be executed for the particular function call. If that code performs the same operations as the original code, our method is subsequently able to show this.

C. Changes in Enumeration Values

In C, the `enum` keyword allows one to create a list of named constants. Usually, the numerical values themselves are not

important and when they are changed, it is not considered a semantic change. Such changes often occur when a new value is inserted into the middle of an `enum`. All identifiers after the added one then get assigned a different value by the compiler.

Detection condition: s_1 and s_2 are instructions containing a constant operand that corresponds to an `enum` identifier. To detect such a situation, the function analyses debugging information and collects possible mappings of values to `enum` identifiers. Moreover, to determine which *enum* identifiers are used at the C line from which the compared instructions were generated, we use a simple analysis of the C source code associated with the compared LLVM IR.

Definition of $compare_p$: Instructions are compared via *cmpInst*, but if a constant operand corresponding to an *enum* identifier is checked, the identifier string is compared instead of the value[1].

D. Changes in Source Code Location

This semantics-preserving change is specific for the Linux kernel, in particular kernel warning functions. Calls to these functions contain two kinds of information that can be omitted without changing the semantics. First, the warning message is not important. Second, calls to these functions often contain the line number and absolute path to the C source file where the call occurs. Nonetheless, a change of such information does not affect the semantics of the caller function. We handle this by the following SPCP-specific functions:

Detection condition: $op(s_1) = op(s_2) =$ `call` and the same kernel warning function is called.

Definition of $compare_p$: compare the calls using *cmpInst* but do not compare operands that represent a warning message, a line number, or a file name (we identified a list of such operands by manually analysing all kernel warning functions).

E. Inverse Branch Conditions

A common pattern that we identified among kernel refactoring commits covers situations when a branching condition is replaced by its inverse condition. Such a change is semantics-preserving if the true- and false-case successors of the concerned branching instruction are swapped.

Detection condition: s_1 and s_2 are comparison instructions and the produced boolean variables v_{s_1} and v_{s_2} are only used as conditions to branching instructions b_1 and b_2, respectively.

Definition of $compare_p$: compare the instructions using *cmpInst*, but if the instructions evaluate inverse conditions, return *true* and swap the order of successors of b_1 and b_2.

F. Code Relocations

The last semantics-preserving change whose support we consider as crucial wrt our study from Section III-A is relocation of a piece of code into a different part of a function. For the concerned functions to be semantically equal, it is necessary that the relocated code is independent of the code that is skipped by the relocation. Currently, we require the relocated code to be sequential (without branching), but it may be relocated into any part of the same function. Based on our experiments with the Linux kernel presented in Section III-A, this is the most common case for code relocation.

[1]Note that one can construct artificial programs that the described method would claim semantically equal although they are not. This might happen if the programs compare *enum* identifiers with actual constants that they represent. Such constructs, however, break the purpose of enumeration types and do usually not occur in real-world code as our experiments show.

```
18 if ¬equal then
19   if R = [] then // Relocation detection
20     if (R = detectRel(s1, s2, f1)) ≠ [] then  f_r = f_1
21     else if (R = detectRel(s1, s2, f2)) ≠ [] then  f_r = f_2
22     else return false
23   else // Relocation matching
24     if f_r = f_1 then  s_c = s_1,  s_1 = R[0]
25     else  s_c = s_2,  s_2 = R[0]
26     insert (s_1, s_2) to Q
27   continue
28 else // Relocation checking
29   if f_r = f_1 ∧ s_1 is the last instruction of R then
30     if (s_1, s_c) depends on R then return false
31     s_1 = s_c, insert (s_1, s_2) to Q
32   else if f_r = f_2 ∧ s_2 is the last instruction of R then
33     if (s_2, s_c) depends on R then return false
34     s_2 = s_c, insert (s_1, s_2) to Q
```

Algorithm 5: Handling code relocations in Algorithm 1

```
1 detectRel (s_1, s_2, f_r):
2     backup s_1 and s_2,  R = []
3     while op(s_1) ≠ branch ∧ op(s_2) ≠ branch do
4         if cmpInst(s_1, s_2) = equal then return R
5         if f_r = f_1 then append s_1 to R,  s_1 = succ(s_1)
6         else append s_2 to R,  s_2 = succ(s_2)
7     restore s_1 and s_2
8     return []
```

Algorithm 6: Detection of code relocations

Code relocation cannot be covered by our main algorithm via the notion of effective SPCPs. The reason is that to handle it we need multiple interconnected phases as shown below, and, moreover, we want to apply it with the lowest priority (i.e., only when no other SPCP is applicable). Therefore, we handle it as shown in Algorithm 5 that replaces Line 18 in Algorithm 1. The method works in three phases, namely relocation detection, relocation matching, and relocation checking.

