BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

AUTOMATIC FORWARD SLICING OF PROGRAMS

AUTOMATICKÉ DOPŘEDNÉ PROŘEZÁVÁNÍ PROGRAMŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **NIKOLAS PATRIK**
AUTOR PRÁCE

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

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Patrik Nikolas**
Programme: Information Technology
Title: **Automatic Forward Slicing of Programs**
Category: Software analysis and testing

Assignment:

1. Get acquainted with DiffKemp, a tool for automatic comparison of semantics of functions and parameters of the GNU/Linux kernel.
2. Study existing methods for static slicing of programs.
3. Design a method for static forward slicing of programs that would be able to safely remove code that is independent of a value of a chosen parameter of the GNU/Linux kernel.
4. Implement the proposed method as a part of program pre-processing within the DiffKemp project.
5. Evaluate the created solution on at least two publicly available versions of the GNU/Linux kernel. Discuss the influence of your extension on results of the analysis performed by DiffKemp.

Recommended literature:

- Official website of DiffKemp: https://github.com/viktormalik/diffkemp
- A. De Lucia, "Program slicing: methods and applications," Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, Florence, Italy, 2001, pp. 142-149.
- Harman, Mark & Hierons, Robert. (2001). An Overview of Program Slicing. Software Focus.

Requirements for the first semester:

- The first two points of the assignment.

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

Supervisor: **Malík Viktor, Ing.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: July 31, 2020
Approval date: July 25, 2020

Abstract

This thesis presents designing new forward slicing solution for the DiffKemp tool. After strenuous analysis of currently implemented solution in DiffKemp for forward slicing we decided to retain current solution and extend it by few enhancements that should improve the analysis provided by DiffKemp in a quite big scope. We have implemented extensions so DiffKemp can perform analysis on fields of structured types which might represent run-time parameters and also we extended slicing criterion with the value of analyzed variable. Also we added support for slicing module kernel parameters. After implementing this solutions, we did experiments which proved that implemented solution has improved the analysis performed by DiffKemp.

Abstrakt

Táto práca popisuje návrh a implementáciu nového riešenie pre nástroj DiffKemp na automatické dopredné prerezávanie programov. Po zdĺhavej analýze súčasného riešenia, sme sa rozhodli súčasné riešenie ponechať a rozšíriť ho o zopár vylepšení. Implementovali sme rozšírenie ktoré dovoľuje DiffKempu vykonávať analýzu nad prvkami štrukturovaných typov, pridali sme k súčasnému prerezávaciemu kritériu aj hodnotu premennej a na záver pridali podporu na analýzu parametrov modulov jadra. Po implementovaní týchto vylepšení sme vykonali experimenty ktoré potvrdili zlepšenie analýzi ktorú DiffKemp vykonával.

Keywords

static analysis, DiffKemp, forward slicing, llvm, llvm ir, clang

Kľúčové slová

statická analýza, DiffKemp, dopredné orezávanie programov, llvm, llvm ir, clang

Reference

PATRIK, Nikolas. *Automatic Forward Slicing of Programs.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Viktor Malík,

Rozšírený abstrakt

Táto práca popisuje návrh a implementáciu nového riešenie pre nástroj DiffKemp na automatické dopredné prerezávanie programov. Z počiatku sa práca zaoberá nástrojom DiffKemp, pre ktorý je určený výsledok tejto práce. Nástroj DiffKemp slúži na porovnávanie sémantiky dvoch verzií jadra Linuxu. DiffKemp sa spúšťa vo dvoch fázach. Prvou fázou je fáza generate, ktorá sa zaoberá prípravou zdrojových kódov a zbieraním informácií ktoré sú určené pre danú analýzu. Výsledná analýza je potom spustená až vo fázi compare. DiffKemp na túto analýzu využíva LLVM. Framework LLVM je súbor nástrojov, ktoré slúžia na uľahčenie práce so zdrojovými kódmi a taktiež umožňuje vykonávať rôzne transformácie. DiffKemp v súčastnosti už obsahuje riešenie na dopredné orezávanie programov avšak z dôvodu výslednej kvality riešenie sme sa ho rozhodli analyzovať. Po zdĺhavej analýze súčasného riešenia, sme sa rozhodli toto riešenie ponechať a rozšíriť ho o zopár vylepšení. Keďže táto práca je celá založená na prerezávaní programov rozhodli sme sa opísať prerezávanie programov v samostatnej kapitole. Táto kapitola obsahuje zadefinovanie rôznych pojmov ktoré budeme neskôr v práci používať. Jedným z týchto pojmov je aj prerezávacie kritérium. V následnej kapitole sme sa venovali implementácií rozšírení, ktoré dovoľuje DiffKempu vykonávať analýzu nad prvkami štruktúrovaných typov. Toto rozšírenie je založené na porovnávaní indexov inštrukcie, ktorá má za úlohu, vypočítať ukazateľ, ktorý ukazuje na prvok štrukturovaného typu. Táto inštrukcia sa nazýva Get Element Pointer inštrukcia a keďže táto inštrukcia je často mylne chápaná ako indexovací operátor v jazyku C, rozhodli sme sa ju bližšie opísať v tejto práci, keďže je na nej založené jedno z rozšírení prezentovaných v tejto práci. Ďalej sme pridali sme k súčasnému prerezávaciemu kritériu aj hodnotu premennej. Implementácia tohoto rozšírenia zahrňovala dve fázy. V prvej fáze sme nahradili všetky výskyty globálne premennej. Následne sme potom spustil rôzne štandardné transformácie ktoré poskytuje LLVM aby sme odstránili nedostupný kód potom čo sme vykonali prvú fázu. Posledné rozšírenie ktoré sme pridali je podpora na analýzu parametrov modulov jadra. Keďže časť riešenie DiffKempu už túto podporu obsahovala, rozhodli sme sa ju pridať aj do zvyšku. Na spúšťanie analýzi pre parametre modulov, sme sa rozhodli použiť formát podobný súčasnému avšak tak aby splňoval zopár jednoduchých kritérií. Po implementovaní týchto vylepšení sme vykonali samostatné experimenty. Pre rozšírenie ktoré implementovalo prístup k prvkom štrukturovaných typov sme nezaznamenali taký úspech ako sme očakávali, čo bolo pravdepodobne spôsobene už predošlým správnym vyhodnoteným z nástroja DiffKemp. U nasledujúcich rozšírení sa nám však podarilo ukázať ich skutočnú hodnotu a ako sa nám pomocou nich podarilo vylepšiť analýzu ktoré nástroj DiffKemp vykonával. Na záver sme zhodnotili výsledky tejto práce a spomenuli taktiež rozšírenia, ktoré by mohli taktiež vylepšiť analýzu vykonávanú nástrojom DiffKemp. Jedným z týchto rozšírení by mohla byť napríklad implementácia kontroly aliasovania ukazateľov.

Automatic Forward Slicing of Programs

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Viktor Malík acknowledgment

. . . .
Nikolas Patrik
July 29, 2020

Contents

1 Introduction **3**

2 The DiffKemp Static Analysis Tool **5**
2.1 Generate Phase 6
2.2 The Compare Phase 6
2.2.1 SimpLL 7
2.3 Current Automatic Forward Slicing Solution 7
2.4 LLVM intermediate representation 8
2.4.1 Global variables representation in LLVM IR 8
2.4.2 GEP instructions in LLVM IR 9

3 Program slicing **12**
3.1 Backward Slicing 13
3.2 Forward Slicing 13

4 Proposed Extensions of Slicing in DiffKemp **14**
4.1 Current slicing solution in DiffKemp 14
4.1.1 Computing control and data-dependent instructions in current slicing algorithm 15
4.2 Adding support to slicing against concrete fields of structured data types . 15
4.2.1 GEP Instruction Dependence 16
4.2.2 Call Instruction Dependence 16
4.2.3 Other Instructions Dependence 17
4.3 Slicing w.r.t. a Concrete Value of a Parameter 17
4.4 Adding support for slicing against kernel module parameters 18
4.4.1 Generate phase 18
4.4.2 Compare phase 19

5 Implementation of proposed solutions **20**
5.1 Current implementation of the DiffKemp 20
5.1.1 The Generate phase 20
5.1.2 The Compare phase 22
5.2 Slicing w.r.t certain field of structured data type representing run-time parameter 23
5.2.1 SimpLL 23
5.2.2 VarDependencySlicer pass and GEP instruction dependence 23
5.2.3 Compare phase and computation of the Call instruction dependence 24
5.3 Slicing w.r.t a Concrete Value of a Parameter 26

5.3.1 VarValueDependencySlicer 27
5.4 Supporting slicing against kernel module parameters in python interface . . 27

6 Experimenting with implemented solutions **29**
6.1 Slicing against concrete fields of structured types experiments 29
6.2 Slicing w.r.t to concrete value of global variable experiments on the kernel modules 30

7 Conclusion **32**

Bibliography **33**

Chapter 1

Introduction

In recent years where CI/CD (continuous integration and continuous delivery) techniques make it easier and faster to deploy more often, we find a need to analyze programs in such a pace as they are deployed. Unfortunately, analyzing programs is very complicated and time-consuming job. The usual solution, which is widely used across all the bigger projects, is running large set of test cases to find out any bugs. But this solution doesn't cover all of the paths the program can get into. The process of running test cases on running program is called dynamic analysis. Oppose to this methodology is static analysis. This way we can analyze all of the paths of a program. The task is sadly very complex and usually we can't analyze this way whole source code of application. For this purpose we use slicing methods where we use criterion of slicing. Criterion tells us which part of the application property we want to analyze. Based on this criterion we can afterwards analyze all of possible paths this criterion influenced in application. One of the tools out there, that are doing this kind of analysis is DiffKemp. DiffKemp is able to compare semantics of two version of Linux kernel by using these methods.

