
**BRNO UNIVERSITY OF TECHNOLOGY**
**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**FACULTY OF INFORMATION TECHNOLOGY**
**FAKULTA INFORMAČNÍCH TECHNOLOGIÍ**

**DEPARTMENT OF INTELLIGENT SYSTEMS**
**ÚSTAV INTELIGENTNÍCH SYSTÉMŮ**


# AUTOMATION OF EVALUATION OF DIFFKEMP ON OPEN-SOURCE PROJECTS

**AUTOMATIZACE VYHODNOCENÍ NÁSTROJE DIFFKEMP NA OPEN-SOURCE PROJEKTECH**

**MASTER'S THESIS**
**DIPLOMOVÁ PRÁCE**


**AUTHOR** **LUKÁŠ PETR**
**AUTOR PRÁCE**

**SUPERVISOR** **Ing. VIKTOR MALÍK, Ph.D.**
**VEDOUCÍ PRÁCE**



**BRNO 2025**

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

# Master's Thesis Assignment

164987

| | |
|---|---|
| Institut: | Department of Intelligent Systems (DITS) |
| Student: | **Petr Lukáš, Bc.** |
| Programme: | Information Technology and Artificial Intelligence |
| Specialization: | Information Systems and Databases |
| Title: | **Automation of evaluation of DiffKemp on open-source projects** |
| Category: | Software analysis and testing |
| Academic year: | 2024/25 |

Assignment:

1. Get acquainted with DiffKemp, a tool for automatic static analysis of semantic equivalence between versions of C projects. Focus on exploring various large-scale open-source projects and benchmarks that DiffKemp has been evaluated on in the past.
2. Propose and implement a solution for automatic evaluation of DiffKemp on selected open-source projects and benchmarks which will allow to assess the impact of new DiffKemp features. Integrate the solution with GitHub to allow executing it on pull requests either automatically or on demand.
3. Propose and implement a solution for automatic execution of DiffKemp on new versions or patches of selected open-source projects. Deploy the solution in a way which will simplify reviewing of the findings and identification of true and false positive results.
4. Evaluate the created automation on the history of at least 2 selected projects (for example the Linux kernel, various system libraries, etc.). Discuss the discovered issues in both the analysed projects and DiffKemp itself.

Literature:


- Malík, V., Vojnar, T.: Automatically checking semantic equivalence between versions of large-scale C projects. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). pp. 329–339. IEEE (2021)
- S. Badihi, Y. Li and J. Rubin: EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In: 2021 18th IEEE/ACM International Conference on Mining Software Repositories (MSR). pp. 610-614. IEEE (2021)


Requirements for the semestral defence:
The first point and the design part of the second point of the assignment.

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

| | |
|---|---|
| Supervisor: | **Malík Viktor, Ing., Ph.D.** |
| Consultant: | Ing. František Nečas |
| Head of Department: | Kočí Radek, Ing., Ph.D. |
| Beginning of work: | 1.11.2024 |
| Submission deadline: | 21.5.2025 |
| Approval date: | 31.10.2024 |

# Abstract

The goal of this work is to propose and create two automation for DiffKemp, a semantic equivalence checking tool, which would be useful for its development. The first aim is to create an automation that would simplify the impact assessment of new DiffKemp features. The second goal is to create a solution that would automatically evaluate new versions of open-source projects with DiffKemp, simplify reviewing of the results, allow classification of their correctness, and record them. The first goal is achieved and is documented in this thesis, the second part is implemented but not documented. The EqBench dataset and the RHEL kernel projects were chosen for the automatization. Comparing the results of these projects gained by using the original DiffKemp version with results attained by using the version containing the new feature was selected as the assessment method. A GitHub App was created using the Probot framework and Podman container for the automation. Finally, the created solution was evaluated on previous features that were added to DiffKemp.



# Abstrakt

Cílem této práce je navrhnout a vytvořit dvě automatizace pro DiffKemp, nástroj kontrolující sémantickou rovnost, které by byly užitečné pro jeho vývoj. Prvním cílem je automatizovat vyhodnocování dopadu nových vylepšení tohoto nástroje. Druhým úkolem je vytvořit řešení, které by automaticky vyhodnotilo nové verze projektů s otevřeným kódem pomocí nástroje DiffKemp, ulehčilo vyhodnocení jejich výsledků, umožnilo je klasifikovat a uložit. První cíl byl dosažen a je zdokumentován v této práci, druhá část je naimplementována, ale není zdokumentována. Pro automatizaci byla vybrána datová sada EqBench a jádra systému RHEL. Pro vyhodnocení dopadu byla vybrána metoda, která porovnává výsledky těchto projektů získané použitím původní verze nástroje DiffKemp s vylepšenou verzí. Byla vytvořena GitHub aplikace s využitím platformy Probot a konejneru Podman. Vytvořené řešení bylo nakonec vyhodnoceno na předchozích vylepšeních, která byla do DiffKempu přidána.



# Keywords

DiffKemp, semantic differences, automation, software development, pull requests, evaluation, continuos integration, Linux kernel, EqBench benchmark, GitHub App

# Klíčová slova

DiffKemp, sémantické rozdíly, automatizace, vývoj softwaru, požadavek na stažení, vyhodnocení, průběžná integrace, Linuxové jádro, sbírka programů EqBench, GitHub App


# Reference

PETR, Lukáš. *Automation of evaluation of DiffKemp on open-source projects.* Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík, Ph.D.

# Rozšířený abstrakt

Od některých knihoven se očekává, že si zachovají konzistentní chování napříč více verzemi i v případě, že došlo k refaktorizaci kódu. Nástroj jako je DiffKemp je tu od toho, aby s tím pomohl a to pomocí kontroly sémantické rovnosti rozhraní mezi dvěma verzemi knihoven. Nástroj není dokonalý a někdy poskytuje nesprávné výsledky – tvrdí uživateli, že se rozhraní sémanticky změnilo, i když se nezměnilo. Je to proto, že DiffKemp v současné době podporuje pouze podmnožinu refaktorizačních vzorů. Nástroj se stále vyvíjí, průběžně se přidává podpora nových vzorů a další vylepšení.

Když je do DiffKempu přidáno **nové vylepšení, je důležité vyhodnotit, že pozitivně ovlivňuje proces porovnávání**. V minulosti bylo použito více metod, metrik a projektů k posouzení přínosu nových vylepšení. Tyto experimenty byly prováděny ručně.

Prvním cílem této práce je automatizovat proces vyhodnocování nových funkcí, aby se zjednodušilo vyhodnocování všech významných/podstatných změn přidaných do nástroje DiffKemp. Cílem je automaticky spustit DiffKemp a poskytnout recenzentovi zpětnou vazbu o tom, jak dané změny ovlivňují výkonnost (správnost) porovnání/vyhodnocení prováděného nástrojem DiffKemp.

Za účelem automatizace procesu vyhodnocování a poskytování zpětné vazby, tato práce studuje a analyzuje experimenty provedené v předhozích pracích a projekty, které v nich byly použité. Na základě tohoto průzkumu je navržena metoda a vybrány projekty pro hodnocení dopadu nových změn. Dále je pak vybrána technologie pro implementaci a je vytvořen návrh a popsána implementace. Nakonec je vytvořené řešení otestována a je vyhodnocena na vylepšeních, která byla přidána do DiffKempu v minulosti.

Vzhledem k tomu, že DiffKemp zatím není aktivně používaný, je také vhodné **sledovat, jak nástroj bude užitečný v realném provozu a co by se mělo vylepšit**. To by vyžadovalo pravidelné spouštění DiffKempu na nových verzích vybraných projektů a zjišťování, zda provádí vyhodnocení správně. Toto by pomohlo vývojářům DiffKempu:

- identifikovat chyby (např. sémanticky neekvivalentní funkce označené nástrojem jako ekvivalentní), které by mohly být opraveny,
- odhalit refaktorizační vzory, které by měly být přidány, protože se objevují opakovaně v kódech reálných projektů,
- zjistit, jaké funkce nástroji chybí a jak by mohlo být vylepšeno používání nástroje (např. přidání dodatečných informací do výstupu, které by zjednodušily interpretaci výsledků), aby byl nástroj pro uživatele více přívětivý.

Druhým cílem této práce je částečně automatizovat tento proces tím, že se nástroj DiffKemp bude spouštět automaticky na vybraných projektech (s otevřeným zdrojovým kódem) a prezentovat výsledky takovým způsobem, které by zjednodušilo ruční kontrolu. To by umožnilo vývojářům rychle zjistit, která verze byla porovnávána, jaké výsledky byly získány a zda se tyto výsledky jeví jako správné.

# Automation of evaluation of DiffKemp on open-source projects

## Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Viktor Malík, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Lukáš Petr
May 21, 2025


## Acknowledgements

I would like to thank my supervisor Ing. Viktor Malík, Ph.D. for his support, guidance, patience and advice during the work on this thesis. I would also like to thank Ing. František Nečas for his advice, as well as my friend Daniel Kříž and my family for their support.

# Contents

# Chapter 1

# Introduction

Some libraries are expected to maintain consistent behavior across multiple versions, even after refactoring. A tool like DiffKemp can help verify this by semantically comparing a library's interface between two versions. The tool is not perfect and sometimes provides incorrect results – indicating to the user that the interface semantically changed even if it did not. This is because DiffKemp currently supports only a subset of refactoring patterns. The tool is still in development, continuously adding support for new patterns and other features.

When a new feature is added to DiffKemp, it is important to **evaluate that the feature positively impacts the comparison process**. In the past, multiple methods, metrics, and projects have been used to assess the contribution of new features. These experiments were performed manually.

The first goal of this thesis is to automate the new features evaluation process, to simplify the evaluation of all the major/vital changes made to DiffKemp. The aim is to automatically run DiffKemp and provide feedback to a reviewer about how the changes affect the performance (correctness) of the evaluation performed by DiffKemp.

To automate the evaluation and feedback process, this thesis studies and analyzes the previous experiments and projects used in those. Based on that, a method and projects are selected for the impact assessment of new features. Then, the technology for implementing the automation is selected. After that, the automation is designed and implemented. Finally, the solution is evaluated on selected feature proposals that were created in the past.

Since DiffKemp is not yet actively used, it is also necessary to **monitor how useful it would be in production and what should be improved**. This would require regular running of DiffKemp on new versions of selected projects and determining whether its semantic equivalence evaluations are correct. This would help DiffKemp's developers:

- identify bugs (e.g., semantically non-equal functions marked by DiffKemp as equal), which could be fixed,
- discover refactoring patterns that should be added because they regularly appear in real codebases,
- determine missing features and usability improvements (e.g., additional information to report to make the interpretation of results more simple) that could make the tool more accessible to end users.

The second goal of the thesis is to partially automate this process by automatically running DiffKemp on selected open-source projects and organizing the results in a way

that simplifies manual review. This would allow developers to quickly see which version was compared, what results were produced, and whether those results appear correct.

This work is organized as follows. Chapter 2 is an introduction to DiffKemp, describing the basic concepts and analysis evaluations and experiments done with the tool in the past. Then, in Chapter 3, we focus on (a) development automation using continuous integration, (b) DiffKemp development process and automatizations used in it, and (c) script created for automatization of usage DiffKemp for some projects. After that, in Chapter 5, the created solution is evaluated. Finally, in Section 4, we focus on solving the first goal of this thesis – requirements for the automation are collected, and the solution is designed and implemented.

# Chapter 2

# DiffKemp

DiffKemp[1] is a tool for automatic static analysis of two versions of the same program. It checks if the semantics of the specified symbols changed between the two versions. If the tool identifies semantic changes, it reports the locations where the semantic differences were found. The primary purpose of DiffKemp is to check that a refactoring of the code did not cause semantic differences. The tool can be used for example:

- for simplifying code reviews by eliminating parts of code whose semantics did not change or
- for verification of stability of library API functions, which should remain stable across the library versions.

DiffKemp was initially developed to compare Red Hat Enterprise Linux (RHEL) kernel versions. Later, it was extended to compare versions of any project written in C which uses `make` as a build system [9]. DiffKemp first compiles the project's versions to LLVM IR (the intermediate language of the Clang compiler), and then it compares the versions on the level of the LLVM IR instructions.

There exist many tools for static semantic equivalence checking based on various approaches and techniques. Most tools use formal methods, for example Clever [18] (uses symbolic execution with SMT solving), Rêve [3] (uses reduction to Horn constraints solved by SMT solver), and many more, some of them are mentioned in [16]. There are also fast and lightweight tools like Unix `diff`, but these tools do not handle very well even light semantic changes. DiffKemp tries to be in the middle by combining the approaches – where possible it compares program versions using lightweight instruction-by-instruction comparison of their LLVM IR code representation. Otherwise, it uses more complex methods like *code change pattern matching* and SMT solving. This approach makes DiffKemp highly scalable, allowing it to correctly evaluate semantic equivalency of more complicated refactorings than `diff` can handle, but also be able to compare thousands of functions in minutes.

This chapter introduces the most important concepts in DiffKemp. Section 2.1 describes the main parts of DiffKemp and explains how DiffKemp is used. Section 2.2 gives more information about the comparison process. Finally, Section 2.3 mentions evaluations done with DiffKemp in the past, projects used for the evaluations, and metrics considered by them.

[1] Available from https://github.com/diffkemp/diffkemp/

## 2.1 DiffKemp Architecture

DiffKemp consists of three main components: (1) snapshot generator, (2) snapshot comparator, and (3) result viewer. Typically, DiffKemp is used in the following way:

1. First, it is necessary to have two versions of a project and a list of symbols which we want to compare for semantic equality.
2. Using the snapshot generator, the versions are transformed into so-called snapshots. A *snapshot* contains one version of the project compiled into LLVM IR files and a metadata file.
3. The selected symbols are compared within the snapshots using the *snapshot comparator*. The comparator informs about symbols that were evaluated as semantically changed.
4. Finally, the locations where semantic differences were found can be visualized using the *result viewer*.

