DiffKemp: Automatic Analysis of Semantic Differences in Kernel Versions

Viktor Malík [1,2] Tomáš Vojnar [1] Petr Šilling [1]

[1]Brno University of Technology [2]Red Hat Czech

Introduction

DiffKemp is a framework for **automatic static analysis of semantic differences** between different versions of large-scale C projects. Our main target is the Linux kernel, in particular the kernel of **Red Hat Enterprise Linux (RHEL)**.

The RHEL kernel contains a list of functions, so-called **Kernel Application Binary Interface (KABI)**, which are guaranteed to remain stable across a single major RHEL release. The purpose of DiffKemp is to automate checking of semantic stability of these functions, allowing the process of the kernel development and deployment to be more efficient and reliable.

The approach of DiffKemp is based on compiling the code to be compared into LLVM IR followed by using a **combination of light-weight program transformations and pattern matching** to analyse the code. Thanks to this unique method, DiffKemp is able to analyse semantic equivalence of code of the size of the Linux kernel in the order of minutes while providing a very low number of false positive results. To the best of our knowledge, this is beyond capabilities of any other existing approach.

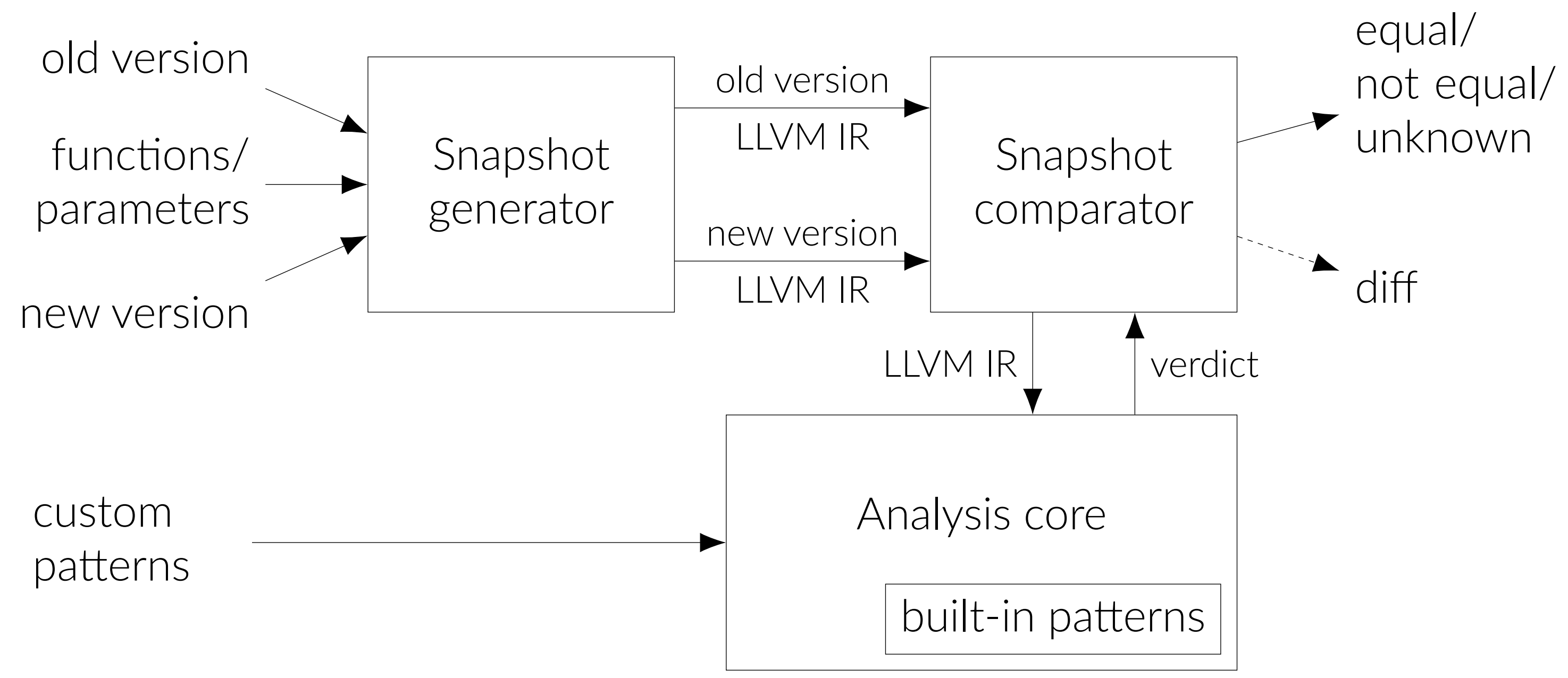
General Approach

The analysis in DiffKemp is built on several core concepts:

- Compared versions are compiled into the **LLVM Intermediate Representation (LLVM IR)** to make the comparison simpler.
- Where possible, versions are compared **instruction-by-instruction** which is sufficient for the (usually large) parts that are syntactically equal.
- Programs are **pre-processed** with **semantics-preserving transformations** (constant propagation, dead code elimination, ...) which allow the instruction-by-instruction comparison to succeed more often.
- If differences are still observed, DiffKemp checks if they correspond to one of the pre-defined **semantics-preserving change patterns**. If so, the observed changes are claimed as semantics-preserving.

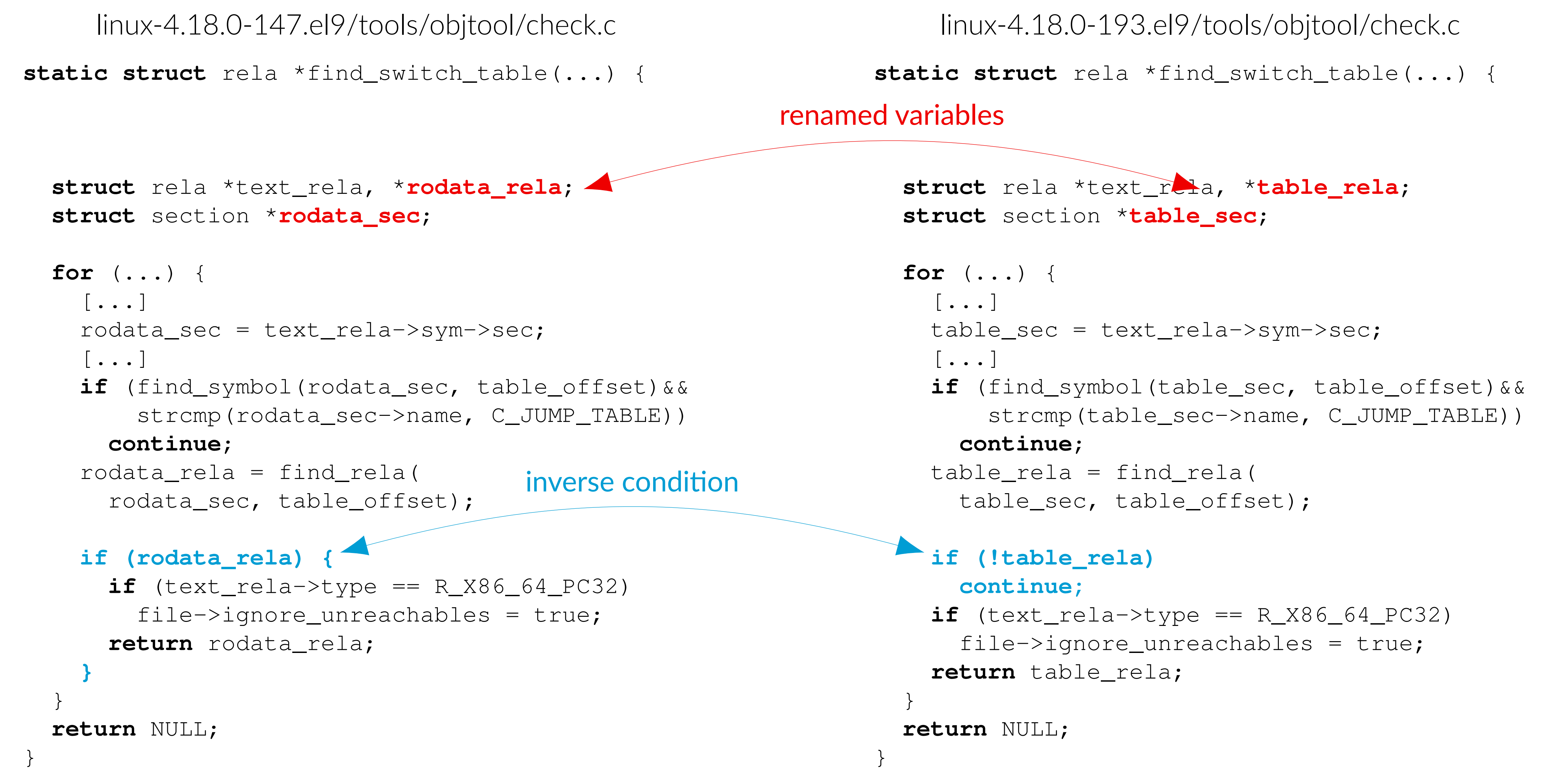
DiffKemp Architecture

On its input, DiffKemp takes sources of the **compared versions** and a **list of functions or system parameters** whose semantics should be compared. On its output, it provides the **verdict** for each function/parameter (i.e., whether its semantics changed or not) and, if a semantic change was detected, the **diff** that caused it.



A Motivation Example

Are the following functions semantically equal?



linux-4.18.0-147.el9/tools/objtool/check.c

```c
static struct rela *find_switch_table(...) {