Relocation detection is run if the current pair of synchronisation points is compared as non-equal and no potentially relocated block R is being processed. For that, the function *detectRel*, shown in Algorithm 6, is used. It tries to find a synchronisation point in one of the functions under comparison that would match the current synchronisation point in the other function. If such a synchronisation point is found, the block of instructions that were skipped during the search is marked as a *potentially relocated block* R, and s_1 and s_2 are moved such that they are synchronised again. The function in which the relocated block was found is remembered in f_r.

The second phase, *relocation matching*, is run if the current pair of synchronisation points is compared as non-equal and a potentially relocated block R has been previously identified (i.e., the comparison of the two given functions has arrived to the location where R is assumed to be relocated). In this case, the current synchronisation point of f_r is remembered in s_c and then moved back to the first instruction of R, with the pair

TABLE II: Checking semantic equivalence of KABI functions

RHEL versions	KABI functions	DIFFKEMP verdict: equal/not equal/ unknown	Total functions compared	Total LOC compared	Runtime (mm:ss)
7.5/7.6	739	608/125/6	4,954	138,546	08:15
7.6/7.7	769	636/126/7	5,155	144,971	08:46
7.7/7.8	798	611/178/9	5,319	149,030	08:44
8.0/8.1	471	360/86/25	3,374	85,514	07:16
8.1/8.2	521	335/160/26	3,607	87,722	13:33

(s_1, s_2) re-inserted into Q. This way, Algorithm 1 will take care of comparing the relocated code for semantic equality.

Finally, if the equality of the entire block is confirmed, the *relocation checking* phase checks if the code that has been skipped by the relocation (which is the code between the last instruction of R and the instruction s_c from which the comparison will continue) is not data-dependent on the relocated block. Two blocks of code are data dependent if one of the blocks reads a value that is written to by the other block. If the blocks are independent, the relocation is semantically equal, and the comparison continues normally from the remembered synchronisation point s_c.

V. IMPLEMENTATION AND EXPERIMENTS

We have implemented the proposed methods in a tool called DIFFKEMP. The tool is able to automatically compare the semantics of functions from two different versions of a project compiled into LLVM IR. Moreover, for projects written in C (such as the Linux kernel), which are the primary target of DIFFKEMP, it is able to precisely locate the C source symbol (a function, a macro, or a type) that causes the detected semantic difference. Currently, DIFFKEMP supports all versions of Clang/LLVM from 5 to 10. It is distributed as source code[2], an RPM package[3], or a Docker container[4].

We have performed several experiments with DIFFKEMP in order to demonstrate capabilities of the proposed methods. The experiments and the results are presented below.

Experiment 1—KABI functions: As mentioned earlier, the ability of our tool to detect undesirable changes in a software project may be particularly useful for developers of those Linux distributions that aim at stability and compatibility. One of such distributions is the *Red Hat Enterprise Linux (RHEL)* whose KABI functions, cf. Section I, should be stable across minor RHEL releases. A change of the semantics of a KABI function may lead into a compatibility breakage. To show that DIFFKEMP can indeed be helpful to ensure that this does not happen, we used it to check preservation of the semantics of KABI functions on the 7 most recently released versions of RHEL. Table II shows the obtained results.

For each pair of the kernel versions, Column 2 shows the number of KABI symbols that were compared. Column 3 displays the number of functions that are claimed by DIFFKEMP to be semantically equal, non-equal, as well as those whose equality could not be determined. The last kind of result occurred typically in cases in which DIFFKEMP was not able to find the function's definition in the kernel source (some functions are, e.g., generated during kernel compilation from macros). In all experiments, not a single function comparison ended with a tool crash nor the tool timed out.