Hence, the goal of this thesis is to propose automatic forward slicer which can slice off all unnecessary statements from Linux kernel sources based on some parameters it contains. The parameter we choose is called criterion, based on which, we decide, what statements can be deleted. Product of slicing is called slice and it contains all of statements (instructions) which are dependant on certain program variable. The variable usually represents parameter of Linux kernel module. This way we can prove semantics of both kernel versions of certain module, is same. The DiffKemp tool, which automatic forward slicer is intended for, already contains some solution for forward slicing based on provided parameters. After learning how the current solution works and analyzing its results, we decided for retaining current solution because it was producing quality results in most of the cases. After analysing the current solution we noticed some deficiencies. Because we decided to retain current solution, we have added support for these deficiencies in form of extensions. These extensions are including support for slicing against fields of structured data types which might represent run-time parameters. Next extension is slicing with respect to concrete value of global variable. Last but not least extension provides support for analyzing the kernel module parameters, because current solution only provides support for run-time parameters.

The rest of thesis is organised as follows. In Chapter 2, is described DiffKemp, a static analysis tool which we use for analyzing Linux kernel modules (their functions) and compare these modules as two different versions. The goal of this process is to prove that semantics of certain parameter of module stays the same between versions. In its subsection we closely

introduce two phases in which DiffKemp performs its analysis. We also introduce LLVM intermediate representation which is very often referenced when describing how the various constructs work in DiffKemp. In following Chapter 3, we describe various slicing techniques and how they are used in practice, also how DiffKemp utilize these, in its own process of analysis and then we look more in depth for two slicing techniques which are forward slicing and backward slicing. Next chapter 4, presents brief description of the current solution for forward slicing in DiffKemp, following introduction of new design of solution, which removes all shortcomings of old ad-hoc solution. Then in the chapter with Experiments 6 we show our improvements of current solution really helped improve the results of whole analysis. Finally, conclusion and future work in Chapter 7, which describes the results of this thesis and some other extensions which can be made to improve current slicing solution even more.

Chapter 2

The DiffKemp Static Analysis Tool

DiffKemp is a static analysis tool that is able to automatically compare semantics of two versions of the Linux kernel [10]. Generally, it is not possible to compare the whole kernel at once and hence DiffKemp is designed to compare semantics of individual kernel functions or parameters. Parameters are usually represented by global variables in the Linux kernel modules. When comparing semantics of parameters, DiffKemp compares semantics of all functions that use the global variable corresponding to the parameter. Therefore, in the rest of this work, we assume that two functions are compared for semantic equality.

The practical use of DiffKemp is to partially automate checking of backwards compatibility and of stability of parts of the kernel. A kernel user may use some kernel functions and he may expect that a function does not change its behavior between versions. This is especially the case for the Kernel Application Binary Interface (KABI), which is a list of functions that are guaranteed to be stable across minor releases of the Red Hat Enterprise Linux. For kernel parameters, the situation is similar—if a user sets a kernel parameter to some value, he expects that the setting will have the same semantics in future versions and that he can preserve it during an upgrade. Checking whether the behaviour changed is not possible to be done manually, especially in such a large project, potentially containing millions of lines of code, as the Linux kernel is.

In order to compare all possible behaviours, DiffKemp uses various ways of static analysis to check semantic equivalence of two different versions of functions. This analysis is done on the sources of the Linux kernel. DiffKemp, likewise other static analysis tools, uses a structural low-level representation of programs for the analysis. In particular, it uses the intermediate representation of the Clang/LLVM compiler[4], referred to as LLVM IR[5].

In DiffKemp, the semantic comparison is done in two main phases as shown in figure 2.1:

1. **Generate** – creates a so-called snapshot of Linux kernel containing sources of all functions to compare, compiled into LLVM IR. This phase uses as the input the kernel sources and a list of KABI functions and kernel parameters to be compared.

2. **Compare** – compares semantics of list of functions represented as two snapshots created in the *generate* phase.

In the following sections, we describe the individual phases in a more detailed way.

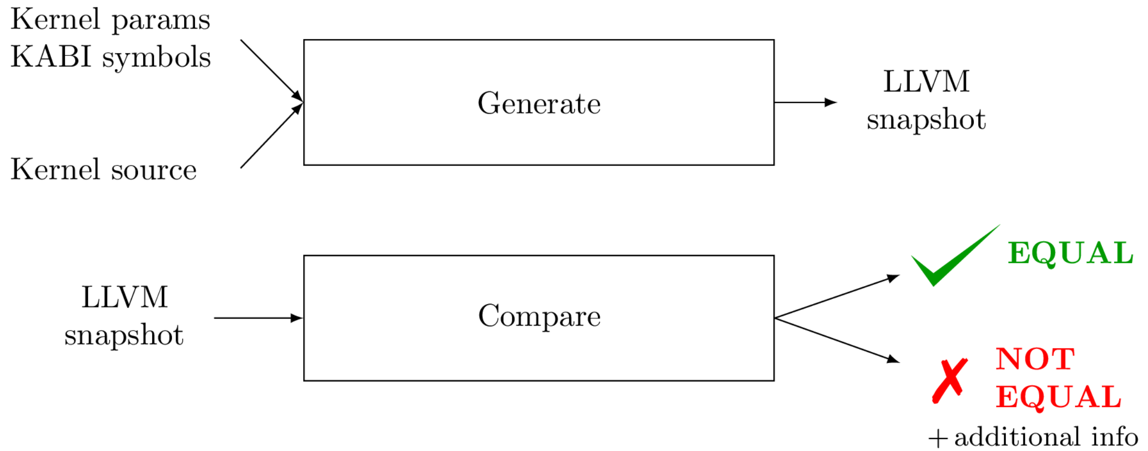

Figure 2.1: Architecture of DiffKemp - simplified[3]

2.1 Generate Phase

The phase of generating snapshot consist of two sub-processes. First, as shown in the figure 2.2, this phase take as and input the Linux kernel source with list a of KABI functions or kernel parameters which are to be analyzed. Subsequently, Diffkemp uses the utility `cscope`, and searches for definitions of the given functions (or of the functions using the given parameters) and creates mappings of these to the source files, where it finds these definitions. Afterwards, the found source files are compiled into LLVM IR. Finally, the LLVM snapshot is produced and it contains the original and the compiled sources of kernel, the file `functions.yaml` with mappings of functions to LLVM IR files and the files of source finding utility `cscope`, allowing quick searching of function definitions.

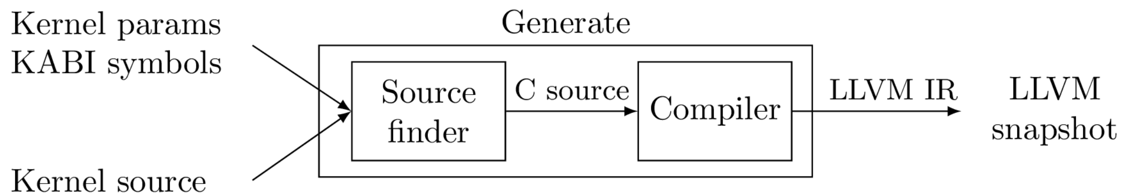

Figure 2.2: Generate phase – architecture[3]

2.2 The Compare Phase

The phase compares semantics of all functions from two snapshots created in the *generate* phase. This phase consist of three sub-processes as shown in Figure 2.3. First, the compared functions are simplified using various techniques. The goal of these simplifications is to remove parts of the functions not relevant for comparison of semantics. For instance, when comparing functions using a global variable (representing a kernel parameter), it is not necessary to compare the whole function. On the contrary, it suffices to compare those parts of functions that are influenced by the variable. DiffKemp uses a technique called slicing in order to remove parts of functions not influenced by a variable. Since this phase

is the core of this work, it is described in detail in Chapter 3. After the simplification the following phase is comparing of semantics of two simplified/sliced functions which ends either in semantic equivalence or, if the functions are not equal, an additional phase is run. In this last phase, when it was shown that the compared functions are not the same, DiffKemp tries to localise the difference between the two and displays the lines in the C code, where the difference occurs.