The following subsections break down the individual parts of DiffKemp in more detail.

### Snapshot Generator

The first part of DiffKemp creates a snapshot of the project's source code. DiffKemp provides multiple build commands that can create snapshots from different sources. There are two main kinds of snapshot generators:

- Generator for creating snapshots from project written in C that uses the `make` tool as the build system.
- Generator specific for creating snapshots from the source code of the Linux kernel.

All generators take a list of symbols. The symbols will later be compared with symbols in the second version of the project by the snapshot comparator. Generally, the expected symbols are names of functions, but in the case of the snapshot generator for the Linux kernel, there is an exception. For the kernel, it is possible to specify not only a list of functions but also a list of *sysctl parameters* which should be compared instead. Sysctl parameters are kernel parameters that can be set during the operating system runtime without the need to recompile the kernel and are used to control system settings that affect its functionality, security, and performance [1]. The snapshot generator uses the symbol list to compile necessary source files (e.g., containing the symbol definition) to LLVM IR files.

**LLVM IR** is a lightweight low-level typed language. It allows to cleanly represent multiple high-level languages [25]. It is a representation that allows efficient transformations and analysis and is used, e.g., by the Clang compiler. LLVM IR file is also called an LLVM module and is usually equivalent to one C source file. The module contains functions, global variables, and metadata. An LLVM function has a form of a control flow graph consisting of basic blocks connected by conditional and unconditional branching instructions. Functions start with one entry basic block and end with one or more return instructions. A basic block contains a sequence of assembly-like instructions ending with a terminator instruction (branching or return instruction). Computation instructions always save result of the operation to a new register variable, this property is called SSA (Single Static Assignment) and it simplifies analysis and transformations.

The LLVM modules created by the snapshot comparator are then optimized using certain *optimization passes* to simplify the resulting LLVM IR code and make the subsequent comparison more effective by making the compared programs more similar to each other. The passes include, e.g., dead code elimination, which eliminates unreachable code, and memory-to-register promotion, which eliminates unnecessary load and store instructions by leaving the values in register variables. The latter makes it possible to take advantage of the SSA form in the comparison process.

The result of snapshot generation is a directory containing the LLVM IR modules, the original source files, and a metadata file that maps symbols to LLVM IR files in which the symbols' definitions are located.

### Snapshot Comparator

When two versions of the same program are built into snapshots, it is possible to use the snapshot comparator command to analyze the semantic equivalence of the symbols specified during snapshot creation. In the case of sysctl parameters, functions that handle the setting of the parameters and all functions using the data variable that the sysctl parameter sets are compared.

When using the snapshot comparator, the user can choose which DiffKemp's built-in patterns of changes (code relocation, variable grouping, ...) should be considered when running the comparison. By default, all supported *semantic preserving change patterns* (SPCPs) are considered. Users can also specify custom *code change patterns* (CCPs) to define their own kind of changes that they want to be considered when running the comparison and not to be reported. The defined CCPs do not have to be necessarily semantic preserving but can describe changes that should not be considered – e.g., security patches like cleaning memory. The CCPs can be written either in the LLVM IR itself [30] or in the C language [11].

The comparison itself is managed by a component of DiffKemp called SimpLL, which is a library written in C++. The selected functions are compared, including the functions they call. The comparison process is described in more detail in Section 2.2.

Snapshot comparator reports for which user-selected symbols were discovered semantic differences and locations where the differences were found. DiffKemp is not, of course, a perfect tool, so it can evaluate semantic equality incorrectly. We can divide the results into four categories based on the expected result and the result provided by DiffKemp:

- **True negatives**: correct results for semantically equal symbols.
- **False positives**: incorrect results for semantically equal symbols.
- **True positives**: correct results for semantically not equal symbols.
- **False negatives**: incorrect results for semantically not equal symbols.

DiffKemp does not handle all possible refactorings, so it usually provides a certain number of false positives, but the way it is implemented, it should not give any false negatives. Of course, the implementation can contain bugs, so it is theoretically also possible that the result will contain some false negatives.

### Result Viewer

The last part of DiffKemp is the result viewer. It is a web application that allows the user to see the locations of the found semantic differences inside the source code of the projects. The viewer shows the source code of both project versions next to each other, enabling the user to evaluate the found differences by himself. Because the differences are usually not found directly in the compared functions (e.g., the API interface) but in some of the called functions, the viewer provides a context in the form of call stacks representing a relationship between the compared symbols and the symbols where the difference was found. It also allows the user to view the definitions of individual symbols.

## 2.2 Comparison Process

DiffKemp compares two project versions or, more precisely, snapshots representing the versions using the SimpLL library. The library is used to compare the selected functions (or the identified functions in the case of sysctl parameters). It always compares the two versions of the same function in the following way:

1. Firstly, **analysis and transformation passes** are run over the modules containing the functions. The purpose of this is to simplify the LLVM code of the functions such that they can be compared more easily. For example, in the case of a sysctl parameter comparison, a slicing pass is run, which keeps in the compared functions only the parts of the function that are influenced by the global variable that the sysctl parameter sets.

2. After that, the comparison begins.

   (a) The comparison is done **instruction-by-instruction**. The process includes creating mappings for the function parameters and variables between the two versions of the functions. The process is complicated and is more formally described in [16]. In short, the instruction-by-instruction comparison takes one instruction from both versions at a time and checks that they perform the same operation over the same or semantically equivalent operands. If the instructions are compared as equal, it continues with the following instructions. If the functions are calling other functions, the comparison continues by comparison of the called functions.

   (b) In case the instruction-by-instruction fails, it is checked if it is not possible to match the change with some enabled **semantic preserving change pattern**. If the match is successful, the comparison continues by comparing instruction-by-instruction functions after the match. Currently, the built-in SPCPs support changes in structure data types, moving part of code into functions, code relocations, inverting of branching conditions, variable grouping, and binary operations reordering. The patterns are described in more detail in [16] and [31].

   (c) If both instruction-by-instruction and SPCPs matching fails, it is checked if the changes do not correspond to changes described by some of the **custom code change patterns** supplied by the user. The process of this matching is formally described in [30]. If the changes correspond to a pattern, the comparison continues with instructions after the changes described by the pattern.

(d) Newly, if none of the previous is successful, DiffKemp tries to check if the difference is limited to sequential (non-branching) blocks of code and, if so, encodes the semantics of the differing blocks into a first-order logic formula and uses an **SMT solver** to check whether the blocks are equal. This extension is described in detail in [20]. If the blocks are compared as equal, the comparison process continues.

3. If the comparison fails, the location of the failure, i.e., of the semantic difference, is recorded and later provided to the user in the form of report.

## 2.3 Evaluations Done With DiffKemp in the Past

This section focuses on evaluations and experiments that were done with DiffKemp in the past. The goal is to analyze the projects, methods, and metrics that were used for the evaluation to be able to choose the evaluations/experiments that are generally useful and should be automated.

### 2.3.1 Existing Works on DiffKemp

The source of information about the previous evaluations are **past theses**, which can be divided into the following groups:

- Theses which **improve comparison process (reduce false positives)**. Here belong theses which (1) added support for custom code change patterns [11, 30], (2) added new built-in refactorings patterns [31], (3) extended slicing [22], (4) added application of formal methods (SMT) [20], and (5) thesis which in detail describes the entire whole DiffKemp [15]. Some of the experiments generally show a number of found differences between certain projects' versions. More often, the experiments show how newly created extension influences the results of projects' versions comparison – how it reduces the number of non-equal symbols (false positives), or in some cases just that the extension does not have a negative impact on the results (provides the same results even with the extension) or that it does not have a negative impact on the comparison itself (that it does not slow down too much the comparison process if at all). In most cases, the projects themselves are sufficient for evaluation, but in the case of CCP extensions, it was necessary to first create patterns specific to the projects to be able to evaluate them. Evaluations done in these theses usually also involve adding/extending tests that verify the functionality of the implemented extensions.

- Thesis which adds **support for new types of projects**. Here belongs a thesis which added support for snapshot generation from `make`-based projects [9]. Experiments show results of comparison for projects that were not previously possible to be compared using DiffKemp (or at least it was not simple to do so). The thesis also contains caching enhancement, which is evaluated by comparing runtimes of comparison with and without the enhancement.

- **Other theses** involves (1) result visualization enhancement [23], (2) an extension that slices semantically equal parts of from the compared programs allowing further more formal analysis [13], and (3) extension for automatically generating code change

patterns from similar differences allowing to ignore differences similar to them [12]. These extensions are usually evaluated by creating tests specific for the extension.

For completeness, it is worth noting that there are also papers about DiffKemp [16, 14]. However, the experiments and evaluations presented in these papers are also covered in the previously mentioned thesis [15].

### 2.3.2 Evaluation Methods and Metrics

Now, we will look more closely at the individual methods of evaluations which were done in the theses and metrics that they considered:

1. Evaluations analyzing **results of comparison**: As already mentioned, these experiments focus on the comparison result for some project versions. More frequently, they compare the same pairs of versions without and with the extension, and they analyze how the result of the comparison changed. The metrics the evaluations consider are the total number of found differences, comparison runtime, and the number of selected symbols evaluated as equal, not equal, and unknown. Some evaluations consider a number of true negatives, false positives, true positives, and false negatives instead of the number of symbols evaluated as equal and not equal, but for this, it is usually necessary to manually re-evaluate the changes between the project versions. Depending on the extension, the evaluations sometimes consider only some metrics like the comparison time. The metrics are used to show that with the extension, DiffKemp provides better or at least the same results, that the comparison runtime is not much affected, or that it optimizes the runtime (e.g., case of slicing).

2. Evaluation of extension by **testing**. This includes adding new tests that test the correct behavior of the new extension, rerunning the original tests to verify that nothing was broken, and performing manual testing.

3. **Comparison with other tools** is used to evaluate how DiffKemp handles analysis with respect to other tools by comparing the results of DiffKemp for certain projects with the results of other tools (e.g., Clever and Rêve for EqBench dataset).

4. Evaluation of the **effectiveness of individual semantic preserving change patterns**. This includes analysis of how much individual SPCPs helped eliminate false positives for projects by comparing results of comparison with all SPCPs enabled and then with individual patterns disabled.

5. Cross-validation of sliced functions with the result of original functions is an evaluation that was done in [13] for verification of the extension. It first compares project versions using DiffKemp. Then, it uses the slicing extension, which eliminates from functions that were evaluated as semantically non-equal parts of code that are semantically equal. Afterward, it compares the sliced functions, showing that even the sliced functions are evaluated by DiffKemp as semantically non-equal.

6. Evaluation of SMT runtime is an experiment specific for [20]. In the experiment, the amount of time that SMT solving took was analyzed for cases when SMT solver successfully evaluated the code of blocks as semantically equal.

### 2.3.3 Evaluated Projects

Now we will look in more detail at individual projects which the theses used for the evaluations.

#### RHEL Kernel

The first and the most frequently used project is the Red Hat Enterprise Linux (RHEL) kernel. This project was originally the main target of DiffKemp. RHEL is a commercial operating system built from the Linux kernel and providing long-term support as well as the latest open source innovation with stability and support [26].

The project is used for regression testing of DiffKemp and in experiments that analyze the results of comparison. The experiments compare either sysctl parameters or functions from the KABI list between multiple RHEL versions. Kernel Application Binary Interaface (KABI) is a set of kernel symbols that are considered stable and safe for third-party drivers to use. The semantics of the symbols should remain stable for a particular RHEL major release (this is true for RHEL v7 and v8) [27]. This is the reason why DiffKemp was developed to verify that the symbols' stability is maintained.

#### Upstream Linux Kernel

Another project used by the the experiments is the upstream repository of the Linux kernel[2]. The project is used for analyzing refactoring and optimization commits, which are found by searching for keywords in the repository commit messages. The semantics of functions changed by the commit are compared for the individual commits. The experiments analyze the correctness of the result provided by DiffKemp or how the extensions influence it. It is important to note that even if the commits are marked as refactoring, they sometimes contain semantic differences caused, e.g., by added safety checks [16].

#### Standard C Libraries

Other evaluations use releases of musl libc[3] to check the preservation of the semantics of the library functions. musl is a lightweight implementation of the C standard library. The experiments analyze either the results alone or their correctness.

In [20], there is also an experiment that compares semantics between two implementations of the standard C library. It compares and analyzes the implementations of functions in musl libc with diet libc[4].

#### Cryptographic Libraries

In [31], cryptographic libraries were analyzed using DiffKemp, specifically Nettle[5], Sodium[6], Mbed TLS[7] and wolfSSL[8]. The thesis compares for multiple library releases their API functions, which should perform mathematically well-defined operations (e.g.,

[2] Linux kernel upstream repository: https://github.com/torvalds/linux
[3] musl libc: https://musl.libc.org/
[4] diet libc: https://www.fefe.de/dietlibc/
[5] Nettle: https://www.lysator.liu.se/~nisse/nettle/
[6] Sodium: https://doc.libsodium.org/
[7] Mbed TLS: https://www.trustedfirmware.org/projects/mbed-tls/
[8] wolfSSL: https://wolfssl.com/

hashing, signing, or encryption) and whose semantics should not change between releases. The thesis evaluated the correctness of comparison results provided by DiffKemp and analyzed the effectiveness of individual SPCPs.

In [9], an experiment was done with another cryptographic library – OpenSSL[9]. The experiment analyzed the results of the comparison.