Furthermore, we manually inspected the functions claimed by DIFFKEMP to be non-equal, and we have found at most units of (for some versions even zero) functions whose semantics seems unchanged. As far as we can say, the changes were mostly security fixes or bug fixes; they should not break anything in applications relying on KABI, but they indeed change the semantics to some degree as correctly announced by DIFFKEMP. Checking the potential impact of such semantic changes is left for the developers. However, the results clearly demonstrate that DIFFKEMP heavily reduces the amount of the human effort needed—while we were able to check the changed functions in a reasonable amount of time (each pair of versions took a few hours, relying on the changes highlighted by DIFFKEMP), this would be impossible if one had to inspect all functions potentially reachable from some KABI symbol—their exact numbers are given in Column 4.

To further manifest the scope of the task performed in this experiment, Column 5 gives the total number of compared lines of C code. The last column gives the execution time that DIFFKEMP, run on a 4 core, 2.6 GHz Intel Xeon Ivy Bridge machine with 8 GB RAM, spent on comparing the given pair of versions compiled to LLVM IR (the compilation time is not included). It shows that DIFFKEMP is able to check semantic equality for a huge amount of code in the order of minutes, which is sufficient, e.g., to integrate it into the continuous integration process.

Experiment 2—Refactoring commits in the Linux kernel: In our second experiment, we apply DIFFKEMP on various refactorings of the Linux kernel that produced code that is syntactically different but should have the same semantics. In particular, we took the last 42 commits (as of May 2020) from the upstream kernel containing the word "factor" or "refactor" in the commit message and compared the semantics of the functions changed by each commit.

Prior to running the experiment, we manually investigated the commits and discovered that, despite the commits are marked as refactorings, many of them actually contain a semantic difference. These are caused, e.g., by added assertions, safety checks (such as a check that a pointer is not `NULL` before dereferencing it), mutex locking, or by the fact that the new version of the function covers more behaviour than the original version did. Out of 22 such functions DIFFKEMP correctly identified all to be semantically not equal. The remaining 20 functions are indeed semantically equal. From them, DIFFKEMP was able to confirm the equality in 50 % of the cases. The rest of the commits contain more complicated refactorings that are beyond the capabilities of the light-weight approach of DIFFKEMP. Such changes seem to require a heavier-weight approach, perhaps relying on more complex formal methods, where, however, the scalability is a problem (as indicated also by Experiment 4 presented later on).

[2]Source code is available from: https://github.com/viktormalik/diffkemp.
[3]https://copr.fedorainfracloud.org/coprs/viktormalik/diffkemp/
[4]https://hub.docker.com/r/viktormalik/diffkemp

TABLE III: Analysis of the *musl* C standard library functions

musl libc versions	`diff` chunks	Semantically equal DIFFKEMP equal	Semantically equal DIFFKEMP not equal (FP)	Semantically not equal DIFFKEMP equal (FN)	Semantically not equal DIFFKEMP not equal
1.1.14/.15	138	107	3	0	28
1.1.15/.16	149	74	4	0	71
1.1.16/.17	163	37	3	0	123
1.1.17/.18	2	0	0	0	2
1.1.18/.19	133	62	0	0	71
1.1.19/.20	199	77	6	0	116
1.1.20/.21	598	470	12	0	116
1.1.21/.22	118	33	1	0	84

All in all, this experiment demonstrates another valuable use case of DIFFKEMP—it can be used as a pre-processing tool when reviewing whether a change is truly semantics-preserving. In such an application, DIFFKEMP is able to handle a significant number of changes (in our case 50 %) that do not need to be reviewed anymore (either manually or by other, more costly, approaches).

Experiment 3—The musl *Standard C library:* Even though the main desired application of DIFFKEMP is on the Linux kernel, the methods it implements are generic and applicable on any project compiled into LLVM IR. We demonstrate this in our third experiment where we check preservation of the semantics of library functions from the C standard library. For simplicity, we chose the *musl libc* implementation since there exists a project that allows this library to be compiled into LLVM IR (https://github.com/SRI-CSL/musllvm). We took the last 9 versions and compared the semantics of all exported functions. Then, we compared the differences identified by DIFFKEMP with the set of all syntactic differences obtained from the versioning system by using the `diff` command (since we built the project for x86, we excluded the differences for non-x86 architectures). Table III shows the results.