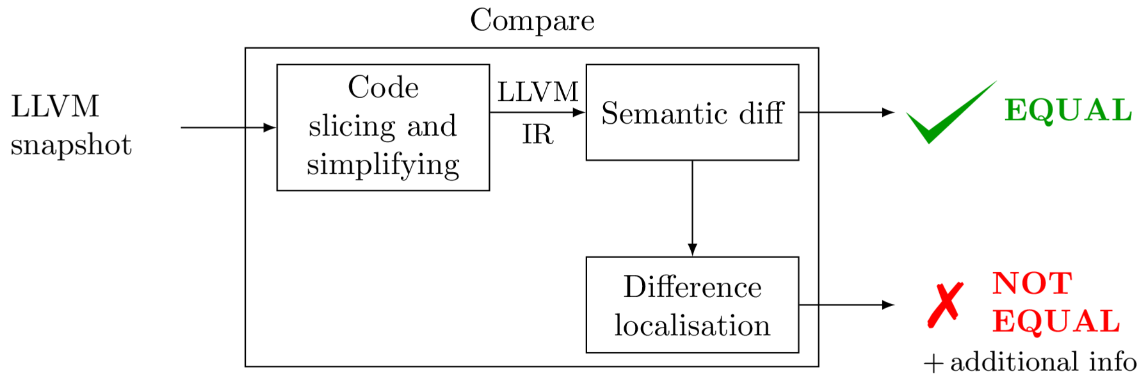

Figure 2.3: Compare phase – architecture[3]

2.2.1 SimpLL

This utility is the core of the compare phase. As its input it uses the LLVM IR sources of a pair of compared functions. On these sources, it runs various transformation passes, that are responsible for code simplifying. An instance of `SimpLL` is run for every functions that is compared. It simplifies and compares the given functions. Usually, the functions are simplified based on criterion, which might influence certain parts of a functions and therefore comparison of the given functions is easier. In some cases this criterion might be parameter of the Linux Kernel modules. This part of functionality of code slicing is implemented inside LLVM pass `VarDependencySlicer`, which we aim to improve in this work.

2.3 Current Automatic Forward Slicing Solution

The DiffKemp currently implements solution for automatic forward slicing. By analyzing how the current solution work we figure it out that retain current solution is better way to go and we decided to just extend its functionality by few improvements which can improve analysis in great manner. This improvements are:

1. Slicing with respect to certain field of structured type which represents run-time parameter.

2. Slicing with respect to concrete value of global variable.

3. Adding support for analyzing module parameters.

First improvement should improve currently performed analysis of DiffKemp a lot, because there are lot of run-time parameters which are represented as field of global variable

of structured type. This is because if there's a false positive that DiffKemp marks the result as not equal but non-equality was in part of the code that wasn't affected by given parameter for slicing, it means that after implementing this solution it should be evaluated correctly. On the other way if there wouldn't be any improvements by slicing algorithm this doesn't necessarily means that it didn't improved anything but that it might be in all of the cases difference even in the dependent parts of the code and the equality was previously evaluated correctly even tough it has contained the code which shouldn't be analyzed.

Second improvement improves the cases where current solution produces non-equal results for all possible values of global variable. In practice, users which would probably wanted to run the analysis wanted to test their current setup with concrete values that they have set. Slicing of the code based on the concrete values could potentially improve results because once again, the difference might be located in independent part of the code based on the concrete value of variable.

Last improvement is improvement in wider scope, because current slicing parameters set in current slicing algorithm are quite general, but DiffKemp provides support only for run-time parameters or the functions of the Kernel Application Binary Interface (KABI). So we added support also for analyzing module parameters because they are represented the same as run-time parameters are.

For implementing these extension and also for the current forward slicing algorithm is enormously used the LLVM framework. This framework provide easy manipulation with code which we analyze. LLVM is not only used for slicing solution but is largely used in other parts of DiffKemp. Because this thesis highly reference LLVM key concepts, we introduce these concepts in next subsection 2.4

2.4 LLVM intermediate representation

Huge part of the current solution is based on a LLVM framework. The LLVM framework is a collection of modular and reusable compiler and tool-chain technologies [6]. The most important sub-project of LLVM framework for this thesis is LLVM Core. The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support. These libraries are built around a well specified code representation known as the LLVM intermediate representation („LLVM IR“). LLVM IR is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation, and as a human readable assembly language representation[5]. The three different forms of LLVM are all equivalent. For purpose of this thesis there are two concepts from LLVM which must be explained to fully understand presented solution. These are global variable representation and GEP instruction. In the next subsection 2.4.1 we'll introduce how global variable are defined in LLVM IR and after that we'll describe how GEP instruction works in section 2.4.2.

2.4.1 Global variables representation in LLVM IR

Global variables define regions of memory allocated at compilation time instead of run-time. To define global variable we must first assign itself an identifier. LLVM identifiers come in two basic types: global and local. Global identifiers (functions, global variables) begin with

the '@' character[5]. Local identifiers (register names, types) begin with the '%' character. So definition of global variable might look like something like this:

```
@global_variable = ...
```

The three dots represent all creation parameters for defining global variable. It might contain linkage type, alignment, if it's constant or not, type and default value. For purpose of this thesis we just need to remember that global variable are defined with type and might have default value i.g. initializer. By C standard, global variable is always initialized with zero, unless it is external. Also worth to mention is that identifier of global variable doesn't represent the global variable itself but it only represent pointer to place in memory where given global variable starts. For working with the global variable we have to load a global variable with `load` instruction. This approach has its restrictions when dealing with the global variable of structured data types and we'll describe it more closely in the following chapter 4.

2.4.2 GEP instructions in LLVM IR

Component part of the proposed solutions in this thesis are extensions which makes use of GEP instruction. GEP is shortcut for Get Element Pointer which is pretty self-explanatory of what this instructions does. Anyway there are still some thing which are often misunderstood about this instruction. What GEP instruction really does is that it calculates resulting pointer based on the arguments given to the instruction, but it **never reads a memory**. The first argument is always a type used as the basis for the calculations. The second argument is always a pointer or a vector of pointers, and is the base address to start from. The remaining arguments are indices that indicate which of the elements of the aggregate object are indexed[5]. The interpretation of each index is dependent on the type being indexed into.

First index of GEP instruction

The first type indexed must be a pointer value, however the following types can be structured types as arrays, vectors or structs. Also the first index must always index into the second argument pointer type. This is often misunderstood because people tend to relate it to known concepts from other programming paradigms, most notably C array indexing and field selection. Confusion with first index in GEP instruction usually arises when people think of GEP instruction like it was C indexing operator[7]. Let's look at this example:

```
 AType *Foo;
...
X = &Foo->F;
```

There might be temptation to say there would be only one index and thus selection of field `F`. But `Foo` is pointer and therefore must be indexed explicitly. So we would provide GEP instruction two indices. The first operand indexes through the pointer and the second one select field `F` of the structure. This would analogically be implemented in C like this:

```
 X = &Foo[0].F;
```

Type of the index

The type of each index argument depends on the type it is indexing into. When indexing into a (optionally packed) structure, only i32 integer constants are allowed (when using a vector of indices they must all be the same i32 integer constant). When indexing into an array, pointer or vector, integers of any width are allowed, and they are not required to be constant. These integers are treated as signed values where relevant.

In the following listings is shown how the C code would be interpreted by Clang compiler:

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};

int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

The LLVM code generated by Clang would be then:

```
%struct.RT = type { i8, [10 x [20 x i32]], i8 }
%struct.ST = type { i32, double, %struct.RT }

define i32* @foo(%struct.ST* %s) nounwind uwtable readnone optsize ssp {
entry:
  %arrayidx = \
  getelementptr inbounds %struct.ST, %struct.ST* %s, \
  i64 1, i32 2, i32 1, i64 5, i64 13
  ret i32* %arrayidx
}
```

Although the syntax of LLVM IR is little bit different from C syntax and as we mentioned earlier that is more like assembly we can easily associated certain parts of the code in LLVM IR with C code. At the beginning we define our structures. As shown above, defining structures is just declaring that it will contain fields of certain types. Next we define function `foo` with argument of the struct type we declared earlier.

Finally we get to the GEP instruction. Let's look at it more closely. In this chapter we mentioned that the first argument is type used as basis for calculation and the second one is actual base address from which the calculation is made. Remaining arguments as we mentioned earlier are the indices. Every index consist of pair of type and value. As we can see in C code fragment first index is 1. This is index into the pointer so we don't have to use explicit 0 as shown in previous chapter. This index is `i64` type, which implicitly indicates that the machine where the code will be run is 64 bit architecture, because pointers are just memory addresses and knowing the address is 64 bit long we derive previous statement. Next argument is index 2 with `i32` type. As we already know indices indexing fields of

structure must be i32 types and the field with index 2 is the structure of `struct RT` type. The next index `1` indexes into field of array of type `struct RT`. Again we index into structure so the index type is `i32`. Finally we index into two dimensional array of integers which is the same as it is in the C code fragment.