### EqBench Dataset

The last used project is the EqBench dataset[10] that was introduced in [2]. The dataset is divided into benchmarks where some were taken from different projects (Clever and Rêve), and the rest were created by the authors themselves. Each benchmark contains multiple programs. Almost every program has two variants – each variant contains two versions of a program (old and new) and a metadata file describing changes made to the program. The programs are written in Java and C language, the latter makes it possible to use DiffKemp for its comparison. The previously mentioned two variants of programs are a variant containing semantically equivalent changes and a variant with non-equivalent changes. The dataset totally contains 147 equivalent pairs of programs and 125 non-equivalent, which is, according to the author, the largest dataset for equivalence checking analysis. The equal variants of programs contain function-level code refactorings to represent realistic software evolution scenarios, and the non-equal variants contain random, non-equivalent changes. The dataset contains complex language constructs (e.g., non-linear arithmetic), making it challenging even for the most formal equivalence-checking techniques.

Experiments done using this dataset involve comparison of the versions for each program variant and evaluation of the correctness of results provided by DiffKemp, which is simple thanks to the fact that the variants are labeled if they contain equal or non-equal changes. It is also possible to compare DiffKemp with tools like Rêve and Clever, because the dataset contains benchmarks created by these tools and on which they were evaluated.

[9] OpenSSL: https://www.openssl.org/
[10] EqBench dataset: https://github.com/shrbadihi/eqbench

# Chapter 3

# Current State of Development Automation in DiffKemp

Software development automation tries to streamline repetitive tasks in a development process that would otherwise be performed manually. Automation speeds up the process by using specialized tools and methodologies. The automation improves team collaboration, saves time and resources, and boosts developers' productivity by allowing them to concentrate on higher-priority / more complex tasks. Automation includes code sharing and versioning, automated testing, code quality checks, documentation generation, monitoring, deployment, and automatic code generation from natural language description. In the future, DiffKemp would like to be part of the automation of software libraries' development, helping them to check the semantic stability of the symbols they provide.

One goal of this thesis is to improve DiffKemp's development automation by simplifying the assessment of how new features affect the tool's functionality and accuracy. For this reason, Section 3.1 briefly describes development automation in general – specifically the CI/CD pipeline and the available technologies. Next, Section 3.2 focuses on DiffKemp development process and the automation that it uses. Section 3.3 describes existing scripts created by DiffKemp's developers to automate evaluations of projects. Lastly, containers, which are sometimes used in development and its automation, are described in Section 3.4.

## 3.1 Development Automation

One of the key automation practices in modern development is *continuos integration* (CI), *delivery* and *deployment* (CD). Continuous integration ensures that software is in a deployable state, that it is compilable, and that the code has a reasonably good quality. It also tries to validate the software by automatically running tests, guaranteeing that it works as expected even after changes have been made to the code. Continuous delivery automates the creation of software releases and continuous deployment automatically deploys changes to production, making it possible to catch bugs faster and immediately fix them [28].

CI needs a shared central code repository in the form of **version control system** (VCS). This ensures that the code is kept in one place, which makes it possible for developers to collaborate on the same project, check out the source, make changes, and commit those changes [28]. VCS allows reverting files back to a previous state, comparing changes over

time, seeing who last modified something that might be causing a problem, recovering lost files, and more [29]. Popular VCSs include Mercurial[1], Subversion[2] and Git[3].

DiffKemp uses **Git** as its VCS. Git is a *distributed* source control system, meaning that each developer has a fully mirrored repository. It allows features to be developed independently in parallel in different branches. There are multiple hosting platforms for Git, among popular belongs GitHub[4], GitLab[5] and Bitbucket[6]. DiffKemp uses GitHub to host its repository. In addition to VCS, GitHub also allows code reviewing, issues tracking and more.

The main part of continuous integration is the **CI server**, which should, among others, build the software, run tests (unit tests, integration tests, ...), check its quality (calculate code coverage, check style guidelines, ...), and provide feedback to the developer if a check fails [28]. Typically, a CI server runs the configured checks whenever a change is committed to the central repository. CI practice involves frequently committing/integrating small changes to the repository, which helps to avoid build failures, detects errors sooner, and makes it easier to merge changes from different developers.

Now, we will look more closely at multiple available automation technologies. This will be important in Section 4.2.1, where we will select the technology for creating an extension that will automate the assessment of the impact of new DiffKemp features.

### GitHub Actions [6]

DiffKemp uses GitHub Actions, which allows developers to create *workflows* (automated processes) which run when a specific event occurs in the repository. The workflows are described by YAML files which are committed to the repository.

The GitHub Actions server is called a *runner*. When a workflow run is triggered, a new runner is used. Workflow can run one or more jobs, each job is a set of steps executed on the same runner. A runner is always an isolated virtual machine that runs a specified operating system. The runners are destroyed at the end of the workflow process [6].

GitHub provides a free plan **standard runner**. This is what DiffKemp uses. The runners have a limited storage size to 14 GB. The plan also allows to reuse the created data in subsequent workflow runs by caching them from the runner. The data is automatically removed if it has not been accessed in over 7 days. The same data can potentially be cached multiple times because CI workflows are restricted – the workflows can recover only caches created in either the current branch or the default (*master*) branch. The total size of all cached files in a repository is limited to 10 GB.

In addition to standard runners, GitHub also provides **large runners**. These provide more resources – RAM, CPU, and disk space (14–2040 GB). They also provide additional features (e.g., a static IP). To be able to use the large runners, users must have the GitHub Team or GitHub Enterprise Cloud plan, which are monthly paid per user.

GitHub also allows using **self-hosted runners**. This means that you use your own machine as the continuous integration server. The machine connects to GitHub using the runner application. It is not much recommended for use with public repositories (which DiffKemp is), because pull requests can potentially run dangerous code on the runner

[1] https://www.mercurial-scm.org/
[2] https://subversion.apache.org/
[3] https://git-scm.com/
[4] https://github.com/
[5] https://about.gitlab.com/
[6] https://bitbucket.org/

machine. The default usage of the self-hosted runner is as an *always-on persistent runner* [19]. This means that the runner processes one job at a time and remains active after the processing. Another option is to dynamically auto-scale up and down the runners based on usage, but this option requires a Kubernetes cluster [19].

Outside of GitHub Actions, GitHub also provides other ways to automate things by interacting (open issues, comment on pull requests, ...) with the repository – specifically custom actions, OAuth apps, and GitHub apps. **Custom actions** [6] are „just“ blocks of code that can be reused by multiple GitHub Actions workflows and that can use GitHub's APIs. GitHub and OAuth apps are described in the following paragraphs.

**GitHub App** [7] is an integration that runs on the owner's server. It runs persistently and listens and reacts to events happening on GitHub. It allows us to use *webhooks*, which, unlike *polling*, does not have to periodically call an API to check the data to recognize if an event occurred (e.g., a new pull request was created) [8]. Webhooks allow us to subscribe to important events and if an event occurs, GitHub will automatically deliver relevant data as an HTTP request to our server. The app provides permission settings, allowing granting access to repositories and specifying what data it can read or modify.

To react to events (modify data), the GitHub app can use **GitHub APIs** – a REST API[7] or a GraphQL API[8]. The modifications can be either attributed to the app or a user.

It is possible to use apps from the GitHub marketplace or to create custom apps. The process of creating your own app includes:

1. **App registration:** This consists of (a) an app name, (b) a selection of permission which data the app can access and modify, (c) a selection of events to be delivered, and (d) a specification of the URL to deliver the event to.

2. **Server creation:** The server should contain HTTP endpoints for receiving the webhook events. When a HTTP payload with data about an event is recieved, it should process it and react on it by using GitHub API. To be able to call the API, the application first needs to authenticate by generating an access token. The authentication process is relatively complicated process and note that tokens expire in 1 hour.

3. **App installation:** The last step is to install the registered app to a repository. After that, the app server can interact with the repository and events occurring in the repository will be sent to the server.

**OAuth App** [7] is similar to GitHub App, but GitHub App is preferred because, among others, it uses fine-grained permissions and gives more control over which repositories the app can access.

### Other CI platforms

Because DiffKemp uses GitHub, we mainly focussed on the GitHub integrations, but we will look shortly at different continuous integration platforms that could be used for the extension. The three top platforms are Jenkins, Travis CI, and Circle CI[9]. We will focus on their free plan because it is what DiffKemp uses.

[7] https://docs.github.com/en/rest
[8] https://docs.github.com/en/graphql
[9] This is based on article at https://www.browserstack.com/guide/continuous-integration-tools.

### Travis CI[10]

Similarly to GitHub Actions, Travis CI is a cloud-based CI platform. Travis CI free plan comes with a limited number of building minutes which are not replenished and are invalided after a 14-day trial.

### Circle CI[11]

Cloud-based platform running on credits. The free plan contains 30,000 credits per month, making it approximately 6,000 minutes if minimal resources (RAM, storage, and CPU cores) are used. The cache size is unlimited but uploads over 2 GB are charged 420 credits per GB. The cache storage duration is 15 days by default.

### Jenkins[12]

The leading open-source CI server. It is not cloud-based and needs to be deployed on its own server. It provides a number of plugins for integrating into GitHub.

## 3.2 DiffKemp Development Process

As already mentioned, DiffKemp uses Git as the version control system and the repository is hosted on GitHub. For developing/contributing, DiffKemp uses *fork and pull request* workflow [5]. This means the developer is supposed to create their own copy of the original repository (*fork*) on GitHub. Then, the developer should clone the fork to his computer and create a new branch for the changes he wants to make. Changes should be committed frequently in the branch. Once the feature is developed and ready to be integrated into the original repository, the branch should be pushed up to the fork. The changes should then be proposed by creating a *pull request* to the original repository.

### Pull Requests

When a pull request is created, GitHub Actions (CI) is triggered and information about the passed and/or the failed checks is provided to the user. Other DiffKemp developers review the pull request, providing feedback and suggestions for improvement to the pull request creator. The code is iteratively improved by the feature developer based on the feedback, suggestions, and CI results until all the CI checks pass and the feature is approved by a different DiffKemp developer. Approval can only be given by a developer (collaborator) with write permissions to the repository.

If the PR is approved and all the CI checks pass, the changes from the pull request can be integrated into the DiffKemp *master* branch. DiffKemp uses *rebase and merge* strategy [4, 29] for integrating the changes. This means that the changes (individual commits) in the pull request branch are individually reapplied on the *master* branch without creating a merge commit. After that, the branch is fast-forward merged to the *master* branch. The rebasing makes DiffKemp history cleaner looking like all the work happened in series, even if it happened in parallel. GitHub handles the rebase automatically, but in cases where there are merge conflicts between the *master* branch and the branch with

[10] https://docs.travis-ci.com/
[11] https://circleci.com/docs
[12] https://www.jenkins.io/doc/book/

the extension, which are not possible to be automatically resolved, the extension developer must first manually rebase the branch on the *master* branch solving the collisions. The collisions can be complex, and mistakes can be made by the developer; therefore, the pull request is integrated after all CI checks pass again, which should ensure that the collisions were successfully solved.

### Development Environment

For development, DiffKemp provides Nix flakes [21]. Nix is a package manager that simplifies sharing of reproducible development and building environments. Flakes are the unit for packaging. They specify dependencies of the package/project locked to specific revision, process how to build the project, and other things. Nix flakes allow DiffKemp to be built automatically with a single command, eliminating the need to manually install dependencies. It also enables start of a bash shell with an environment containing the project dependencies. This is useful for testing when developing. Nix flakes also allow building and running a bash shell with variants of the project. DiffKemp supports multiple Clang (LLVM) versions (9-17), and for each version exists a Nix flake variant, allowing to use/test DiffKemp on that specific Clang version.

### Continuos Integration

As already mentioned DiffKemp uses GitHub Actions as a platform for continuous integration (CI). It uses the free-plan standard runners. CI workflows are run (triggered) when a pull request is opened but also on every push of changes to a branch in DiffKemp repository. The workflows (1) check code style, (2) build DiffKemp, (3) check that the build is successful, and (4) test the project on multiple LLVM versions. For building, the previously mentioned Nix flakes are used.

The CI uses cache for caching dependencies (e.g., Nix) and testing data. Totally around 2.5 GB of data is saved in the cache. If a cache hit occurs, the CI runtime currently takes approximately 11 minutes, otherwise it takes around 30 minutes. An analysis of CI runs on the main repository[13] shows that 480 CI runs were triggered from June 2023 to September 2024, ranging from immediate (caused, for example, by pushing multiple branches at the same time) to almost a month between the execution of subsequent runs. The average time between runs is around a day.

### Tests

DiffKemp contains multiple tests, all of them being run by the CI. The tests can be divided into the following groups:

- Unit tests for the core part of DiffKemp – the SimpLL library. These tests cover (1) analysis and transformation passes, (2) correctness of instruction-by-instruction comparison, (3) correct application of SPCPs, and (4) correct usage of CCPs. The correctness is evaluated on small functions created in C++ using the LLVM API or directly written as string literals in LLVM IR.
- Unit tests for creating the snapshots.

[13] Done using `gh run list -a --json createdAt -L 5000 -w CI -R diffkemp/diffkemp |\ jq -r '.[].createdAt' | sed lis.tmp -e 's/T/ /' -e 's/Z//'`.

- Regression tests which create snapshots of certain RHEL versions and compare a few selected functions and sysctl parameters and check if the current results are the same as those that have been specified by YAML files committed to the repository. This should verify that DiffKemp still performs as expected even after making changes. The tests also use manually created CCPs to verify that matched changes are still filtered out. Since creating snapshots is time-consuming, the files necessary for testing are cached. This caching speeds up the CI testing runtime.
- Unit tests for the result viewer, which test the correct rendering and event handling of individual components.

## 3.3 DiffKemp Scripts

For DiffKemp, additional scripts have been created in the past for the purposes of simplification and automation of certain evaluations. Of these scripts, three are publicly available and worth mentioning.

### Script for Comparing Multiple Versions of a Project[14]

The script was created as part of [31] to evaluate cryptographic libraries. It allows us to build and subsequently compare multiple versions of a provided C project. The tool requires a YAML configuration file containing:

- the URL of the project repository,
- the commands for configuring the project,
- options required by DiffKemp for building the project versions into snapshots,
- the list of functions to compare, and
- the list of versions (git tags) that should be compared by DiffKemp.

