DiffKemp: Automatic Analysis of Semantic Differences in Kernel Versions

Viktor Malik 1,2, Tomáš Vojnar 3, Petr Šilling 2
1Brno University of Technology 2Red 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 converting the code to be compared into LLVM Intermediate Representation (LLVM IR) followed by using a combination of light-weight program transformations and graph-matching procedure to analyze the code. Thanks to this unique method, DiffKemp is able to analyze 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 checked. 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?

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

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

The results show that DiffKemp is able to compare thousands of functions in the new version of the Linux kernel against the old version in very short time.

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) 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.

Supported Change Patterns

Built-in semantics-preserving change patterns:

- Changes in structure type definitions.
- Functions.

Splitting code into functions.

Inverse branching conditions.

Code Fernandez. 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 recognize the patterns inside the compared programs [1].

An Experimental Evaluation

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

<table>
<thead>
<tr>
<th>RHEL kernel</th>
<th>DiffKemp verdicts</th>
<th>Task function: Total (LLVM + number of function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6/7.7</td>
<td>729</td>
<td>608/125/6</td>
</tr>
<tr>
<td>7.4/7.5</td>
<td>789</td>
<td>630/126/7</td>
</tr>
<tr>
<td>7.7/7.8</td>
<td>788</td>
<td>611/178/9</td>
</tr>
<tr>
<td>8.0/8.1</td>
<td>471</td>
<td>360/68/25</td>
</tr>
<tr>
<td>8.1/8.2</td>
<td>521</td>
<td>353/160/26</td>
</tr>
</tbody>
</table>

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

References


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

The project is supported by Red Hat Research, Czech Science Foundation projects 20–07487S and 23–06508S, FIT BUT internal project FIT-S-23-8151, and Europe Horizon project CHESS.

vmalik@redhat.com