With the base knowledge of the LLVM we can now introduce proposed solution which are presented further in this thesis.

Chapter 3

Program slicing

Program slicing is a viable technique for simplifying programs by focusing on selected aspects of semantics. It was firstly introduced by Mark Weiser[11] and it was motivated by the need to help students understand and debug their programs. Nowadays, it is used to restrict the focus of a task to specific sub-components of a program. This is done by removing every statement and predicate in the program, which is not part of the interest. This process then produces a set of program statements and predicates which is called the *slice*.

From the formal point of view, the produced slice is based on the concept of slicing criterion. Slicing criterion is a pair *<p, V>* [9], where p is a program point and V is a subset of program variables. A program slice on the slicing criterion <p, V> is then defined as subset of program statements that preserves the behavior of the original program at the program point p with respect to the program variables in V. Since this slicing method, defined by Mark Weiser, preserves behaviour on every input of original program, it was named a *static slicing* to differentiate it from other slicing methods that preserve behaviour of the original program for certain subset of inputs only.

In contrast of *static slicing* [9], there are many others slicing techniques. The most known is probably the *dynamic slicing* method, which uses dynamic analysis to identify all the statements that affect the variables of interest, on the particular anomalous execution. This approach is used, besides debugging, in software testing, software maintenance and program comprehension. Other methodologies might be *quasi static slicing* which is a hybrid slicing method ranging between static and dynamic slicing. Other than that is a method derived from dynamic slicing called *simultaneous dynamic slicing*, which is similar to dynamic slicing, but slices the program against set of test cases instead of a single test case. These methods are not part of this work, but they are listed just for completeness of known slicing methods. This work focuses on implementing an automatic forward slicer. Forward slicing is a sub-methodology of static slicing, described in more detail in Section 3.2. Before describing the main slicing techniques, we introduce the Program Dependence Graph, which is the key structure of all slicing methods.

Program Dependence Graph (PDG) [1] is a program representation where nodes represent program statements and predicates, while edges carry information about control and data dependencies between the nodes.

PDG can be understood as combination of Control Flow Graph (CFG) and Data Flow Graph (DFG).

In the following, we describe the two main methods of static slicing, namely the backward and the forward slicing. These are important for understanding the solution that this paper work presents.

3.1 Backward Slicing

The most known way of program slicing is the backward static slicing. Every developer has probably met with this way of slicing even if it was not automatic nor it was producing any slice as output.

The way most of the developers probably met with this kind of slicing is because of a static analysis of their written programs. When statically analyzing a program we usually meet with a need of knowing why is the value of a certain variable in a particular point of program exactly as it is. Backtracking the analyzed code until all statements, which affected value of variable, are found, we are able to tell that these statements are product of backward slicing.

This is how a simple definition of backward slicing might be: *backward slicing* is a way of finding all program statements that might have affected the value of certain variable in a particular point of program [2]. However, the above mentioned definition is quite naive and therefore we define it more formally including all dependencies. To this, we use the Program Dependence Graph, which contains of all program dependencies. Using PDG, we can find an algorithmic way of finding dependencies on certain point of program. The PDG based algorithm considers slicing criteria of type <p, V>, where p is a program point and V is the set of variables referenced at p. A slice with respect to such a slicing criterion consists of the set of nodes that directly or indirectly affect the computation of the variables in V at node p. This formal definition of backward slicing requires the backward traversal of PDG. As oppose there is a other way of slicing, which uses a forward traversal of PDG, called the forward slicing.

3.2 Forward Slicing

Formally, a forward slice is defined as the set of program statements and predicates affected by the computation of the value of a variable v at a program point p, defined in the slicing criterion. It could represent a program comprehension of certain parts of the program. This behaviour is useful when we change a certain part of program and we want to know which parts of the program has been influenced, so we can confirm changing the program point will not cause any unsolicited behaviour. Similar principles are also used in DiffKemp.

Chapter 4

Proposed Extensions of Slicing in DiffKemp

The primary goal of this work to extend the existing approach to slicing present in DiffKemp so that it is capable to handle more cases of comparison of Linux kernel run-time and module parameters. Therefore, we first investigate the existing solution and identify its drawbacks. The main principle of the current slicing algorithm is presented in Section 4.1.

The algorithm is capable to slice a program against a global variable, which is a general way kernel represents parameters. It is based on tracking how the parameters influences as control flow of the program, as well as as data flow of the variables. This solution is satisfactory for most of the cases we compare. Unfortunately, some run-time parameters are not represented by an entire variable but by a single structure field of a variable only. For such parameters, slicing against the entire variable may be insufficient to determine semantic equality, therefore adding support for slicing again a particular field of a variable could potentially increase quality of the analysis done for the run-time parameters. A proposal for such a solution is presented in Section 4.2.

Sometimes are not compared functions equal when we slice and compare with respect to certain variable. In some cases, it can happen that when the value is set for a variable it might change the result because it will filter non-dependent code out, based on the provided value. Process how this is done is fully explained in the next section 4.3.

Moreover, the DiffKemp doesn't currently support analysis of kernel module parameters, but the tool for comparison and simplifying named SimpLL does. Therefore to these proposed solutions we add an interface for DiffKemp to work with kernel module parameters. This interface will allow running the DiffKemp tool for analyzing semantics of kernel module parameters by passing the module kernel paramters to the interface of SimpLL. To implement this, proposed solution includes similar structure as is already known from KABI functions and their `function.yaml` file which is generated in snapshot and similar approach is chosen for modules and their parameters.

4.1 Current slicing solution in DiffKemp

Current slicing algorithm takes a function f as input and global variable g as inputs. The output is the function f' that is the minimal slice of f containing all instructions that are dependent on the value of g in some way. Current solution works in the three main phases: (1) computing control and data-dependent instructions that are to be preserved,

(2) restoring the data-flow among the dependent instructions, and (3) restoring the control flow among the dependent instructions and those needed to preserve the data-flow so that the produced CFG is valid. These phases are most of the time evaluated correctly but there are some cases where evaluations might be wrong. Most of the incorrect evaluations which instructions to slice off happen in the first phase. Because of that, extensions proposed in this chapter focus on improving the evaluations in the first phase and how the first phase of the algorithm works. To understand how proposed solutions works, we need to understand how the first phase of current slicing solution works. This is described in the next Subsection 4.1.1.

4.1.1 Computing control and data-dependent instructions in current slicing algorithm

Currently integrated slicing solution for calculating dependence of the instruction has got a function and a global variable as slicing criterion. At the start of the algorithm it initialize an empty set of the dependent instructions. Then it iterates through the instructions of the function. On an every instruction it checks whether it contains the given global variable as an operand or if there is the operand which represents an instruction which is already in set of dependent instructions. This covers how data dependence is calculated, but there must be also control flow dependence computed to include all dependent instructions. Control flow dependence happens when we mark branching instruction as dependent. Branching instruction takes three arguments, result of condition based on which jump is made to first label given as second argument if the condition is true or second label given by third argument, if condition evaluates to false. Afterwards the basic blocks affected by this branching instruction are added to the set of dependent instructions.

However this solution is simple and thus it in most of the cases produces correct results, there are some shortcomings in a way of dealing with structured data types which we'll describe in the next Section 4.2.

4.2 Adding support to slicing against concrete fields of structured data types

As described in previous Section it is common case that run-time parameters are represented as field of structured data types (e.g. arrays, vectors and structures). DiffKemp currently doesn't support slicing against the fields of structured data types. DiffKemp currently slices with respect to all fields of structured type. In previous chapter it is described that when looking for dependence of an instruction we look if one of the operands is global variable. Then in Chapter 2 in Section 2.4.2 is implicitly said that access to the structured data types is done by GEP instruction which calculates final pointer which represents wanted field. As we said earlier we get global variable which is represent all the fields of structured data type. While iterating through the instructions we find instruction that uses the given global variable but it's not using the certain field which represent the run-time parameter. Even tough, the current solution marks the instruction as dependent because it does not take into account the indices that GEP instruction uses. These indices are used to calculate the final pointer that represents wanted field. On the other way it compares only the base pointer of GEP instruction which are the same but the GEP instruction does not point to the same field. This evaluation is wrong and therefore we propose solution for removing this insufficiency.

There's also question when we compare only indices for GEP instruction, if there are going to be other instruction which will suffer of this wrong evaluation mentioned above. As we mentioned in the chapter about GEP instruction, GEP instruction does not read memory, it only calculate new pointer from base address. In case of primitive data types there's no real point to calculate a new pointer because this would represent completely different object in the memory and unless there's some weird pointer arithmetic going on, we don't want that. So to work with the global variable of the primitive type we would use `load` instruction which takes as argument the identifier of global variable (named memory address where global variable is stored). In case of structured type, the base pointer is can not be load, because we can load only first class objects which are basically primitive types. This is why we use GEP instruction to access field of structured type. After GEP instruction returns the pointer which points to memory address of certain field we can load this pointer then. Because of this we cannot include check for load instruction for structured types because this can not be done and GEP instruction with primitive types because there is no meaning for that.