  struct rela *text_rela, *rodata_rela;
  struct section *rodata_sec;

  for (...) {
    [...]
    rodata_sec = text_rela->sym->sec;
    [...]
    if (find_symbol(rodata_sec, table_offset)&&
        strcmp(rodata_sec->name, C_JUMP_TABLE))
      continue;
    rodata_rela = find_rela(
      rodata_sec, table_offset);

    if (rodata_rela) {
      if (text_rela->type == R_X86_64_PC32)
        file->ignore_unreachables = true;
      return rodata_rela;
    }
  }
  return NULL;
}
```

linux-4.18.0-193.el9/tools/objtool/check.c

```c
static struct rela *find_switch_table(...) {

  struct rela *text_rela, *table_rela;
  struct section *table_sec;

  for (...) {
    [...]
    table_sec = text_rela->sym->sec;
    [...]
    if (find_symbol(table_sec, table_offset)&&
        strcmp(table_sec->name, C_JUMP_TABLE))
      continue;
    table_rela = find_rela(
      table_sec, table_offset);

    if (!table_rela)
      continue;
    if (text_rela->type == R_X86_64_PC32)
      file->ignore_unreachables = true;
    return table_rela;
  }
  return NULL;
}
```

Yes, they are! But how to automatically check that?

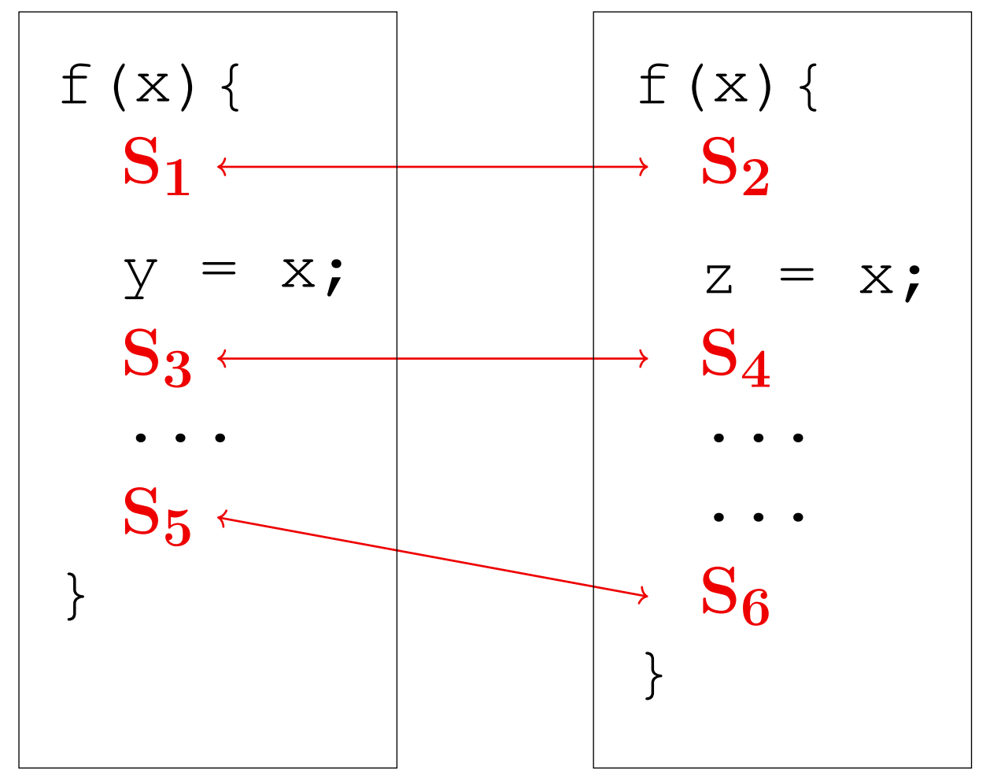
DiffKemp allows to confirm the equivalence of the functions using several concepts:

- The semantic equivalence of the renamed variables is confirmed thanks to using **LLVM IR** and thanks to **tracking semantically equivalent variables** between the compared versions.
- The semantic equivalence of the inversed condition is confirmed thanks to a **built-in semantics-preserving change pattern**.

The Basic Comparison Algorithm

The main semantic comparison algorithm of DiffKemp is built on several basic ideas:

- Compared functions are split into smaller **chunks** using the same number of **synchronisation points**.
- Synchronisation points denote places where the functions are (or should be) in **semantically equivalent** states.
- The code between corresponding pairs of synchronisation points is checked for semantic equality.
- Placing of synchronisation points:
  - Always done **lazily** to maintain high scalability of the approach.
  - Where possible, synchronisation points are placed **after every instruction**.
  - Where not possible, DiffKemp tries to apply one of the **pre-defined patterns** and places the following points **after the matched code**.



an illustration of synchronisation points

Supported Change Patterns

Built-in semantics-preserving change patterns:

Changes in structure data types	Covers changes in user-defined structures such as additions, removals, or renamings of fields and changes in the structure size.
Splitting code into functions	Covers situations when a part of a function is moved into a new or an existing function which is called from the place where the original code was.
Inverse branching conditions	Covers situations when a branching condition is replaced by an inverse condition with the branches swapped.
Code relocation	Currently the most complex pattern which covers situations when a piece of code is moved to a different part of a function (e.g. from inside a loop to before the loop). The relocated code must be independent from the code skipped by the relocation.
Changes in source code location	Pattern specific to the Linux kernel which covers invocations of special kernel functions and macros that report the file name and location of the invocation.
Changes in enumeration values	Covers situations when a new value is added into an enumeration type, causing the remaining values to be shifted.

Custom user-defined change patterns:

DiffKemp allows users to define their **own patterns** of changes that they wish to ignore (evaluate as semantically equal) during the comparison process. Note that these do not necessarily have to be semantics-preserving changes but also semantics-altering changes which are known to be safe and therefore do not have to pollute the output of the comparison report.

Custom change patterns are represented using **parametrized control flow graphs**, and DiffKemp uses a specialized **graph-matching procedure** to recognise the patterns inside the compared programs [1].

An Experimental Evaluation

We compared the semantics of KABI functions for the most recent RHEL kernels:

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

The results show that DiffKemp is able to compare thousands of functions in the order of minutes while providing small numbers of false results (verified manually).

References

[1] Viktor Malík, Petr Šilling, and Tomáš Vojnar. Applying custom patterns in semantic equality analysis. In *Proc. of the NETYS 2022*, pages 265–282. Springer, 2022.

[2] Viktor Malík and Tomáš Vojnar. Automatically checking semantic equivalence between versions of large-scale C projects. In *Proc. of ICST 2021*, pages 329–339. IEEE, 2021.

https://github.com/viktormalik/diffkemp/

The project is supported by Red Hat Research, Czech Science Foundation projects 20-07487S and 23-06506S, FIT BUT internal project FIT-S-23-8151, and Europe Horizon project CHESS.

vmalik@redhat.com