The tool clones the project repository, builds the specified versions into snapshots, and compares the specified functions in consecutive pairs of snapshots. The tool's output is saved to a file containing a report for each pair of compared versions. Each pair contains the result of the comparison for all specified functions. The result can be labeled as:

- **semantic**: DiffKemp evaluated the function as semantically non-equal.
- **syntactic**: The function has a syntactic difference, but it was evaluated as equal (semantic preserving).
- **without a difference**: A function without syntactic difference, evaluated as equal.
- **unknown**: DiffKemp did not manage to evaluate the function.

[14] Script is available from https://github.com/zacikpa/diffkemp-analysis.

### Script for Automatically Evaluating Programs from the EqBench Dataset[15]

The script was created by me and was used for experiments in [20]. It uses DiffKemp to build each program version into a snapshot and compare them. The script creates a file reporting for each program (1) its name, (2) the expected result (whether changes made to the program are semantically preserving or not), and (3) the result equal/not-equal provided by DiffKemp. The script also outputs a summary of the evaluation, providing the number of true positives, false negatives, true negatives, and false positives.

Addionally, the repository also provides a script for running evaluations with multiple different DiffKemp options. Particularly with enabled/disabled usage of SPCPs, and most importantly, with used optimization options. The latter has an impact on how the source files are compiled into LLVM IR – which transformation passes are used during snapshot generation on the compiled LLVM modules. This can potentially bring the LLVM IR of the analyzed function versions syntactically closer to each other. The script runs evaluation using `-O1`, `-O2` optimization passes as well as custom/default passes selected by DiffKemp developers.

The script shows that DiffKemp gives the best results (least false positives) with the `-O2` optimization option. Further experiments were carried out using different versions (9-17) of the Clang compiler (used for the compilation of the sources to the LLVM IR), which showed that the custom/default passes provide more stable results across different Clang versions than `-O1` and `-O2`. Table 3.1 summarizes these results.

Table 3.1: A comparison of DiffKemp on an EqBench dataset with different optimization options showing a range of false positives across multiple (9-17) Clang versions.

| optimization options | range of false positives |
|---|---|
| `-O1` | 51–67 |
| `-O2` | 49–55 |
| custom/default | 88–91 |

### Script for Checking Semantic Equality of Several Commits[16]

The last script was created to evaluate refactoring and optimization commits in the upstream Linux kernel. The script is general and can also be used to analyze commits in different kernel repositories.

The script takes a list of commits to analyze and a path to the project repository. For each commit, the script:

1. Identifies the functions changed by the commit by analyzing the commit diff.
2. Creates snapshots of the project version before and after each commit.
3. Compares the created snapshots, analyzing the identified changed functions for each commit.

The script saves results to a file, with each line reporting about one commit. The report contains:

- a hash of the commit,

[15] Repository with the script available in https://github.com/diffkemp/eqbench-workflow.
[16] Script is available from https://github.com/FrNecas/commit-analysis/.

- the names of identified functions,
- the amount of functions evaluated as equal/non-equal/unknown and where error occurred, and
- a verdict. The verdict is either equal or not-equal. It is not-equal if an error occurred or any of the functions was evaluated as semantically not-equal, otherwise, the verdict is equal, meaning that the commit does not change the semantics of the project.

The report also includes a tag indicating whether all changed functions for a commit were successfully localized. The function localization process is not perfect and may require manual corrections.

## 3.4 Container Technology (Containerization)

Container technology is later used by the created solutions. Containerization allows to create *container* that bundles code, configuration files, libraries, and dependencies. In this, it is similar to the Nix package manager mentioned earlier. However, containers are isolated, reducing the chance that malicious code running in one container will impact other containers or even the host system [10].

# Chapter 4

# Automation for Evaluation of the Impact of New DiffKemp Features

The first goal of this thesis is to extend DiffKemp's development automation by creating a new component which would simplify the assessment of the impact of new DiffKemp features. This chapter focuses on the extension requirements, which are specified in Section 4.1. After that, Section 4.2 deals with the extension design. Lastly, Section 4.3 describes the implementation.

## 4.1 Specification and Analysis of Requirements

Firstly, we need to determine the requirements of the extension by analyzing the following:

- What kind of features are being added to DiffKemp?
- What method should be used to simplify the assessment of the impact of new features?
- What projects (if any) should be used for the determination of the impact?
- How should the extension be used?

### 4.1.1 Analysis of Existing Features (Pull Requests)

To figure out the requirements of the extension, it should be useful to look at the previous features that were already added to DiffKemp. Some of the features, particularly the more complex ones, were already mentioned in Section 2.3.1. Naturally, this list is not complete and for this reason, we will look at DiffKemp's pull requests (PRs).

At the time of the analysis, there were 283 PRs, of which:

- 261 PRs were closed and merged to the *master* branch,
- 11 PRs were closed but were not merged (e.g., because their development became stale or changes have contained too many collisions with the *master* branch),
- 8 PRs were opened and prepared for review or merging but were not yet merged, and
- 3 PRs were opened as draft (meaning being developed and not yet ready for review).

Table 4.1: Categorization of DiffKemp pull requests to groups based on the types of changes.

| Category | Amount | Types of PRs |
|---|---|---|
| Snapshot generation changes | 20 | Changes of options used for compiling source files to LLVM modules and its linking, modifications of the executed optimization passes, bug fixes, refactorings, ... |
| Snapshot comparison changes | 100 | Support of new LLVM versions, comparison enhancements (e.g., new transformation passes, new built-in patterns [31]), bug fixes, refactorizations, optimizations, ... |
| Minor changes | 17 | Changes in the localization of the code causing a semantic difference, simpler refactorizations, caching introduction, optimizations, ... |
| Snapshot comparison changes not visible by default | 12 | Extensions requiring usage of options to be enabled (comparison of control flow, usage of SMT [20]), requiring additional inputs (CCPs extensions [30, 11, 12]) or more complex usage (equivalence slicing [13]), ... |
| Other | 134 | Collection of statistics, creation of releases, tests, helper scripts, command options, new snapshot generators [9], the result viewer [23], changes in documentation, in the continuous integration, output formats, development environment, logging, ... |

We analyzed the pull requests and divided them into multiple categories based on the changes they contained. Changes were deduced from the PR title, description, and changes made to the files. The categories are shown in Table 4.1. For individual categories, the name of the category, the types of PRs ranked to the category, and the amount of PRs belonging to it are presented.

Categories **snapshot generation changes** and **snapshot comparison changes** are considered to have the largest impact on the comparison results. Most PRs in these categories should positively influence the results by eliminating false negatives and positives. The categories also contain PRs with a high risk that they could have negatively affected the results. This includes PRs adding support for new LLVM versions and some larger refactoring and optimization changes.

Adding **support for new LLVM versions** is something that is done in DiffKemp regularly. This is because operating systems usually come with the latest version of LLVM (Clang). It is desirable that DiffKemp would be usable on these systems. New LLVM versions usually change some parts of the LLVM API (replace methods with different ones) and sometimes make changes in the LLVM instructions (e.g., a transition from explicit pointers to opaque pointers done in LLVM 15[1]), or introduce new instructions. To make possible use of the new versions, changes must be made to DiffKemp's SimpLL library

[1] https://llvm.org/docs/OpaquePointers.html

that uses the LLVM API. DiffKemp also supports older LLVM versions, and it seems desirable that the tool should try to provide the same results across all supported versions.

PRs in the category of **minor changes** could be added to the previously mentioned groups. Unlike them, they should not have a big impact on the comparison results (semantic equality), but they may rather affect the runtime of the analysis either positively (optimization, caching) or negatively (more precise code differences localization).

The next category of **changes in snapshot comparison not visible by default** contains PRs that usually add new functionality to snapshot comparison which is not enabled/used by default. Here belong, for example:

- the CCPs patterns that are necessary to be manually created based on the analyzed project and provided to the comparator or
- the SMT extension that needs to be manually enabled by passing an additional option to the snapshot comparator command.

The last category (**other**) includes almost half (47 %) of all PRs. The PRs usually change things related to DiffKemp development (e.g., CI and documentation) or introduce totally new things (e.g., the result viewer).

### 4.1.2 Selection of Method and Metrics

To simplify the assessment of the impact of new DiffKemp features, a suitable method must be chosen. For the selection of the method, we consider the evaluation methods used in the past and described in Section 2.3.2. From the mentioned methods, the only reasonable ones are testing and **analysis of comparison results** on open-source projects. It does not make sense to create a testing extension because tests are already being run as part of the CI, and a feature developer should create and add new tests for the new feature. The analysis method would be useful for the evaluation of approximately 48 % of the total existing PRs, specifically those belonging to the first three categories in Table 4.1. The method should ensure that changes like refactoring do not negatively affect performance (e.g., slower runtime, more incorrect results), and on the other hand, new enhancements contribute to improved comparison results.

Using the method, our extension should first compare certain projects' versions using DiffKemp located on the *master* branch, gaining *base comparison results*. Then, it should compare the same versions of the projects using DiffKemp containing the proposed changes, gaining *feature comparison results*.

The created extension should provide a comparison of the base results with the feature results, showing how the feature improved or worsened the results. The results should contain metrics described in Section 2.3.2 – (1) the total number of found differences, (2) the comparison runtime, and (3) the number of functions evaluated as equal, not equal, unknown, and number of symbols where an error occurred during the comparison. Instead of equal and non-equal numbers, numbers indicating true positives, false negatives, true negatives, and false positives could be provided. The latter-mentioned numbers would provide more useful information on the impact of the feature. The reason is that an increase in symbols evaluated as equal can indicate a decrease in false positives, but in reality, the increase can mean occurrences of false negative cases.

### 4.1.3 Selection of Projects

Now, the important thing to decide is which projects and versions to use for the comparison. For the selection of the projects, we consider the projects used in the past and mentioned in Section 2.3.3. The projects are:

- **RHEL Kernel:** As the RHEL kernel is the main target of DiffKemp, it is logical to evaluate new PRs on this project. To make the evaluation useful for a broader range of new features, the KABI functions as well as the sysctl parameters should be compared. Both comparisons are necessary because certain features might enhance only one of these comparison.

  The extension should compare minor versions of the RHEL kernel version 8, specifically pairs of versions between RHEL version 8.0 and 8.5. These versions were selected because DiffKemp has been executed on them the most, and the results were reviewed numerous times. The reason for comparing multiple versions is to cover more cases of code changes. The extension could be extended in the future with the comparison of more versions, but first, the results of DiffKemp on those versions should be manually reviewed.

- **EqBench dataset:** A very useful project as it should be the largest dataset for the analysis of equivalence checking [2]. Moreover, the programs that the dataset contains are already labeled as semantically equal or non-equal, making it easy to decide if a new feature increases or decreases the number of correctly evaluated programs. This makes the project a good candidate for our extension.

  As already mentioned, the programs in the dataset can be compared by running DiffKemp with different optimization options (e.g., DiffKemp's default, `-O1` and `-O2`). This raises the question of which option should we use for our extension. The best one seems to be the DiffKemp's default optimization option for multiple reasons:

  - As the results of the experiment captured in Table 3.1 in Section 3.3 showed, it provides the most stable results across multiple versions of Clang/LLVM. This seems to be an important property for the evaluation of PRs that introduces support for new LLVM versions.
  - The other options use function inlining which helps in getting better results but in cost of precise localization of the difference in the code, which can be very useful in real life usage. Furthermore, currently, this is the only option available for generating snapshots of the Linux kernel.

  It could also be useful to provide the best possible results for the project in addition to the one gained by using the default options. Currently, the best results provide both `-O2` and `-O1` optimization options. In the past `-O1` provided slightly worse results, so `-O2` seems to be a better candidate.

- **Upstream Linux kernel**, **standard C libraries**, and **cryptographic libraries:** These projects could be also used but the EqBench dataset and RHEL kernel should already provide good base for impact evaluation.

### 4.1.4 Design of the Extension Usage

Having selected the method and the projects for the evaluation, the next critical question is when the feature assessment should be performed. As described in Section 3.2, when a new feature is developed, it is being proposed by creating a pull request, then it is reviewed and merged into the *master* branch. It is logical to assess the impact of the feature during the review process before merging it into the *master* branch.

Looking again at the existing PRs described in Section 4.1.1 and grouped into categories in Table 4.1, we can obtain additional requirements and information for usage:

- To evaluate the impact of PRs that introduce support for new LLVM versions, the solution should **utilize the latest supported LLVM version** for building and comparing snapshots.
- To allow assessment of PRs belonging to the category *snapshot comparison changes not visible by default*, specifically those which are not enabled by default (e.g., the SMT extension), the solution should provide a way to **specify additional options** that will be passed down to the snapshot comparator. Concerning other PRs in this category, there are also extensions using the custom change patterns. Our extension could use some patterns for analysis of the RHEL kernel, but there do not exist many PRs that change the patterns. Furthermore, the functionality of the patterns is already being checked by existing regression tests. If there was a new feature regarding the patterns, it would most likely extend the patterns' syntax. This means that even if we added usage of some patterns to our solution it would probably not provide meaningful insights into the impact of such extension.
- It does not make sense to run the extension for every PR, but rather **on demand** of the reviewer. This approach is justified because the solution would not be useful in assessing approximately 47 % of existing PRs (category *other*).

Now, we will look closer at the selected projects, specifically on the time needed for the semantic comparison of these projects. The runtimes are shown in Table 4.2 and contain the time taken by snapshot generation and comparison. The times can differ depending on the used hardware. For EqBench, the table contains the time needed for analysis using a single optimization option. For the RHEL kernel, the time necessary for creating a snapshot of one version and comparison of two snapshots is stated. The times vary slightly across RHEL versions – the times specified in the table represent averages for RHEL 8.0–8.5. The times are also different based on whether KABI symbols are compared or sysctl parameter.