4.2.1 GEP Instruction Dependence

Run-time parameters are defined in certain modules of Linux kernel based on which group of the run-time parameters they fall in. In these modules there's an array of structures of run-time parameters definitions. Every definition contains the name of run-time parameter, data field, which is the pointer to data which represents the global variable representing run-time parameter, and some other fields which are not needed to be explained for purpose of this thesis. For our analysis is the most important the data field which contains the pointer to a global variable. In LLVM intermediate representation is this assignment expressed as GEP instruction in case of structured types, which contains the global variable (base pointer) and the indices that represent concrete field. Or they contain only the global variable identifier which represent the global variable of primitive data type.

This way we can gain indices for the GEP instruction and therefore we can extend our slicing criterion or an input to current slicing algorithm by these indices. Afterwards when iterating throughout the instructions of the given function we can handle GEP instruction dependence just by comparing their indices if they contain also the global variable as the second operand which represents base pointer from which computation of new pointer is made.

4.2.2 Call Instruction Dependence

Most of the accesses to a field of structured type will be represented as GEP instructions. Thanks to the SSA form of LLVM IR we can check all the uses of the GEP instruction to find all dependent instructions. But there's also one more case where this kind of behaviour isn't sufficient. Now we check whether GEP instructions contains also the right indices. But what if there's call instruction which calls the function which uses as argument given global variable without indices. Then inside the body of function it uses the GEP instruction to compute new pointer representing the field of structured data type. In this case, call instruction is dependent only if it uses exact access into to structured type as defined run-time parameter. While looking inside the function which was called by the call instruction there might occur recursion when there's not usage of exact access into structured type but we still cannot presume it's not dependant because there's also call with the global variable which holds the whole structured type, hence it's the same case with we have started at

the beginning. Solution of this problem will be standalone function which checks if the call instruction is dependent. This function iterates throughout the instructions of the called function and check if two cases occur. The first case is check whether the instruction is GEP instruction which uses exact access to structured type, i.e. global variable pointer and indices match. If they are, function ends with statement that call instruction is dependent. The second case which is more complicated, check the call instructions. This is the place where the above mentioned recursion comes in use. This function just recursively calls itself to just check if the inner functions contains GEP instruction. If the whole recursion check doesn't return anything and no dependent instructions are found, given call instruction isn't dependent.

4.2.3 Other Instructions Dependence

Other instructions should compute the dependence the same way as they are already computing because thanks to SSA form most of the instructions will be using the previous instruction which loaded the pointer from the GEP instruction calculations and as we described earlier computation whether instruction is dependent basically depends on two criterion, whether it contains the global variable or other dependent instruction as its operand. But there's some point worth to mention about the LOAD instruction. As we already mentioned in previous section there's no point for the primitive type global variable to make GEP on itself, there's also no point for structured type global variable make LOAD on themselves because even LLVM prohibits it, because LOAD instruction can be used only on **first class**, which are simply said the primitive data types. Therefore we don't need to make any additional checks when computing dependence with or without GEP instruction indices because there will be no other occurrences of LOAD instruction or GEP instruction when is one of the cases mentioned above.

4.3 Slicing w.r.t. a Concrete Value of a Parameter

Presented solution here, isn't fully capable of slicing off all the statements that aren't dependant on certain value of parameter (i.g. global variable). Instead it's series of transformations which leads to removing all the unneeded statements. First part of this solution, is getting value of parameter. This can be done by using default value which is set when variable is defined or we can define new slicing criterion with respect to value of variable. Let's say the slicing criterion *<p, V, i>*, where p, V we already know and i represents the value of variable. Because we slice against certain value of variable and not for all possible values this methodology is similar to dynamic slicing while preserving static slicing methodology. This kind of approach is similar *quasi static slicing* except there is no execution of program. Nonetheless, we already know the value of variable by simple definition above, so we don't have to use any new techniques for slicing. Next step of our series of transformation is to replace all the occurrences of variable by its constant value. This way all of the unreachable statements will be clearly recognizable, because branching of program will be conditioned by simple **constant** condition. After that, we call *Dead Code Elimination pass*[8], which is customized to detect these conditions and remove whole unreachable basic blocks and simplify CFG of the program. In the following listings is shown, how to whole process of slicing with respect to value of variable, consisting of four basic-blocks, BB1 and BB4, which aren't dependant on the variable and BB2 and BB3, which are.

```
global_variable = 0;
BB1
if(global_variable!= 0)
    BB2
else
    BB3
BB4
```

Listing 4.1: Original part of the program

```
global_variable = 0;
if(global_variable!=0)
    BB2
else
    BB3
```

Listing 4.2: After slicing w.r.t. variable

```
global_variable = 0;
if( 0 != 0 )
    BB2
else
    BB3
```

Listing 4.3: Replacing all occurrences

In the listing 4.1 is shown initial set of statements, which are going to be sliced. Next in the listing 4.2 is shown the removal of BB1 and BB4 as we mentioned they are not dependant on variable. After that, replacing value of variable in all its occurrences takes place in listing 4.3 and lastly the *dead code elimination* removes all unreachable basic-blocks. Therefore the BB3 has left.

4.4 Adding support for slicing against kernel module parameters

Component part of the previous solution is also the support for slicing against module parameters (and its values if needed). As mentioned above SimpLL tool already contains the support for slicing against run-time parameters because of run-time parameters same as module parameters are represented as the global variables in the source code. This way we only need to implement support only in the python interface of DiffKemp. This comes under two parts: generate phase and compare phase.

4.4.1 Generate phase

The Generate phase proposes solution similar for two other use cases of the DiffKemp. Analysis of KABI functions in Diffkemp and run-time parameters analysis takes as input for a generate phase the file containing all information needed for generating the snapshot. We chose this way because we wanted to maintain the way of how DiffKemp works. Now, when we know that we will use file containing information about how snapshot will be created we must decide which format to use to present the information for slicing. There are lot's of serializations format as `json, xml, yaml` which could be used as the list of the parameters with its values. However there are some requirement that must be met before choosing one.

First requirement, most important was to be really easy processed by machine. Serialization formats like json, yaml are supposed be processed easily, otherwise `xml` is bit harder when it comes to defining right structure for XML document and making XPath queries, which are used for access to certain part of the XML document. This process is complicated and also is quite limited for access to elements which must fulfil certain conditions. Therefore we chose not to use `xml` format.

Second requirement which was needed to be satisfied is the file must be easily written by hand, this means that for testing and other purposes file can be written by user in least time as possible. Again, this is where XML would be removed from considered options because of its opening and closing tags, attributes and other syntactic trash which is hard to be

written by hand. But `json` fall short too in this situation where this kind of requirement is needed. Even though `json` is much easier to be written by hand than the `xml` format, it has still some cons due to which we decided not to use this option either. Therefore `yaml` is what's left, but unfortunately we decided not to use this option too. This is because of the last requirement.

We wanted that the file could be automatically generated on the host machined for easier analysis of host system configuration. There was no tool which could directly generate `yaml` file with the configuration of the module parameters. We decided to use format of output of `sysctl -a` command which generates all sysctl paramaters in format of *key* = *value* pairs. This format was used along with all information needed for analysis of module parameters and in the results it will be:

```
<module directory>/<module name>:<parameter> [ = <value> ]
```

Module directory represents directory of the kernel where the file with the module name will be found. After that in the built module we find given parameter. Value in this format is optional and it only defines whether solution mentioned in previous section will be used. Then we generate *snapshot.yaml* file which contains all of this information for snapshot to be loaded and snapshot folder with its built modules that contains module parameters.

4.4.2 Compare phase

After the snapshot is produced we can proceed to next phase of comparison. This phase was more or less prepared for analysis against module parameters because the snapshot file already contains name of the global variable, which is supported in SimpLL tool,in contrary to sysctl/module parameter name which are not mentioned in *snapshot.yaml* file. Only support that was needed to be added was for the value to be read from snapshot and then make corresponding call to SimpLL tool to make slicing w.r.t value of variable, more explained in the previous Section 4.3.

Chapter 5

Implementation of proposed solutions

Implementation of DiffKemp consist of two parts: Python part which handles running all parts of the analysis made by DiffKemp, contains loads of module which simplifies manipulating with source code, compiling modules into LLVM IR or finding and gathering information need for analysis. The other part of DiffKemp is written in C++. This part is run in instances from DiffKemp's Python part and is in charge of preprocessing modules and simplifying them. The current implementation of DiffKemp is more specificaly described in Section 5.1

In the following section we describe implementation of proposed solutions. In Section 5.2 we describe how the solution of slicing with respect to concrete field of structured type was implemented. After that in Section 5.3 is described the slicing against the value of concrete value of a parameter and lastly in Section ?? how the support for module parameters was added.

5.1 Current implementation of the DiffKemp

From the previous chapter about DiffKemp we already know that DiffKemp analysis consists of two phases: Generate and Compare Phase. These phases are represented as two functions in DiffKemp's part written Python. In these functions there are various task which are run based on specification of the phase. In the next Subsection 5.1.1 we describe how the Generate phase is implemented and right after that we take a look at the Compare phase in Subsection 5.1.2.