For each pair of versions, the table shows the total number of `diff` chunks obtained from the versioning system as described above (Column 2). We use chunks since they give a sufficient granularity (often, a single chunk represents a single change) while their number is quite simple to obtain. The following two columns show the numbers of chunks that do not change the semantics (as determined by a manual analysis) and that were marked by DIFFKEMP as equal (Column 3) or as not equal (Column 4). The results in Column 4 represent false positives (FP) where a chunk was determined to be semantically different although it is not. The number of such results is quite low for each pair of versions[5].

The last two columns of Table III show numbers of semantically different chunks (again, determined manually) that were marked by DIFFKEMP as equal (Column 5) and not equal (Column 6). Column 5 represents false negative (FN) results, where a differing chunk is not identified. We may observe that DIFFKEMP produced no such result.

Overall, this experiment shows that DIFFKEMP can successfully identify a large number of syntactic differences to be semantics-preserving (for some versions, there is more than 80 % of such differences) in a huge real-world project while providing a relatively small number of false results.

Experiment 4—A comparison with other tools: To compare DIFFKEMP with some other existing tool for checking semantic equivalence, we considered the tools mentioned in Section I. As our first choice to compare with, we took the LLREVE tool [15] as it is recent and open-source. Moreover, it runs on Linux and uses LLVM IR, which allowed us to use our tooling for compiling the Linux kernel into LLVM IR. We chose 30 functions from our previous experiments, including both functions where DIFFKEMP succeeds as well as fails to provide a correct result, and compared their semantics using LLREVE. Unfortunately, both the Linux kernel and the *musl* C library use program constructions that LLREVE does not support (such as calls via function pointers, inline assembly code, or floating-point data types). Due to this, LLREVE crashed for a large number of programs. Using various tweaks, we were able to run it on several examples; however, out of these, only a single comparison succeeded (here, LLREVE confirmed the result of DIFFKEMP)—the rest timed out (on a 30-seconds time-out). This demonstrates that the complexity and the size of the comparison is too large for a tool based on heavy-weight formal methods.

As the second tool, we chose SYMDIFF [16] as it is also quite recent and open-source. However, SYMDIFF uses the *Boogie Verification Language* as its internal representation and the available compilers are not able to compile the Linux kernel. We tried to use an LLVM-to-Boogie translation provided by the SMACK formal verification tool [24], but, despite all our efforts, we were unable to get the considered programs into a form that SYMDIFF could handle.

VI. SUMMARY AND FUTURE WORK

We have proposed a light-weight and highly scalable approach to automatically checking semantic equivalence of functions in large-scale industrial programs. Our method is aimed at showing equality of programs containing semantics-preserving changes typically resulting from refactoring. In total, it handles 24 out of 29 patterns from the list of common C refactoring patterns introduced in [9], and, in addition, several other semantics-preserving change patterns common in the Linux kernel. Our experiments with the DIFFKEMP tool implementing our approach show that it is able to successfully analyse large projects, such as the Linux kernel, reasonably fast and with not many false results.

Existing approaches based on heavier-weight formal roots can, in theory, show equality of more functions, but their scalability is limited. However, an interesting direction of future work is to use results from DIFFKEMP to simplify the programs under comparison to small fragments of code considered non-equal by DIFFKEMP and then compare these fragments by some heavier-weight approach.

Acknowledgement: The authors were supported by the project 20-07487S of the Czech Science Foundation and the FIT BUT internal project FIT-S-20-6427.

[5]We admit that the last column of the table may include some (perhaps units of) complicated refactorings for which, during our manual analysis, we could have failed to see that they preserve the semantics.

REFERENCES

[1] APIWATTANAPONG, T., ORSO, A., AND HARROLD, M. J. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th International Conference on Automated Software Engineering* (2004), IEEE, pp. 2–13.