Table 4.2: Approximate runtime of DiffKemp's analysis on the selected projects – the EqBench dataset and the RHEL kernel.

| **Project** | **Snapshot generation runtime** | **Snapshot comparison runtime** |
|---|---:|---:|
| EqBench dataset | 2.2 min | 1.0 min |
| RHEL kernel (KABI symbols) | 36.8 min | 12.5 min |
| RHEL kernel (sysctl parameters) | 19.8 min | 6.0 min |

As visible in Table 4.1 in Section 4.1.1, only 20 PRs of 283 were estimated to change snapshot generation. By further analysis, where we search for PRs changing files containg source code of the snapshot generator, additional 14 PRs were discovered. This means that totally 12 % of all PRs changed the snapshot generation. Because this number is relatively low and the snapshot generation for the RHEL kernel takes some time (e.g., sequentially building four kernels would take around two hours), it would make sense to **reuse existing snapshots** by default and just compare them to speed up the process. The solution should also allow us to specify that the snapshots should be rebuilt if necessary.

Table 4.3: Summary of Requirements

| ID | Description |
|---|---|
| `CMP_RES` | The created extension should provide a comparison of the base results with the feature results to assess the impact of a new feature. |
| `CMP_RHEL` `CMP_EQBENCH` | The results should be obtained by semantic comparison of the EqBench dataset and the RHEL kernel. |
| `CMP_TIME` | The results should contain a comparison runtime. |
| `CMP_STATS_1` | The results should also contain the total number of found differences and the number of functions evaluated as equal, not equal and unknown. |
| `CMP_STATS_2` | Alternatively to `CMP_STATS_1`, the results may contain numbers indicating true positives, false negatives, true negatives, and false positives. |
| `CMP_BASE` | The base results should be gained by comparing the projects with DiffKemp located on the *master* branch. |
| `CMP_FEATURE` | The feature results should be gained by comparing the projects with DiffKemp located on the feature (PR) branch. |
| `EQBENCH_OPT` | The EqBench dataset should be compared by using the default and the `-O2` optimization options. |
| `RHEL_KABI` `RHEL_SYSCTL` | For the RHEL kernel, symbols from the KABI list as well as sysctl parameters should be compared. |
| `RHEL_VERSIONS` | The RHEL kernel comparison should involve multiple minor versions of RHEL v8 (8.0 – 8.5). |
| `RUN_OPEN_PR` | The impact assessment should be done on opened pull requests before its merger. |
| `LLVM_LATEST` | The projects analysis should be done using the latest LLVM version supported by DiffKemp. |
| `CMP_OPTIONS` | The solution should provide a way to specify additional options passed down to the snapshot comparator. |
| `RUN_REQUEST` | The assessment should be performed only at the request of the reviewer, rather than automatically for every pull request. |
| `SNAPSHOTS_REUSE` | By default, existing (pre-built) snapshots of the projects should be used to speed up the analysis. |
| `SNAPSHOTS_REBUILD` | The solution should provide a way of specifying to rebuild the snapshots if necessary. |

### 4.1.5 Summarization of Requirements

The requirements mentioned in this section are summarized in Table 4.3.

## 4.2 Design of the Extension

This section focuses on the design of the extension which should simplify the impact assessment of new features. In Section 4.2.1, the main technology for the extension will be selected. Next, Section 4.2.2 concentrates on the design of the extension launch mechanism. Section 4.2.3 designs the visualization of results for the impact assessment. The design of the overall architecture and the details of usage will be proposed in Section 4.2.4. Lastly, in Section 4.2.5, we will look on the architecture in more detail.

### 4.2.1 Selection of the Technology

As described in Section 3.2, DiffKemp uses **Github Actions** as its CI, so it would be logical to use this technology to implement the new extension. By exploring the storage size needed by the selected projects and by further analysis of the technology, it seems that this technology would not be the best choice.

Table 4.4: Storage size taken by the selected projects.

| **Analyzed project** | **Source size** | **Source size after snapshot generation** | **Snapshot size** |
|---|---|---|---|
| EqBench | 11 MB | 71 MB | 69 MB |
| RHEL kernel (KABI symbols) | 1009 MB | 3.2 GB | 646 MB |
| RHEL kernel (sysctl parameters) | 1009 MB | 2.9 GB | 423 MB |

Table 4.4 shows the storage size needed by the selected projects. *Source size* represents the size of the project. *Source size after snapshot generation* represents the size of the project directory after a snapshot is generated. The project size increases because the source files compiled to the LLVM modules are first saved to the project directory before they are copied to the snapshot. In the case of the RHEL kernel, a symbol cross-reference database is also created which DiffKemp uses to find a source file containing the definition or usage of an analyzed symbol. The database takes up some space. The last column, *snapshot size*, contains the snapshot directory size. Similarly to Table 4.2, the RHEL kernel sizes are presented for a single version and are obtained as an average between versions 8.0 to 8.5.

The problems with the **GitHub Actions standard runners** (which DiffKemp currently uses) concerning the extension that we want to create are:

1. **Limited storage size (14 GB):** Just looking at the average size of the RHEL kernel directory after snapshot generation for KABI symbols, we can see that we would be able to maximally compare around 4 versions before reaching the limit. (Note that the limit can be partially bypassed by the removal of some unused software from the server.) If we later decided to add additional projects to be used for the assessment, we would reach a big problem with the limit, and we would need to compare fewer

RHEL versions or use different technology. Moreover, `DiffKemp` has already problems with this limit in the past[2].

2. **Limited cache size (10 GB):** As decided in the requirement 14 (Section 4.1.5 Table 4.3), we do not want to rebuild the project snapshots every time to speed up the process. We would need to *cache* the created snapshots (and potentially also the project sources) for this. So, similarly to the previous point, this would be a problem, we could not cache many snapshots/projects because of the limit. Furthermore, with this limit, DiffKemp also had problems in the past[3].

   There is a solution[4] that allows to use own cache server which should theoretically solve this problem, but it would not solve the previous one.

3. **Cache clearing:** The last discovered problem is that the caches are cleared if they have not been accessed in 7 days [6]. As mentioned in Section 3.2, the average time between CI runs is around a day, but sometimes it exceeds a week, reaching up to a month. This means that in case of the longer period between CI runs, the cache would be cleared and snapshots would need to be created again slowing down the analysis process.

Now we will discuss other technologies mentioned in Section 3.1:

- **Large GitHub Actions runners:** Paid plan is needed to be able to use them, because DiffKemp's development is not very regular and takes place mainly as work on theses, it probably does not make sense to pay for the plan instead of using the current free plan. Moreover, there would still be a problem related to the cache limit.

- **Self-hosted runner for GitHub Actions:** This would solve the storage size limit given by the standard runner by using our hardware with as much memory as necessary. The problem is that it still uses by default, the limited caches which are hosted on GitHub. There are probably ways to overcome this – for example, by misusing that the persistent runner keeps running even after workflow processing is finished or using some custom caching solutions[5].

- **GitHub App:** Technology that enables us to interact with the GitHub repository from our own server. DiffKemp has in its disposal server that could be used, so this is a viable possibility. It would allow us to create a custom solution without limits (disk storage, cache size, ...) given by other platforms. Also, the extension that we are creating should not be too complicated for us to not be able to create our own „CI server" for it.

- **Travis CI:** Contains only a 14-day free trial plan. The paid plans do not make much sense, as already mentioned.

- **Circle CI:** This platform is better than GitHub Actions concerning caching. The problem is the limited number of credits/minutes per month.

[2] Pull request https://github.com/diffkemp/diffkemp/pull/265 was fixing an issue with lack of disk space on the CI server.

[3] https://github.com/diffkemp/diffkemp/issues/337

[4] https://github.com/falcondev-oss/github-actions-cache-server

[5] https://github.com/runs-on/cache

- **Jenkins:** Similar to the GitHub App, this is also a viable possibility as it is self-hosted CI.

Ultimately, the GitHub App was selected as the technology to implement the extension. However, using Jenkins would also be acceptable.

### 4.2.2 Design of the Launch Mechanism

Requirement `RUN_REQUEST` in Table 4.3 specifies that the evaluation should be manually initiated upon request from a PR reviewer. We evaluated two options how to do this:

1. using webhook *workflow dispatch*, which is triggerable in the browser by selecting the PR branch and clicking on a button on the page with GitHub Actions or

2. by writing a specific comment to the pull request.

The second option was selected, where the reviewer will not have to visit a different page.

A PR evaluation will start when a reviewer adds a comment to the PR containing a line beginning with `\evaluate`. The text represents a command that provides additional optional arguments, as described in Table 4.5. After the evaluation is completed, it should inform the reviewer about the impact of the PR.

Table 4.5: Arguments of the evaluation command.

| Argument | Description |
|---|---|
| `[{-h\|--help}]` | Instead of running an evaluation, information about the evaluator usage will be posted as a comment to the PR. |
| `[--run PROJECT]` | The option specifies one or multiple projects which should be used for the evaluation. The currently supported values are `eqbench` for the EqBench dataset, `rhel-functions` for evaluation of KABI symbols on RHEL kernels, and `rhel-sysctl` for evaluation of sysctl parameters. All projects are evaluated by default. |
| `[--pr-cmp-opt OPTIONS]` | The argument allows the specification of additional options that will be passed to the snapshot comparator when analyzing the new feature (using DiffKemp with PR changes). The argument is useful if the PR adds a new DiffKemp option not enabled by default. |
| `[--cmp-opt OPTIONS]` | Similarly to the previous one, the argument allows specifying additional options which will be passed to the snapshot comparator, but this time, it is for creating the feature results as well as the *base results*. It is useful when the PR should be evaluated with a certain, already existing, option not enabled by default. |
| `[--rebuild]` | The option forces rebuilding of snapshots instead of using the pre-built ones. It is helpful for PRs that would change the snapshot generation command. |

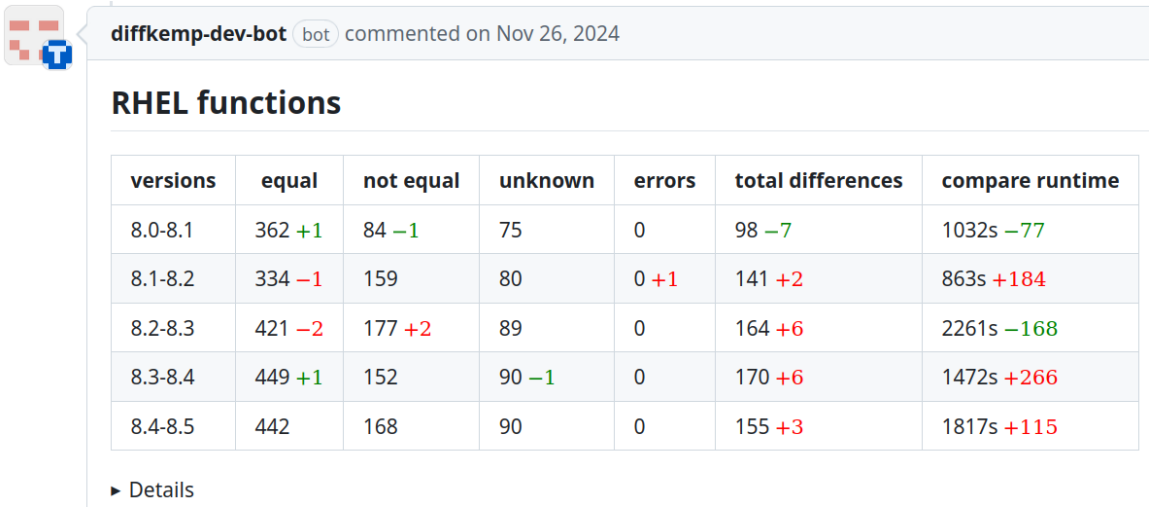
### 4.2.3 Design of the Visualization

This section will propose how the evaluation results should be visualized to the user. For this, multiple options are viable: (a) *a job summary*[6], (b) *a workflow artifact*[7], or (c) *a pull request comment*. The first two options are specific for GitHub Actions. This means we are left with the last option, **PR comments**, which is a pretty good solution because the results will be visible directly on the pull request page.

#### Pull Request Comments

For visualization of the evaluation results, we will use the fact that the comments support the **Markdown format**[8]. We will create a new comment for each project result to keep the comments short and clean. We will use a compact table to visualize the comparison between base results and feature results. The compactness will be performed by showing the base result numbers and showing how those numbers change in the feature results. The increase/decrease of the value will be colored to represent if it means a positive or negative impact of the PR on DiffKemp performance. GitHub does not support coloring in comments by default, but it supports Latex math expressions, which can be „misused“ for coloring text[9]. Generally, the positive impact will be colored green and the negative impact will be red.

The output will also provide more detailed information, specifically about programs and functions whose results changed. To keep comments short and clean, *collapsed sections* will be used which are blocks of text hidden by default but expandable by the user.



**diffkemp-dev-bot** bot commented on Nov 26, 2024

**RHEL functions**

| versions | equal | not equal | unknown | errors | total differences | compare runtime |
|---|---|---|---|---|---|---|
| 8.0-8.1 | 362 +1 | 84 −1 | 75 | 0 | 98 −7 | 1032s −77 |
| 8.1-8.2 | 334 −1 | 159 | 80 | 0 +1 | 141 +2 | 863s +184 |
| 8.2-8.3 | 421 −2 | 177 +2 | 89 | 0 | 164 +6 | 2261s −168 |
| 8.3-8.4 | 449 +1 | 152 | 90 −1 | 0 | 170 +6 | 1472s +266 |
| 8.4-8.5 | 442 | 168 | 90 | 0 | 155 +3 | 1817s +115 |

▸ Details

Figure 4.1: Design of evaluation results visualization for the RHEL kernel with compared functions from the KABI list.