5.1.1 The Generate phase

Generate phase/functions takes as input three mandatory arguments and several optional. These are kernel directory, from which the snapshot is generated snapshot, snapshot directory, function list or list of run-time parameters which are going to be analyzed and are listed line by line.

After calling the function generate with these parameters, it iterates throughout the lines of the function list and run various task based on whether function list is list of KABI functions or the list of run-time (sysctl) parameters. Then it stores serialized information extracted from generate phase into snapshot directory which was defined by second argu-

ment inside file named **snapshot.yaml** with all needed modules compiled into LLVM IR. In pseudo-code it might looked like this:

```
function generate(kernel_dir, snapshot_dir, function_list, is_sysctl=False):
    snapshot = Snapshot(kernel_dir, snapshot_dir)
    for line in function_list:
        if is_sysctl:
            Do something with sysctl
        else:
            Do something with function from KABI list
    snapshot.write_snapshot_to_file()
```

Various tasks are run based on whether function list represent the the list of KABI functions or run-time parameters. In following subsections is described the processes which must be handled in both cases.

List of KABI functions

When the function list is a list of KABI functions, generate phase takes the name of the KABI function, it searches throughout source code of the given kernel directory with utility named `cscope`. For every file with found definition of the given symbol it compiles it into LLVM IR. Now the information is stored into snapshot object which represents whole snapshot for compare phase. In pseudo-code it would look like this:

```
srcs = find_sources_with_symbol(symbol)
for src in srcs:
    module = build_module_from_source(src)
    if not module.has_function(symbol):
        continue
    snapshot.add_function(symbol, module)
```

This is simple example how generate function works in DiffKemp for KABI functions. More complicated case is explained in next section.

List of run-time parameters

In case of run-time parameters there is a bit longer flow of things which needs to be done for snapshot. Firstly, we must find module where the run-time parameter is defined. As mentioned in previous chapter for every run-time parameter group there's file which contains definition of run-time parameters. These definitions contains various fields. The most important for us there are `procname`, which represents name of run-time parameter, `data` which represents which global variable contains the value of run-time parameter and `proc_handler` which is the function which runs every time user sets up the parameter.

Now, when we have got module with run-time parameters definitions, we can parse the function list. In the introduction of this section we said that function list contains only one entry at the line, but in this case of run-time parameters there might be a pattern which can represent more than one parameter. Pattern is parsed and after that is every name of run-time parameter which can be represented by the pattern is returned. These returned names of parameters are then looped throughout for cycle and for each we perform another set of tasks.

For each symbol representing run-time parameter we first get it's `proc_handler` function if it has some. If it is, we find the module which contains definition of this function and we compile it into LLVM IR. Afterwards we just add collected information into snapshot object. Next, we extract the data variable from run-time parameters definition. These variables are represented by object `KernelParam` which holds information about data variable name and its indices from the GEP instruction. Continuing the process, we find all the sources which contains all the usages of name of variable representing the run-time parameter. Now, we compile these sources into the modules. In these modules we find again all functions using the given variable name. We add all of these functions into the snapshot object, but we skip now `proc_handler` function because that's already contained inside the snapshot object. Little peek how the implementation of this might be done can be seen in the listing below:

```
sysctl_module = get_sysctl_module(symbol)
sysctl_params_list = parse_sysctls_pattern(pattern)
for sysctl_param in sysctl_params_list:
    proc_handler_function_name = get_proc_handler(sysctl_param)
    if proc_handler_function_name:
        sysctl_module = get_module_for_symbol(sysctl_param)
        snapshot.add_function(sysctl_param, sysctl_module, glob_var=None)

    kernel_param = get_data_variable(sysctl_param)
    if not kernel_param:
        continue
    for src in find_src_using_symbol(kernel_param):
        module = build_module_from_source(src)
        for fun in get_functions_using_param(module, kernel_param):
            snapshot.add_function(
            sysctl_param, module, fun, glob_var=kernel_param.name)
```

5.1.2 The Compare phase

The Compare phase is more complicated than the generate phase, but for our solutions there's no need fully understand how its implemented. In a simple explanation, the compare phase loads all the information from *snapshot.yaml* file. Subsequently it runs SimpLL instance that does the pre-processing and module comparison for each entry inside of *snapshot.yaml* file. Afterwards in the Python part it evaluates the produced results of SimpLL and then it prints all the results and diffs into directory and for each result there's a new file containing result and diff if there's difference between two versions. Further proposed extensions of improving slicing algorithm are mostly part of the tool SimpLL. To understand how certain parts are implemented we'll describe SimpLL implementation in next subsection.

SimpLL

Main function of SimpLL contains very simple flow how analysis in this tool is made. First thing done is parsing command line arguments. LLVM provides very powerful interface for dealing with command line arguments. This interface provides template class `cl::opt`, where you create an object of this class with name of the option which will represent the

command line argument. After feeding function `cl::ParseCommandLineOptions` with arguments of the main function, it stores the value of given arguments from command line into `cl::opt` object corresponding to that option. After parsing command line options is finished, object of `Config` class is constructed based on the provided options. Construction of `Config` object does two things: converts command line options into corresponding LLVM objects (`-var` option into `llvm::GlobalVariable` class and so on) and stores all the information needed for further analysis.

Next, SimpLL runs the function `processAndCompare`. This function runs preprocessing of modules and afterwards compares the functions. For purpose of this thesis is the most important the preproccess phase which runs various LLVM passes that simplifies and removes all unnecessary code that can mess up semantic equivalency analysis afterwards. One of these passes is `VarDependencySlicer` which is modified in extension presented in the next Section 5.2 to improve overall analysis.

5.2 Slicing w.r.t certain field of structured data type representing run-time parameter

Implementation of this extension can be resolved into three parts which are divided into these part based of DiffKemp functionality which are self-sufficient so this extension doesn't affect other parts of DiffKemp functionality and also this way it's more easily tested if everything works as it should be.

5.2.1 SimpLL

The biggest part of the implementation is implemented inside part of the DiffKemp called SimpLL. When running analysis with run-time parameters SimpLL takes as input SimpLL these arguemnts: path to the first file which represents module in older version of kernel, path to the second file which represent the same module in newer version of kernel, an option `-fun` with the name of the function containing run-time parameter, an option `-var` with name of the global variable which represent the run-time parameter. Part of the new extensions of SimpLL is also new option `-index` which represents the index into GEP instruction. This option is implemented with the class of `cl::list` where a `cl` is the namespace from LLVM library which handles command line arguments. Using this class for an implementation of command line option for SimpLL means, that this option can be stacked one after another and thus provide more indices for analysis because as we already know GEP instruction can contain more than one index. Instance of class `cl::list` is vector like object. After we extract all the indices from command line, we propagate them through the flow of SimpLL until reached *VarDependencySlicer* pass. Modifications of the *VarDependencySlicer* which handles GEP instruction indices are further described in more detail in the next subsection.

5.2.2 VarDependencySlicer pass and GEP instruction dependence

VarDependencySlicer pass represent the current slicing solution presented in previous chapter. In previous Chapter 4, we also proposed solution which extends the current algorithm in *VarDependencySlicer*. This solution was presented in two main branches of implementation: the GEP instruction dependence and Call instruction dependence.

GEP dependence computation consist of comparing the given indices with indices of GEP instruction which has also operand of `GlobalVariable` class which was given as value of option `-var` and converted into the instance of this class.

Unfortunately given indices can be in various forms. As we mentioned in previous chapter we get the indices from the modules where run-time parameters are defined. These definitions contain a GEP instruction with the index into given `GlobalVariable` operand which basically a base address for computation. As we already know, every index in the GEP instruction has its type based on which type we index into. Also we know that same index with different type does not mean that the calculation that GEP does, will be the same. Unfortunately in these definitions the type into we index first (second argument of GEP instruction) is bitcasted into i8* (bitcasting means changin value into some type we choose without changing the bits) . This will completely change indices and these indices will never match with the ones that are in the modules.

Solution for this is method of `GEPInstruction` class called `accumulateConstantOffset`. This method calculates bitcasted offset based on the given data layout of module. This way we can compare the bitcasted index and calculated constant offset and tell apart dependent and independent instruction. There is also one more problem with implementation of computation of GEP dependence. This is rather clean code problem than the principal problem but solution for this isn't the simple one.

In practice there are two way how GEP instruction is placed in a code: as the standalone instruction the `GetElementPtrInstr` and as operand of some other isntruction `GEPOperand`. Implementation of comparing the indices is same for both but we can not convert `GEPOperand` into `GetElementPtrInstr`. LLVM library does this but it does not always work and if it's not than the LLVM library will convert it into `GetElementPtrConstantExpr`. This class is private for only the LLVM library so we cannot use it. Instead we used a little hack and we `dyn_cast GetElementPtrInstr` to `GEPOperand` other way around as we tried.