[2] BACKES, J., PERSON, S., RUNGTA, N., AND TKACHUK, O. Regression verification using impact summaries. In *International SPIN workshop on Model Checking Software* (Berlin, Heidelberg, 2013), vol. 7976, Springer Berlin Heidelberg, pp. 99–116.

[3] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (USA, 2008), USENIX Association, p. 209–224.

[4] CHURCHILL, B., PADON, O., SHARMA, R., AND AIKEN, A. Semantic program alignment for equivalence checking. In *Proceedings of the 40th Conference on Programming Language Design and Implementation* (New York, NY, USA, 2019), Association for Computing Machinery, p. 1027–1040.

[5] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2004), Springer Berlin Heidelberg, pp. 168–176.

[6] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), Springer-Verlag, p. 337–340.

[7] D'ELIA, D. C., AND DEMETRESCU, C. On-stack replacement, distilled. In *Proceedings of the 39th Conference on Programming Language Design and Implementation* (2018), pp. 166–180.

[8] FOWLER, M. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[9] GARRIDO, A. *Software refactoring applied to C programming language*. PhD thesis, University of Illinois, Urbana-Champaign, 2000.

[10] GODLIN, B., AND STRICHMAN, O. Regression verification. In *Proceedings of the 46thAnnual Design Automation Conference* (New York, NY, USA, 2009), Association for Computing Machinery, p. 466–471.

[11] HAWBLITZEL, C., LAHIRI, S. K., PAWAR, K., HASHMI, H., GOKBULUT, S., FERNANDO, L., DETLEFS, D., AND WADSWORTH, S. Will you still compile me tomorrow? static cross-version compiler validation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), Association for Computing Machinery, p. 191–201.

[12] HUANG, S.-Y., AND CHENG, K.-T. *Formal Equivalence Checking and Design DeBugging*. Kluwer Academic Publishers, USA, 1998.

[13] JACKSON, D., AND LADD, D. A. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of theInternational Conference on Software Maintenance* (USA, 1994), IEEE Computer Society, p. 243–252.

[14] KAWAGUCHI, M., LAHIRI, S., AND REBELO, H. Conditional equivalence. Tech. Rep. MSR-TR-2010-119, October 2010.

[15] KIEFER, M., KLEBANOV, V., AND ULBRICH, M. Relational program reasoning using compiler IR. In *Proceedings of the 8th Working Conference on Verified Software: Theories, Tools, and Experiments* (Cham, 2016), S. Blazy and M. Chechik, Eds., Springer International Publishing, pp. 149–165.

[16] LAHIRI, S., HAWBLITZEL, C., KAWAGUCHI, M., AND REBELO, H. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification (Tool description)* (Berlin, Heidelberg, 2012), Springer Berlin Heidelberg, pp. 712–717.

[17] LAHIRI, S. K., VASWANI, K., AND HOARE, C. A. R. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (New York, NY, USA, 2010), Association for Computing Machinery, p. 201–204.

[18] LATTNER, C., AND ADVE, V. LLVM language reference manual, 2020.

[19] NEAMTIU, I., FOSTER, J. S., AND HICKS, M. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories* (2005), pp. 1–5.

[20] NECULA, G. C. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), Association for Computing Machinery, p. 83–94.

[21] PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. Differential symbolic execution. In *Proceedings of the 16thACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2008), Association for Computing Machinery, p. 226–237.

[22] PRETE, K., RACHATASUMRIT, N., SUDAN, N., AND KIM, M. Template-based reconstruction of complex refactorings. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance* (2010), pp. 1–10.

[23] RAGHAVAN, S., ROHANA, R., LEON, D., PODGURSKI, A., AND AUGUSTINE, V. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th International Conference on Software Maintenance* (USA, 2004), IEEE Computer Society, pp. 188–197.

[24] RAKAMARIĆ, Z., AND EMMI, M. SMACK: Decoupling source language details from verifier implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification* (Cham, 2014), Springer International Publishing, pp. 106–113.

[25] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd International Conference on Computer Aided Verification* (Berlin, Heidelberg, 2011), Springer-Verlag, p. 669–685.

[26] TRISTAN, J.-B., GOVEREAU, P., AND MORRISETT, G. Evaluating value-graph translation validation for LLVM. vol. 46, pp. 295–305.