Figure 4.1 contains the evaluation output proposal for comparison of the **KABI functions** in the RHEL kernel. A similar output is expected for the sysctl parameters. The

[6] https://github.blog/news-insights/product-news/supercharging-github-actions-with-job-summaries/

[7] https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/storing-and-sharing-data-from-a-workflow

[8] https://docs.github.com/en/get-started/writing-on-github

[9] https://github.com/orgs/community/discussions/31570#discussioncomment-3571340

output contains a table where each row describes the results for different pairs of RHEL kernel versions. The row contains the metrics described by requirements `CMP_TIME` and `CMP_STATS_1` in Table 4.3. Shorter comparison times are taken as positive impact and longer as negative. Green is also colored equal increases and decreases of not-equal, unknown, errors, and total differences. The opposite is colored red. If the numbers do not change, they remain black. Note that even if the number is colored green, it can theoretically have a negative impact (e.g., a semantically non-equal function is evaluated as equal because of a bug), but in general it should have a positive meaning. Below the table is an arrow enabling the user to show detailed info – names of functions evaluated differently than on the *master* branch. The figure serves just to demonstrate the output and different possibilities of results, in reality, it is unlikely that certain extensions would have a positive impact on some RHEL versions but a negative impact on a different one.

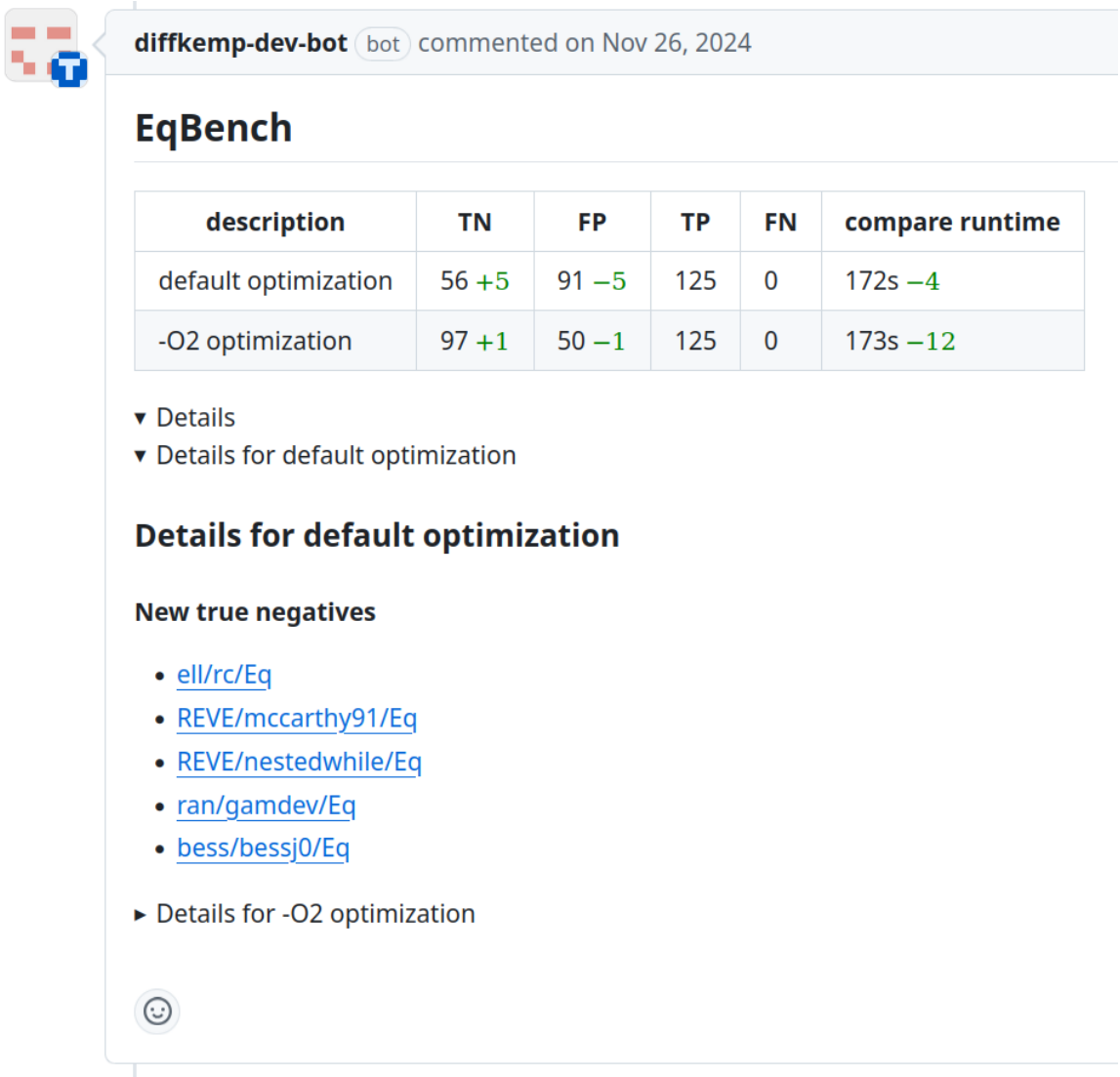



Figure 4.2: Design of evaluation results visualization for the EqBench dateset.

Figure 4.2 shows the evaluation output design for the **EqBench dataset**. Instead of metrics `CMP_STATS_1`, `CMP_STATS_2` are used (see Table 4.3. The increases in the correct

results (true positives/negatives) and the decreases in the incorrect results (false negatives/-positives) are colored green. The opposite is colored red. Below the table is again more detailed information, this time it contains the list of the changed program variants with URL links to their source code in the EqBench dataset repository.

### Labels

Labels describing the output of the evaluation will also be added to the PR. The labels are shown at the top of the pull request and provide fast status information about the PR. There will be three categories of labels, each for a different compared project. One label from each category will be added to the pull request at a time. The labels for the EqBench and the RHEL kernel KABI functions are shown in Figure 4.3, the labels for the RHEL kernel sysctl parameters will be similar to those of the KABI ones but instead of „RHEL functions“, they will contain „RHEL sysctl“.

For the **EqBench dataset**, there are the following labels:

- Success: It is used in the case the results are stable (did not change) or have an increase in correct results (true positives/negatives). The label marks the positive impact of the PR.
- Failure (negative impact): It is used for increases in false negatives/positives.

The worst label is always used – for example, if there are new true positives as well as new false positives, the label *failure* will be used.

For the **RHEL kernel**, there are three different labels:

- Success: It is used when the results did not change.
- Warning: It warns the reviewer that the result is ambiguous. It is used for an increase in the equal functions since, as previously mentioned, it does not necessarily mean a positive impact. It is also used for an increase in the total number of differing functions, which we decided to mark in the comment table as red, however, it may mean that the analysis managed to handle new types of changes and therefore proceeded further than before and analyzed more functions.
- Failure: It is used for the rest of the cases.


Figure 4.3: Caption

### Commit Statuses

Lastly, new *commit statuses*[10] will be added to the existing ones. Statuses are shown at the bottom of the pull request page near the button used for merging the changes to the

[10] https://docs.github.com/en/rest/commits/statuses?apiVersion=2022-11-28#about-commit-statuses

*master* branch. Statuses informs which continuos integration checks were successful and which not. The status of each project will be added, it will be marked as success/failure on the same bases as the labels. In the case of a warning label, the status check will be marked as success.

### 4.2.4 Design of the Architecture

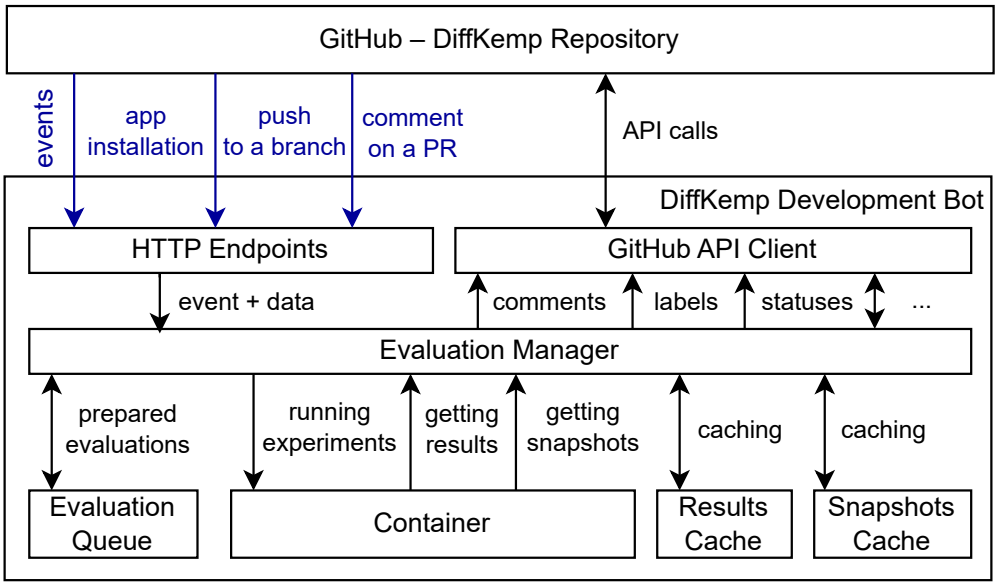



Figure 4.4: Proposed architecture of the extension for the impact evaluation of DiffKemp pull requests.

As described in Section 4.2.1, we will implement the extension for the impact assessment of DiffKemp pull requests by creating a GitHub App – a self-hosted server application. We call the extension **DiffKemp development bot**. The architecture is shown in Figure 4.4 and it contains the following components:

- **HTTP endpoints:** These will use the webhooks to receive data from GitHub about subscribed events as they occur. We will subscribe to three different types of events: (a) an app installation, (b) a push of commits to a branch, and (c) a comment creation on a pull request.
- **Evaluation manager:** It processes the received events based on its type:
  - For **app installation** and **push to a branch** events, the manager will perform semantic comparisons of the selected projects using DiffKemp located on the *master* branch. The comparison results and created snapshots will be saved to the caches to be reused later. This will accelerate the PR assessment because we already have results for the *master* branch.
  - For a **comment creation** event, the manager will run a semantic comparison of the selected projects using DiffKemp located on the PR branch. The cached

snapshots from the *master* branch will be used if possible. From the results cache, the results of the *master* branch will be retrieved. The *master*/base results and PR/feature results will be compared, and a report will be created and posted as a comment to the PR.

- **Evaluation queue:** We will run the evaluation of a maximum of one PR at a time so that we do not unnecessarily overload the server. If the evaluation is running and another evaluation is requested, the new evaluation will be put into the queue, where it will wait for its turn.
- **GitHub API client:** The component will manage creation of authentication with GitHub, sending API requests, and receiving responses from GitHub. This will be used for creating comments on PRs with the results of the analysis.
- **Container:** The project semantic comparison will be run inside a container (see Section 3.4) to ensure isolation from other software on the server. The evaluation of each project will run in parallel with other projects to reduce time. The container image will contain sources of the selected projects and DiffKemp and their dependencies to speed up setting things up for the evaluations. If DiffKemp dependencies change, the image will be updated with new dependencies, so that the dependencies do not have to be installed during a PR assessment.
- **Results cache:** It will contain the comparison results of the projects analysis done on the *master* branch. They will be recovered for comparison with results gained on a PR branch.
- **Snapshots cache:** It will contain snapshots created on the *master* branch. If the rebuild option is not specified, the snapshots will be reused for comparisons with DiffKemp located on the PR branch. Only the latest snapshots will be kept in the cache.
- **Abort mechanism:** It will be used if an evaluation of a branch (the *master* branch or a PR branch) is
  - in progress or
  - located in the evaluation queue

  and during that
  - a push to a branch occurs or
  - a new evaluation of the same branch is requested (e.g., with different options).

  The previous evaluation will be canceled – removed from the queue or terminated if in progress.

### 4.2.5 Detailed Design

In this section, we will look more closely at the processing of the individual events.

#### Push to a Branch

When a push to a branch occurs, it is first checked if another evaluation of the branch is not in progress or in the queue and, if so, the other evaluation is aborted. The aborted evaluation could be:

- **Evaluation of the *master* branch**: If the *master* branch is updated, the evaluation is aborted to release resources and cache the results of experiments for the current *master* version.
- **Evaluation of a PR branch**: If a PR branch with a triggered evaluation is updated, the evaluation is aborted because finishing the evaluation and providing results would not reflect the current state of the branch. The evaluation is not automatically rerun on the updated branch in case more changes are pushed to the branch. To rerun the evaluation, the reviewer must manually trigger it using the command in the PR comments (see Section 4.2.2).

If the event is a push to the *master* branch, evaluation of it is placed in the evaluation queue. The evaluation is later processed in the following way:

1. It is checked if the pushed commits change DiffKemp dependencies. If they do, the container image (see Section 2.1) is updated with the new dependencies utilizing the rebuilding of only the changed image layers (see Section 3.4) to speed up the process.
2. The evaluation manager launches a new container.
3. In the container, DiffKemp is checked out on the *master* branch and set up with the latest supported LLVM version.
4. In the container, snapshots of the projects are generated, followed by their comparison. Individual projects are analyzed in parallel.
5. The created snapshots are saved to the snapshot cache.
6. The results are saved to the results cache.

#### App Installation Event

The event is triggered when our app is installed in the DiffKemp repository. It creates and caches the first comparison results for the *master* branch. The processing is the same as in the case of a push to the *master* branch.

#### Comment on a PR

The processing of a new comment on a PR is following:

1. It is checked if the comment contains the command for evaluation described in Section 4.2.2, if not, processing stops.
2. The author of the comment is checked. The processing is stopped if the author does not have admin or write permission to the repository. This is done to prevent running of unknown code from unknown authors, which could have malicious intentions.