Second part of the solution is computation of call instruction dependence. However this solution is more complicated than GEP computation so it is described in next Section .

5.2.3 Compare phase and computation of the Call instruction dependence

Computation of the Call instruction dependence was described in previous chapter. It was implemented as it was explained in Chapter 4, however there is one implementation detail that was left out. As we iterate through the instructions when we discover the Call instruction we must first find out what function it is calling. When we retrieve the Function object which represent the called function we can iterate through the object to look inside the function and make additional check whether the function is dependent or not. Problem occurs when the function which is called isn't part of the module which is preprocessed. This is called in DiffKemp an missing definition.

When dealing with Call instruction we must differ between full dependence of call instruction and dependence only by global variable, thus pointer which represents all fields of structured data type object. In case of full dependence it is pretty obvious that function is dependent because there's no reason to pass pointer to field of structured data type which represents run-time parameter. On the other way, when function receives only the base pointer we can't decide whether the function call is dependent or not, and hence we decide based of the content of function. First thing we do is that we try to obtain Function object. This is done by method `GetCalledFunction` provided by `CallInstr` class which

represents Call instruction. When we receive the function object from method call, we can perform various operations on the object. So we make sure that function exists within the compiled module. Every `Function` object has method called `isDeclaration` which tells if the function is declaration. If it is, module doesn't contain the body of function and thus we can't iterate through, to check dependence of inner instructions and thus we can't surely tell if function is dependent. Although we can't decide of the result of analysis now, we can link the module which contains the called function and analyze it afterwards. Object `OverallResult`, which represent results of Analysis, contains field which is called **MissingDefs**. This field contains all the missing definitions of the function. This feature was mainly used during comparison phase of analysis where we must look inside to functions to tell if they're same or not. We use the `MissingDef` in this case too so we can gain all the instruction in Function body from another module.

After adding `Function` object to `MissingDefs` field, SimpLL converts this object into `yaml` representation. Python part of the DiffKemp tool reads it, and then compiles the module which contains the function in missing definitions and links it with current analysed module. Then it runs whole analysis again. Analysis ends when the results from SimpLL are produced and DiffKemp is able to evaluate them. However the function returning results isn't run when missing definitions occurred while preproccessing. When for example we can't find function's definitions while in the phase of preprocessing, SimpLL would produced no results with only missing definitions. If this would happend multiple times where function cannot by found for some reason we must secure that DiffKemp produces evaluation at least with the information is has got. This way we must handle condition where DiffKemp received no results but also were unable to find the missing definitions which received. Then we run SimpLL with information that it shouldn't return any missing definitions and try to make analysis only with the information that is present. Although this might seem unlikely that the function definition would be missing but this could be because DiffKemp couldn't compile the function definition into LLVM IR or couldn't find it using `cscope` utility. Pseudo algorithm how this would work is shown below:

```
run_analysis = True
dont_return_missing_defs_while_preprocessing = False

while run_analysis:
    run_analysis = False
    OverallResult = run_simpll(module, dont_return_missing_defs_while_preprocessing)
    if OverallResult.MissingDefs:
        try:
            for function in OverallResult.MissingDefs
                link_function_to_module(function, module)
            run_analysis = True
            continue
        except:
            if not OverallResult.result and not run_analysis:
                run_analysis = True
                dont_return_missing_defs_while_preprocessing = True

process_and_print_results(OverallResult)
```

When exception is caught that we were unable to compile and link module to our current module we check also that it doesn't contain any result. That's because we might get returned missing definition also after preprocess phase where results are already produced. So this way we won't run new analysis because without the new missing definition linked, the result would be the same. After running SimpLL, `VarDependencySlicer` pass is run, where whole implementation for computing Call instruction dependence takes place. How the implementation of this algorithm would look like is shown in next listing.

```
check_call_instruction_dependence(CallInstruction, OverallResult):
    dependent = check_if_instruction_is_fully_dependent(CallInstruction)
    if dependent:
        return True
    else:
        dependent = check_if_instruction_is_maybe_dependent(CallInstruction)
        if not dependent:
            return False
    function = getCallerFunction(CallInstruction)
    if function.isDeclaration:
        OverallResult.MissingDefs = function
        return False
    for instruction in function:
        dependent = check_if_instruction_is_fully_dependent(instruction)
        if dependent:
        # we don't need missing def when we discover that function is dependent
            del OverallResult.MissingDefs
            return True
        if True == check_call_instruction_dependence(instruction)
            return True

    return False
```

5.3 Slicing w.r.t a Concrete Value of a Parameter

Implementation of this solution is mainly contained inside the SimpLL. This extension supports three way of slicing w.r.t to a variable: slicing for all possible values therefore just using the code related to global variable, slicing against the default value of global variable and slicing with respect to concrete value of variable with corresponding type. The first way is already supported by SimpLL with command line option `-var` and argument of name of the global variable. Then the second one and third are supported due to extending format of the option `-var` argument. Now, the argument can contain not only name of the variable but also the value, which should it contain and these two are colon separated. Also it supports keyword *default* or the variable value can be empty, if we want to use default value of given global variable for analysis. Now as we mentioned in the previous chapter, we don't slice off the independent instructions based on value but we just replace them with the object representing their value in instructions which uses the loaded global variable. After the replacement we use standard program tranforamtions like Dead Code Elimination to slice off all independent instructions. This process is done in `VarValueDependencySlicer` which is described in next section 5.3.1.

5.3.1 VarValueDependencySlicer

VarValueDependencySlicer is a LLVM pass which inherits from the ModulePass class. This means that transformation which is done by LLVM pass, is performed above whole LLVM module which was sent into SimpLL as input. This is because it's faster and more simple to iterate through the usages of GlobalVariable class in LLVM than it would be to iterate through the instruction of the function. While iterating through instructions we would had to also iterate throughout its operands and check if any of them is global variable which we are looking for. Reasoning behind this is A/B testing that the general module has more instructions in function than the module has usages of certain global variable. On the other hand flow of this LLVM pass is very simple. In the beginning, takes as arguments module, global variable and the value which is to be replaced for every occurrence of the given global variable. This value can be also `null` which means that default value should be used for replacement. Using default value it's easier for us because every global variable object has function `getInitializer` which returns value which has been set when global variable was initialized. This way we get right instance of `Constant` class which is used later for replacement, because the function used for replacement uses `Constant` class type as argument.

Further, it iterates all usages of global variable as we mentioned previously. When usage of the Global Variable is found we try to `dyn_cast` it into load instruction. Anyway after we find out that current instruction is load instruction we replace all is usages again with the `Constant` object of corresponding type. This is done by the function which provides LLVM interface of `Value` class called `replaceAllUsesWith` which takes as argument an object containing value of variable that we want to replace all uses of load instruction with. However the current flow of this implementation is quite simple there's a tricky part when converting string argument from command line option into corresponding `Constant` object type.

This part is done with `dyn_casting` global variable initializer into various specific `Constant` object types (e.g. `ConstantInt` or `ConstantFP` classes). When we find out which type of specific class it is, we call it's constructor with string argument. Constructor then returns an object of specific `Constant` type and we return it from the function. However the function return the Constant class not the specific sub-classes but due to inheritance from `Constant` object the casting to superclass is done automatically by Clang compiler.

5.4 Supporting slicing against kernel module parameters in python interface

In this section we shortly introduce how the interface for dealing with kernel module parameters was implemented into the current generate phase. As we mentioned in previous chapter this part is needed to be implement only for snapshot generation because SimpLL already support it in its interface the global variables and global variables are representing as the run-time parameters so do they the module parameters. As we know from the previous section there is loop which takes lines from the function list. In this loop there's if condition so we can make different flow whether we analyse run-time parameters or KABI functions. This way we have added the third branch of the if condition which takes care of module-parameters. First we parse the entry from function list based on the format we introduced in previous chapter. After that we find module and compile it into LLVM IR

from the given module directory and module name. Then it finds global variable which represent module parameter. Rest of the flow is same then as it is for run-time parameters when we find all functions using that global variable in all the possible modules. Then we add the found functions into the snapshot.

Chapter 6

Experimenting with implemented solutions

The goal of the experiments of the first two implemented solutions is to prove that results of analysis made by DiffKemp were improved after implementing these solution and integrating them into the current solution of the DiffKemp. For the last solution implemented, we show how can we compare many module parameters in such a easy way just by creating list of them. In the next Section 6.1 we will show what results experiments have returned while testing the first extension which is responsible for slicing against concrete field of structured types. After that, in Section 6.2 we show how we can make an analysis of kernel modules while using the second extension which performs analysis with respect to a concrete value of global variable.

6.1 Slicing against concrete fields of structured types experiments

Experiments of testing functionality of this extension have been tested on CentOs kernel version 7.3 and 7.4. We've generated snapshots containing indices of global variables representing run-time parameters and tested list of run-time parameters presented in Table 6.1 with results they have returned with or without the extension.

After running analysis with extension we can see that no results have turned from *Equal* state into *Not Equal*. Also no *Errors* occurred during analysis which means that implemented solution did not mess up analysis made by DiffKemp. Sadly in the tested list of run-time parameters has occurred only one conversion from *Not Equal* state to *Equal*. After analyzing given case by hand we could confirm that implemented solution has really done what it was supposed to do. This low number of successful cases after adding our solution might be because most of the cases we're evaluated as *Equal* for the set of instruction that covered whole global variable not only the field we wanted. After that subset should be evaluated equally as the given set and therefore still it evaluates to *Equal*.

To conclude this, we have implemented solution that works for slicing with respect to concrete field of variable, however there wasn't much cases as we expected to improve whole analysis performed by DiffKemp. In the future work we aim for verifying if the given solution didn't work for other cases because of some nuances or if there wasn't just that many cases to improve analysis as we firstly assumed.

Run-time parameter	Function	Results	
		w/o extension	with extension
net.core.rmem_defaul	sock_init_data	Equal	Equal
net.core.rmem_max	sock_setsockopt	Not Equal	Not Equal
	tcp_select_initial_window	Equal	Equal
	set_sock_size	Equal	Equal
net.core.wmem_default	ip_send_unicast_reply	Equal	Equal
	sock_init_data	Equal	Equal
net.core.wmem_max	sock_setsockopt	Not Equal	Not Equal
	set_sock_size	Equal	Equal
net.ipv4.conf.all.forwarding	devinet_init_net	Equal	Equal
	_devinet_sysctl_register	Equal	Equal
net.ipv4.conf.all.rp_filter	devinet_init_net	Equal	Equal
	_devinet_sysctl_register	Equal	Equal
net.ipv4.tcp_tw_recycle	tcp_v4_rcv	Equal	Equal
	tcp_v4_connect	Not Equal	Not Equal
	tcp_sk_exit_batch	Equal	Equal
	tcp_time_wait	Not Equal	Equal

Table 6.1: Results of analysis between 7.3 and 7.4 CentOs kernel

6.2 Slicing w.r.t to concrete value of global variable experiments on the kernel modules

This experiments will be performed on two upstream versions of the Linux kernel, on **3.10** and **4.11**. We have decided to use kernel modules in which we have found that they are unequal, but could be equal for certain values. Subsequently we generated snapshots for both version of Linux Kernel and for each we also generated snapshot which uses values and which is not. After running the experiments we have got results as shown in Table 6.2.

As we can see from Table 6.2, we have performed analysis on the kernel module parameters. We implied earlier that adding support to analysis of kernel module parameter will make it easier and faster to analyse difference in kernel modules. From the reported statistics from DiffKemp while comparing the two versions of kernel it returned that DiffKemp compared 48 functions. This is quite a lot for such a small number of tested parameters. However this analysis took maximum of 10 minutes with the time of generating snapshots for analysis and subsequent comparison of these. If we would wanted find out if semantics of 48 functions is equal or not by a hand, it would take much more effort and time, it could take possibly time of more than several hours rather than 10 minutes which lasted the analysis made by DiffKemp.

Also in Table 6.2, we have shown impact of implementing extension which slices w.r.t. certain value of global variable which represents the module parameter. As we can see most of the cases has resulted into *Not Equal*. After implementing the solution some cases has turned into the *Equal* result. This could be caused by removing the code that wasn't dependent of the value of variable and also the code that was marked as dependent didn't contain non-equal code. After analyzing the results, which produced *Equal* result and was previously marked as *Not Equal*, we could confirm that these results are correct. As we can see the results marked as *Equal* without using the extension, retained the result given

Module	**Parameter**	**Function**	**Value**	**Results of analysis**	
				w/o value	**with value**
nf_nat_snmp_basic	debug	snmp_parse_mangle	default	Not Equal	Not Equal
		mangle_address	default	Equal	Equal
drbd	fault_devs	_drbd_insert_fault	default	Equal	Equal
tcp_probe	bufsize	tcpprobe_init	0	Not Equal	Equal
		tcpprobe_avail	0	Equal	Equal
		tcpprobe_read	0	Equal	Equal
		tcp_probe_used	0	Equal	Equal
nbd	max_part	nbd_init	-1	Not Equal	Equal
ipmi_watchdog	ifnum_to_use	set_param_wdog_ifnum	default	Not Equal	Equal
		ipmi_register_watchdog	default	Not Equal	Not Equal
lp	reset	lp_register	default	Not Equal	Equal
applicom	mem	applicom_init	default	Not Equal	Not Equal
		ac_ioctl	default	Equal	Equal

Table 6.2: Comparison of kernel modules parameters from upstream versions 3.10 and 4.11 with results of analysis with extension, which adds value of parameter to slicing criterion

without extension and was marked it as *Equal* too, because when set of instructions was marked as *Equal* at first, then the subset of this set must also be marked as *Equal*.

To conclude this, implemented solution really helped the analysis made by DiffKemp and also it dramatically made it easier to analyse the kernel module parameters.

Chapter 7

Conclusion

The goal of this thesis was to propose solution for automatic forward slicing of Linux kernel modules. Subsequently we implemented this proposed solutions and we made the comparison between two kernel versions for experimentally prove that implemented solutions improved analysis made by DiffKemp.

At the beginning we have described a static analysis tool DiffKemp with LLVM framework which was used to implement an automatic forward slicer for DiffKemp. Automatic forward slicer aims for simplifying analysis of semantic differences between two version of Linux kernel. The purpose of this approach is to remove all statements that aren't dependant on certain function or parameter and thus make it easier for semantic analyzer to compare certain semantic of analyzed source. DiffKemp has already contained automatic forward slicing solution and after extensive analysis and testing we decided to extend its current functionality by few most solutions that should be most efficient based on our assumptions. We have implemented slicing against the fields of structured types which removes considerable chunk of independent code. Also we extended current slicing criterion with the optional value parameter for the global value which represent run-time or module parameters. And as we mentioned we added also the support for slicing with respect to module parameters which also makes it easier to analyze, because number of functions containing the module parameters that are analyzed is quite extensive. After running the experiments we noticed that first implemented solution didn't improved the analysis made by DiffKemp, which could be potentially caused by DiffKemp returning Equal result for set of instruction and therefore subset created by implementing solution from this thesis should return same results. Otherwise experiments with solution providing support for kernel module parameters has made easier analysis and much more faster than it would by done by a hand. Also last extension which adds value of global variable into slicing criterion has also improved analysis done by DiffKemp in a quite big scope.

We implemented three extensions for DiffKemp to improve analysis it performs but there are still some extension which can be done to improve automatic forward slicing solution which DiffKemp uses. This extension could be for example taking into account pointer aliasing when slicing off the statements which might alias with dependent instructions. Also it's good idea to provide support for slicing w.r.t certain value with string type and accesses to structured types. We planned to extend the current solution later in the future because this would be quite extensive amount of work but it wouldn't cover much cases because there are less string-typed run-time parameters than there are other types.

Bibliography

[1] Ferrante, J., Ottenstein, K. and Warren, J. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*. july 1987, vol. 9, p. 319–349. DOI: 10.1145/24039.24041.

[2] Harman, M. and Hierons, R. An Overview of Program Slicing. *Software Focus*. december 2001, vol. 2. DOI: 10.1002/swf.41.

[3] Ing. Viktor Malík. *DevConf - DiffKemp* [online]. 2019. Last updated on 2019-08-16 [cit. 28-07-2020]. Available at: https://sched.co/RknV.

[4] Lattner, C. Introduction to LLVM. *LLVM* [online]. 2012 Jul 07. Available at: http://www.aosabook.org/en/llvm.html.

[5] LLVM Project. LLVM Language Reference Manual. *LLVM* [online]. 2020. Last updated on 2020-01-23 [cit. 24. January 2020]. Available at: https://llvm.org/docs/LangRef.htm.

[6] LLVM Project. LLVM Compiler Infrastructure. *LLVM* [online]. 2020. Last updated on 2020-03-24 [cit. 25-07-2020]. Available at: https://llvm.org/.

[7] LLVM Project. The Often Misunderstood GEP Instruction. *LLVM* [online]. 2020. Last updated on 2020-07-28 [cit. 28-07-2020]. Available at: https://llvm.org/docs/GetElementPtr.html.

[8] LLVM Project. LLVM's Analysis and Transform Passes. *LLVM* [online]. 2020. Last updated on 2020-01-23 [cit. 24. January 2020]. Available at: https://llvm.org/docs/Passes.html.

[9] Lucia, A. Program slicing: methods and applications. In:. February 2001, p. 142 – 149. DOI: 10.1109/SCAM.2001.972675. ISBN 0-7695-1387-5.

[10] Malík, V. DiffKemp. *GitHub* [online]. 2019 Jul 10. Edited 16 Oct 2019 [cit. 24. January 2020]. Available at: https://github.com/viktormalik/diffkemp.

[11] Weiser, M. D. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. USA, 1979. Dissertation. University of Michigan. AAI8007856.