3. Configuration for the evaluation is extracted from the comment.
4. If no other evaluation is being processed, the processing continues, otherwise the evaluation is placed to evaluation queue and the processing continues after it is its turn.
5. Results for the *master* branch are recovered if they exist (commit hash is used for caching and restoring) and if `--cmp-opt` was not specified by the reviewer (see Section 4.2.2). Otherwise, steps 3–4 from processing of a push to the *master* branch are executed, but instead of running all experiments only the one specified by the reviewer are run (see Section 4.2.2). The next processing is done parallely with this step.
6. A new container is launched, in the container DiffKemp is setup – checked out to the PR branch and build with the latest supported LLVM version.
7. If snapshots from the *master* branch are cached they are recovered to the container.
8. Projects selected by the reviewer are evaluated – snapshots for the projects are generated (if they were not recovered) and the snapshots are compared (using additional compare option if provided).
9. When both results (for the *master* branch and the PR branch) are gained, the results are compared.
10. As described in Section 4.2.3, comments are created and labels and commit statuses are updated based on the results.
11. The evaluation queue is checked and if it is not empty, the first evaluation from the queue starts to process.

## 4.3 Implementation

In this section, we will focus on the implementation of the extension (DiffKemp development bot), mainly on the technologies that were used for the architecture described in Section 4.2.4.

### Server, HTTP Endpoints, GitHub API Client

To create the app, we need a server that listens to the GitHub events. We could choose from multiple languages and frameworks to do that. We also need a component to call the GitHub API (e.g., for creating comments with the assessment result). We could create the API calls manually, but the biggest problem would be creating authentication with GitHub which is not quite easy. There exists an official GitHub library (`Octokit`) and multiple third-party libraries solving this issue[11]. Also, multiple frameworks for creating a new GitHub App exist, solving the API call problem and simplifying receiving and validation of webhooks. Some of the frameworks are:

- Quarkus GitHub Bot[12] (a framework for Java),

[11] https://docs.github.com/en/rest/using-the-rest-api/libraries-for-the-rest-api?apiVersion=2022-11-28

[12] https://github.com/quarkiverse/quarkus-github-app

- `go-githubapp`[13] (a framework for Go), and
- Probot[14] (a framework for NodeJS).

The Probot framework was selected as it is quite a popular and active framework with a lot of contributors at the time when the selection was made.

As the framework is for NodeJS, two languages were options to use – JavaScript [17] and TypeScript[15]. TypeScript was chosen for the reason of type checking. REST API was used to call the GitHub API.

### Container, DiffKemp Setup

For isolation and reproducibility, the evaluations will be run in a container. As described in Section 3.4, there are multiple available containerization technologies. Container orchestration was eliminated from the options because we have only one available server, and orchestration is useful for multiple servers. Between Podman and Docker, **Podman** was selected because of the property that it can be run without root privileges, which is good for security purposes.

As mentioned in Section 3.2, DiffKemp provides **Nix flakes** that contain specifications of dependencies required by DiffKemp and allows one to build it with just a single command. By default, it also builds DiffKemp with the latest supported LLVM versions. For this reason, Nix flakes were selected to install DiffKemp dependencies and build it. Also, if DiffKemp dependencies will change in the future (recently happened with the addition of the SMT extension), we will not have to specify added dependencies in our container because the Nix flakes will already contain them. To be able to easily use the Nix flakes inside the container, `nixos/nix`[16] was selected as the base image for our container because it contains the Nix package manager necessary for the Nix flakes.

The container image contains source codes of the selected projects and preinstalled dependencies. If dependencies change, the image layer containing the dependencies is rebuilt using cache busting[17].

When running evaluations for a branch, one container is created. Theoretically, the assessment of each project could be run in its own container, but it was done in this way to eliminate the time required to build DiffKemp and simplify the caching of snapshots and results. The RHEL kernel sources have to be duplicated, so evaluation of the sysctl parameters and KABI functions could be run in parallel without causing data races. For the same reason, all snapshots cannot be compared at the same time, so first is compared one half and then the second one.

### Snapshots Caching

The caching of snapshots is managed by copying them from the running container to the host machine, and recovery is managed by copying them back. Other options were considered, such as saving the snapshots directly to the container image or mounting [24] the snapshots from the host system to the container. Saving them to image would require periodically

[13] https://github.com/palantir/go-githubapp
[14] https://github.com/probot/probot
[15] https://www.typescriptlang.org/
[16] https://hub.docker.com/r/nixos/nix
[17] https://dev.to/kenmoini/cache-busting-in-docker-2ngh

rebuilding the image, which seemed to be an unnecessary overhead. The mounting was not used as:

1. copying the snapshots does not take too long (approximately 40 s to copy all snapshots) and
2. mounting them would expose them to changes.

### Results Caching

Caching of results is managed by serializing the necessary information to a file and saving it to a folder on a host machine specified by the hash of DiffKemp commit on which the evaluation was done.

### Evaluation Manager – Parallel Processing

Initially, a problem with the **parallelization** of the evaluations appeared to arise because JavaScript/Typescript is a single-threaded language [17]. There exists `worker_threads`[18] module that allows the use of *threads* in JavaScript, but there is a big overhead for creating them. In the end, *promises* and *child processes* were used.

A **promise** is an object used to run an asynchronous operation. The promise can be in one of three states: (a) pending, (b) fullfilled, or (c) rejected. The promise is fulfilled if the operation is completed successfully and rejected if an error occurs in the operation. Otherwise, it is pending. Promises can be nested, and even multiple promises can be run concurrently (only one promise will be executed at a given time, but control can shift between different promises) [17].

**Child processes**[19] is a module providing the ability to spawn subprocesses. It is possible to create the subprocesses asynchronously by using promises – which means a new process is spawned in a subprocess, and the promise fulfills/rejects when the subprocess finishes. Meanwhile, other parts of the code (promises) can run.

Because the longest time in our application will be spent by running commands in a container, it is important to parallelize this part. Therefore, multiple subprocesses (child processes) are spawned concurrently by using the promises. The subprocesses then run multiple commands (e.g., each for a different project) in parallel in the container.

Using promises can sometimes be dangerous (for example, forgetting to wait until they are fulfilled), `typescript-eslint`[20] linter was used during the development to overcome this. It contains a rule that checks that the promises are properly treated.

### Abort Mechanism

For the abort mechanism, the NodeJS `AbortControler`[21] class was used. The class allows sending a cancellation signal to a promise by triggering an abort signal on it. Unfortunately, by triggering the signal, the promise does not end immediately. In certain parts of the code, the promise can check if the signal was triggered and change its behavior (e.g., throw an error). It can also set a callback function called when the signal is triggered, but the function itself cannot end the promise.

[18] https://nodejs.org/docs/latest/api/worker_threads.html
[19] https://nodejs.org/docs/latest/api/child_process.html
[20] https://typescript-eslint.io/
[21] https://nodejs.org/api/globals.html#class-abortcontroller

The `AbortController` class was utilized by checking the signal at certain places and throwing an error if the signal was triggered. Running evaluations in the container raised a problem because the signal would be checked after the command running in the container would finish. This is undesirable because the command can be completed after a long time (e.g., after more than 30 minutes for snapshot generations of the KABI functions in the RHEL kernel). To solve this, the callback function was used – killing the running container. This causes an error exception in the promise running a command in the container. The exception can then be processed, and the promises can end as rejected.

### Evaluation Queue

The evaluation queue was implemented using a mutex from the `async-mutex`[22] library. Abortion of an evaluation in the queue is managed by sending the abort signal to it. When the promise with the evaluation starts processing (after the mutex unlocks it), the abort signal is checked, and if it was triggered, then the evaluation ends with an error.

### Evaluation Command Processing

For processing of the evaluation command described in Section 4.2.2, `commander`[23] library was used. Initially, for more familiarity with the interface, a different library was selected – `argparse`[24], a port of Python's argument processing module. The library was exchanged because it ended the entire application server if the command written by a reviewer was incorrect. In the case of the `commander` library, it is possible to catch the problem and inform the reviewer about the incorrect command.

### The EqBench Dataset Evaluation

For the evaluation of the EqBench dataset, the script[25] mentioned in Section 3.3 was updated and used. The changes were related to output format and making the tool compatible with the usage of Nix flakes.

[22] https://www.npmjs.com/package/async-mutex
[23] https://www.npmjs.com/package/commander
[24] https://www.npmjs.com/package/argparse
[25] https://github.com/diffkemp/eqbench-workflow

# Chapter 5

# Evaluations of Created Automations

This chapter evaluates the developed solutions. In Section 5.1, the first part of this thesis is evaluated – DiffKemp development bot, which should simplify reviewers' evaluation of the impact of new DiffKemp features. The second solution was not evaluated because of a lack of time.

## 5.1 Evaluation of DiffKemp development bot

The bot was tested during development on a copy of the main DiffKemp repository by creating and updating pull requests, creating comments on them, and by updating the *master* branch.

The bot was also used on existing/previous pull requests. The bot was not directly evaluated on the main repository in case a bug was found. A script was created that runs the bot on a selected list of pull requests from the main repository. The script, instead of creating comments on the pull requests, saves a report for each pull request to its own Markdown file.

The script was not used on all available pull requests (PRs), but on a selected subset. The pull requests that were considered had to be created after the Nix flakes were introduced to the DiffKemp development environment (see Section 3.2), because the bot needs them to be able to build DiffKemp and run it. Of the possible PRs, fourteen[1] were selected. Thirteen of them are PRs that are changing the snapshot comparison process (see Table 4.1). The last one is the SMT extension (added recently) that requires adding a specific option when comparing snapshots.

While running the bot, a few bugs and problems were discovered. One of the problems was that for older PRs DiffKemp contained Nix flakes that used an older compiler, making it impossible to build snapshots for RHEL v8.4 and v8.5. This was fixed by installing a newer version of the compiler in such a case. Other discovered problems were also fixed.

While analyzing the reports for individual PRs, a few additions were added to report for RHEL functions and RHEL sysctl parameters (see Section 4.2.3) to make it easier for the reviewers to evaluate for the compared functions, which result changed (e.g., from *not*

[1]The selected PRs are 307, 309, 312, 315, 319, 322, 323, 325, 330, 349, 353, 364, 366, and 373 from the main DiffKemp repository https://github.com/diffkemp/diffkemp/pulls.

*equal* to *equal*) if the current result is correct or was the previous. The following things were added:

- In case a new differing function was discovered by DiffKemp or it was eliminated, information about the function is provided:
  1. the name of the function,
  2. path to the file in which is located the definition of the function in the older RHEL version and in the newer RHEL version,
  3. list of compared (KABI) function from which is called the function marked as (not) containing semantic differences, and
  4. syntactic difference of the differing function.

After adding this information to the reports, it was again analyzed, if based on the report, a reviewer would be able to tell if the given PR/feature improves DiffKemp capacity or if it has a negative impact. Or at least if it would simplify the task of further analysis of how the feature impacts DiffKemp functionality. The summarized results of the analysis are shown in Table 5.1.

| **Description** | **Number of PRs** | **PRs ID** |
|---|---|---|
| **Same results** | 5 | 319, 323, 325, 349, 373 |
| **Same results** (would require adding additional projects) | 2 | 309, 364 |
| **Introduced incorrect results** (visible from detailed information) | 1 | 307 |
| **Eliminated incorrect results** (directly visible from a table) | 1 | 366 |
| **Eliminated incorrect results** (some results require manual checking) | 3 | 312, 330, 353 |
| **Would require further analysis** | 2 | 315, 322 |

Table 5.1: Results of manual analysis of reports provided by DiffKemp development bot

In the following text, the findings will be described in more detail:

- For 5 PRs, the report does not show changes in equal, non-equal, or differing functions. Although the PRs changed in some way, the semantic comparison had **zero impact on the used projects**. All of the PRs fix some bugs, but it is unknown on which project the bug was encountered or maybe it was encounter only when implementing different functionality. However, it is unknown how our solution could be improved to make the impact of these changes visible.
- For 2 PRs, there is also no visible impact on the selected projects, but **adding more projects should show PRs impact**, specifically adding the cryptographic libraries (see Section 2.3.3). In PR 319 it is explicitly mentioned in the description that it fixes a bug encountered while analyzing the cryptographic libraries. In PR 364, it is not mentioned, but the PR was created by me and if I am not mistaken, it was also a bug that I encountered while analysing the cryptographic libraries.

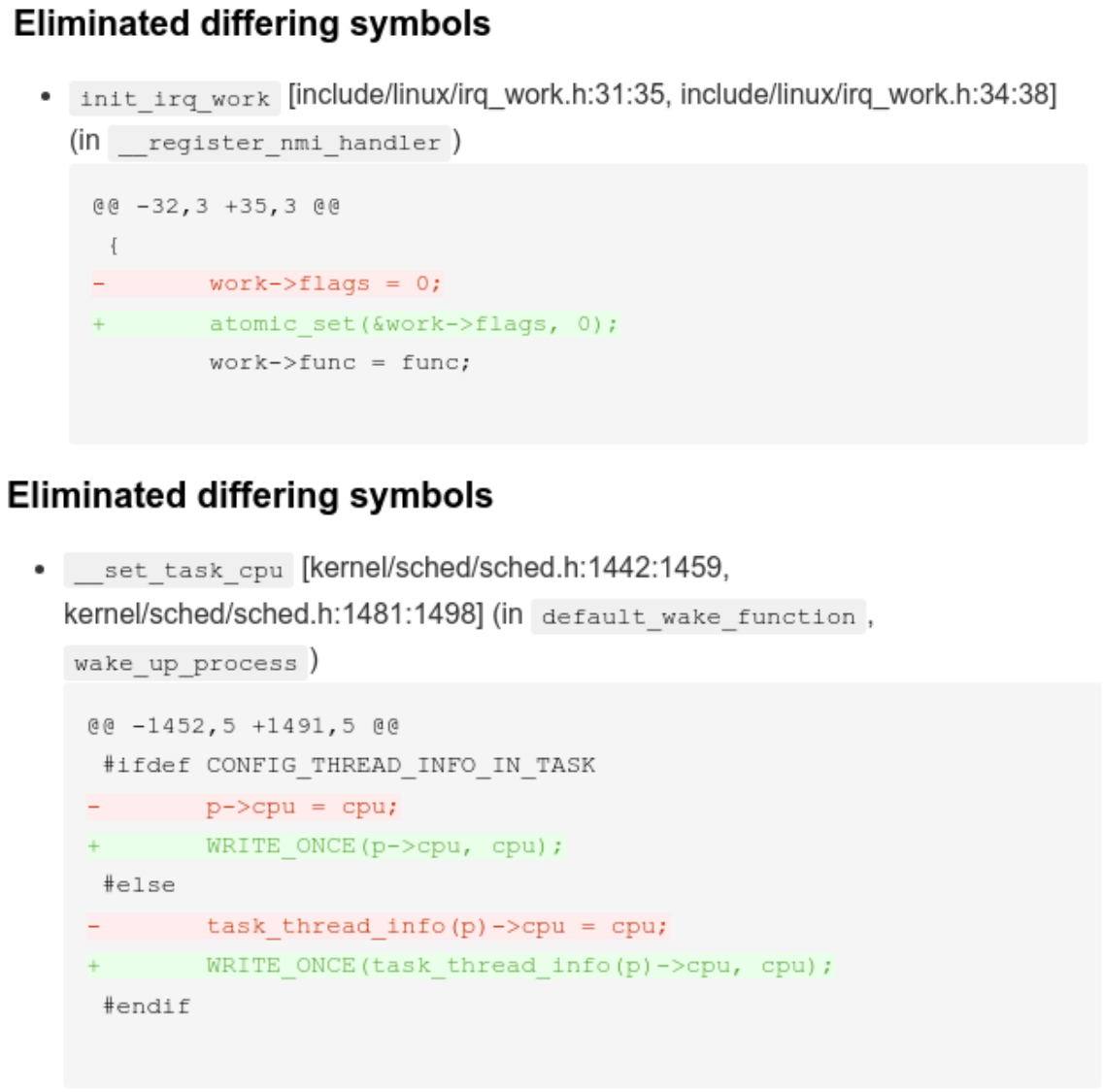



Figure 5.1: Some parts of the detailed report for PR with ID 307

- In the bot report for PR with ID 307, it is visible that the PR had **negative impact on the results**. The PR was fixing a bug and introducing a new pattern (grouping local variables to a structure). On the first sight, the results look like positive because there is increase in the RHEL functions evaluated as equal, which could be thanks to the addition of the new pattern, but when looking at the more detailed information about which functions were previously evaluated as non-equal and now are equal, we can see from provided syntactic differences of these functions that they should not be semantically equal.

  In Figure 5.1, we can see some part of the report, specifically the problematic syntax diffs from which imply that the PR introduces false negatives. We can see two different types of changes:

  1. Setting of variable value using `atomic_set` function instead of directly.

2. Setting of variable value using `WRITE_ONCE` macro instead of directly.

These in low-level code changes the semantics.

- PR with ID 366 added support for debug instruction that does not need to be examined and can be skipped in the comparison process. From the report, it is clearly visible that it **eliminated one false positive** in the EQBENCH dataset.
- For 2 PRs, it is visible from the report that there is a decrease in the number of differing functions in the RHEL kernel. For one (312) there is an increase in non-equal function but the PR fixes the bug previously mentioned and introduced in PR 366. When studying the detailed reports, it is visible for some functions that they should indeed be evaluated as semantically equal (non-equal in case of PR 312), for others it is **not clear from provided information**, and it is necessary to analyze the results further manually. The reason is, for example, that some code was moved to a new function that is called, and it is not possible to review it from the provided information.

  Examples of eliminated differing functions that are clearly visible from the report that are correct are shown in Figure 5.2 and 5.3.

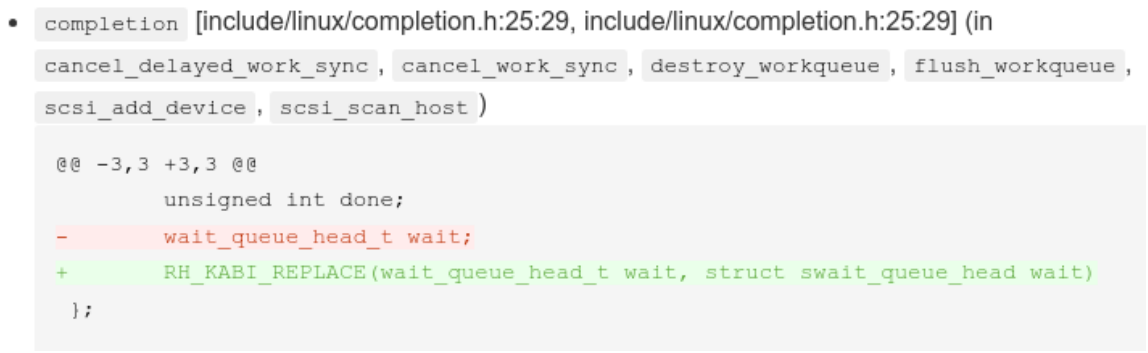



Figure 5.2: Detailed report for eliminated differing function in PR 330


Figure 5.3: Detailed report for eliminated differing function in PR 353

  - In the first figure, `RH_KABI_REPLACE` macro[2] is a construct that maintains KABI compatibility while allowing internal modifications to data structures. It utilizes

[2] https://fedorapeople.org/cgit/thl/public_git/kernel.git/tree/0001-kABI-Add-generic-kABI-macros-to-use-for-kABI-workaro.patch?h=kernel-5.8.0-0.rc0.20200608gitaf7b4801030c.1.vanilla.1.fc31&id=d1b6f8c7af0eb9a0a44b2d4723e58dde5eafa236

a union to overlay a new field onto an existing one, ensuring that the size and alignment of the structure remain unchanged. This false positive was eliminated in PR fixing a bug concerning the handling of union types in DiffKemp.

  - Second figure shows replacing of return type of a function, instead of `int` is used `bool`. This is a semantic-preserving change. The PR that eliminated this false positive added a pass that is run after function inlining.

- For the remaining 2 PRs, it was not possible to check the impact based on the report, and it requires a more thorough examination.

### Summarization

To summarize the findings, if the created solution existed when the selected pull requests were created, the bot would alert the reviewer in 50% of the PRs that they have some impact (with the merger of a PR, DiffKemp will provide different results), if the cryptographic libraries would be added the number would probably increase to 64%. In 5 of 14 PRs, the reviewer would immediately see if the impact is positive or negative, but in 3 cases, it would also need manual review to confirm the impact for all results that changed. Lastly, in 2 cases, it would be visible that the PRs have an impact, but the results would need to be manually inspected. But the report would provide enough information that the reviewer would not need to compare all KABI functions, but only the functions listed in the report, or just more thoroughly examine the definitions of the function whose source location is included in the report.

# Chapter 6

# Conclusion

The objective of this thesis was to create two tools which would be useful for developers of DiffKemp, the static semantic equivalence analysis framework:

1. A tool that would automatically evaluate new versions of DiffKemp on selected open-source projects and benchmarks, providing DiffKemp's developers feedback on how new features impact its performance (specifically, correctness of evaluation).

2. A tool that would automatically run DiffKemp on new versions or patches of selected open-source projects, simplifying the review of results and assisting in the identification of whether the results provided by DiffKemp are correct or not.

The main goal of this thesis was achieved, both tools were created, but the design and implementation of the second tool are not documented here because of a lack of time.

In this thesis, we became acquainted with DiffKemp, its architecture, development process, and additional scripts that were created in the past. We also reviewed related theses and papers, specifically focusing on the experiments that were performed in those works. We analyzed the methods, metrics, and projects that were used in those experiments.

Then, the first tool, the DiffKemp development bot, was designed based on previous pull requests and explored experiments. Its aim is to improve the review of new features and accelerate the incorporation of changes into the production version to developers of DiffKemp. The tool was implemented using the Probot framework. It compares the results provided by DiffKemp with and without the feature on multiple versions of the Red Had Enterprise Linux (RHEL) kernel and the EqBench dataset. The comparison results are then presented to the reviewers.

Evaluation of the created solution on the past pull requests (feature proposals) showed that the bot would inform the reviewer in 50% about the impact of the pull request (PR). In approximately 70% of them, the type of impact would be able to be determined directly from the bot report, and in the rest of the cases, the reviewer would need to do a more thorough manual inspection, but the bot would provide a base direction on what should be analyzed.

Future work could improve the first tool by adding more projects on which new features can be automatically evaluated (e.g., cryptographic libraries), to allow broader evaluation of new features. The output could be enhanced by providing other information to simplify the reviewers' analysis even more. The command interface could be extended with more options that could be useful in certain use cases (e.g., specification of which LLVM version should be used, evaluation of only certain pairs of the RHEL kernel versions, ...).

# Bibliography


[1] AM. *Optimizing Linux Like a Pro: The SysAdmin's Guide to sysctl* online. 2024. Available at: https://medium.com/it-security-in-plain-english/optimizing-linux-like-a-pro-the-sysadmins-guide-to-sysctl-f4d4ce43d0fd. [cit. 2024-12-16].

[2] BADIHI, S.; LI, Y. and RUBIN, J. EqBench: A Dataset of Equivalent and Non-equivalent Program Pairs. In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, p. 610–614.

[3] FELSING, D.; GREBING, S.; KLEBANOV, V.; RÜMMER, P. and ULBRICH, M. Automating regression verification. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2014, p. 349–360. ASE '14. ISBN 9781450330138. Available at: https://doi.org/10.1145/2642937.2642987.

[4] GITHUB. *GitHub Docs: About pull request merges* online. 2024. Available at: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/about-pull-request-merges. [cit. 2025-01-02].

[5] GITHUB. *GitHub Docs: Contributing to a project* online. 2024. Available at: https://docs.github.com/en/get-started/exploring-projects-on-github/contributing-to-a-project. [cit. 2024-12-28].

[6] GITHUB. *GitHub Docs: GitHub Actions documentation* online. 2025. Available at: https://docs.github.com/en/actions. [cit. 2025-01-02].

[7] GITHUB. *GitHub Docs: GitHub Apps documentation* online. 2025. Available at: https://docs.github.com/en/apps. [cit. 2025-01-17].

[8] GITHUB. *GitHub Docs: Webhooks documentation* online. 2025. Available at: https://docs.github.com/en/webhooks. [cit. 2025-01-17].

[9] GLOZAR, T. *Enhancing DiffKemp to Support Generic Projects*. Brno, 2022. Bachelor's thesis. Masaryk University, Faculty of Informatics. Available at: https://is.muni.cz/th/rr7dt/.

[10] IBM. *What is containerization?* online. 2024. Available at: https://docs.github.com/en/webhooks. [cit. 2025-01-19].

[11] KUČMA, T. *Generating Code Change Patterns from C*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/studenti/zav-prace/detail/157087.

[12] Kříž, D. *Automatic Generation of Code Change Patterns*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/studenti/zav-prace/detail/146333.

[13] Malecová, T. *Equivalence-Based Slicing of Programs*. Brno, 2021. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/24038.

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

[15] Malík, V. *Static Analysis of C Programs*. Brno, 2023. PhD thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/phd-thesis/1059.

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

[17] Mdn. *Mdn Web Docs: JavaScript* online. 2025. Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript. [cit. 2025-01-20].

[18] Mora, F.; Li, Y.; Rubin, J. and Chechik, M. Client-specific equivalence checking. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 441–451. ASE '18. ISBN 9781450359375. Available at: https://doi.org/10.1145/3238147.3238178.

[19] Namespace. *Running GitHub Self-Hosted Runners Reliably* online. 2023. Available at: https://namespace.so/blog/running-github-self-hosted-runners-reliably. [cit. 2025-01-16].

[20] Nečas, F. *Applying formal methods to analysis of semantic differences between versions of software*. Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/157112.

[21] Nix. *Nix Reference Manual* online. 2025. Available at: https://nix.dev/manual/nix/2.25/. [cit. 2025-01-04].

[22] Patrik, N. *Automatic Forward Slicing of Programs*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.fit.vut.cz/study/thesis/23100.

[23] Petr, L. *Visualisation of Static Comparison of Semantic Equivalence of Different Versions of Software Written in C*. Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/144949.

[24] Podman. *Podman documentation* online. 2019. Available at: https://docs.podman.io/en/latest/. [cit. 2025-01-19].

[25] Project, T. L. *LLVM Language Reference Manual* online. 2024. Available at: https://llvm.org/docs/LangRef.html. [cit. 2024-12-16].

[26] RedHat. *What's the difference between Fedora and Red Hat Enterprise Linux?* online. 2023. Available at: https://www.redhat.com/en/topics/linux/fedora-vs-red-hat-enterprise-linux. [cit. 2024-12-17].

[27] RedHat. *What is Kernel Application Binary Interface (kABI)?* online. 2024. Available at: https://access.redhat.com/solutions/444773. [cit. 2024-12-17].

[28] Rossel, S. *Continuous Integration, Delivery, and Deployment: Reliable and Faster Software Releases with Automating Builds, Tests, and Deployment.* Packt Publishing, 2017. ISBN 1787284182.

[29] Scott Chacon, B. S. *Pro Git.* 2nd ed. Apress, 2014. ISBN 1484200772.

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

[31] Žáčik, P. *Analyzing semantic stability of cryptography libraries using Diffkemp.* Brno, 2024. Master's thesis. Masaryk University, Faculty of Informatics. Available at: https://is.muni.cz/th/ponkv/.