CHANGE-REGION DETECTION IN LLVM

FLORIAN NIEDERHUBER

MASTER THESIS

Chair of Software Engineering
Faculty of Computer Science and Mathematics
University of Passau

Advisor: Florian Sattler, M.Sc.

Supervisor: Prof. Dr.-Ing. Sven Apel

2nd corrector: Prof. Christian Lengauer, Ph.D.

February 2018

Florian Niederhuber: *Change-Region Detection in LLVM* , © February 2018

ABSTRACT

Code changes are an essential part in the life cycle of a software, whether the program is still under development or already shipped to the customer. Regardless whether such a modification introduces a new feature, prevent errors, or solve compatibility issues, each change can have a major impact on the rest of the software (e.g., functionality, performance, ...). Depending on the level of experience of the developer performing the change, the developer can notice or miss the potential impact on the behavior of the program. This may lead to unexpected problems for users and other developers.

In order to support the developer, we aim to provide a way to analyze changes in a software repository, completely transparent to the developer and, more importantly, in a language independent manner. We propose a tool-chain to identify, organize, and analyze changes, based on the meta-data extracted using Git. To this end, we annotate the source-code of the software and use these annotations during the compilation process. Furthermore, we provide ideas and an actual implementation how to use the annotated regions for analyses and describe how we integrated our approach in the Variablility-aware Region Analyzer (VaRA) and in the compiler framework LLVM.

We used our analyses to infer relations among instructions to detect control-flow and data-flow interactions among commits for a handwritten example and the open-source software GNU zip (gzip). Moreover, we discuss the results of the evaluation and present our findings.

ACKNOWLEDGMENTS

I would first like to thank my family and my girlfriend who have always believed in me and encouraged me during my years of study. Furthermore, I would like to thank my friends for having an open ear when I needed one. Especially, I would like to thank my advisor Florian Sattler who consistently allowed this paper to be my own work, but steered me in the right direction. Another special thanks goes to Stefan Löwe who provided me with very valuable comments on this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

SC-R	source-code region
IR-R	IR-meta region
LLVM-IR	LLVM intermediate representation

AST	abstract-syntax-tree
RCS	revision control system
VARA	Variablility-aware Region Analyzer
SHA	Secure Hash Algorithm
SPEDI	Static Patch Extraction and Dynamic Insertion
CFG	control-flow graph
RISC	Reduced Instruction Set Computer
SSA	static-single-assignment
JIT	just-in-time
WLLVM	Whole Program LLVM
SVG	Scalable Vector Graphics
GZIP	GNU zip
YAML	YAML Ain't Markup Language

1 INTRODUCTION

In software development a lot of projects suffer from the fact that they are getting continuously harder to maintain: One reason for this is the growing size and complexity of these projects, another reason is that software requirements change over time to accommodate the expectations of the user [2, p. 4]. Furthermore, software may evolve over several years before they even considered stable enough for production and few people working on such projects stay until it is ready for production, and even fewer remember the initial requirements or the rationale behind them [2, pp. 4-5]. Lastly, most software projects are changed even if they are already in production. ISO/IEC standard [4] identifies four categories of changes occurring during maintenance in the life-time of a software:

1. Corrective maintenance, changes resolving errors in the software.
2. Preventive maintenance, changes necessary to prevent and detect potential errors in the software.
3. Adaptive maintenance, changes that are necessary to support new environments.
4. Perfective maintenance, changes enhancing the performance and the usability of the software.

Changes are a fundamental part in the life-cycle of software programs. However, modifying single parts of a software may have unexpected impact on regions not targeted by a certain modification. We believe that identifying the affected regions in the source code and providing it to the compiler enables analyses to reason about changes and their impact. Thus, allowing not only software researchers to extract the information but also the creation of mitigation tactics that prevent errors or warn the developer.

1.1 GOAL

Our main goal is to find interactions of changes in a software project and determine their impact in order to aid software developer in the maintenance process, by providing a way to analyze changes and present the affected parts of the code to the developer. Furthermore, the region detection and analyses are built upon and integrated in VaRA in order to allow them to be used in a language independent manner. However, in this thesis we aim on an implementation in clang to support the programming languages C and C++.

1.2 CONTRIBUTION

In this thesis we introduce a way to annotate source code based on software changes, gathered from a revision control system (RCS). We further provide an implementation that is able to utilize the annotations and perform analyses on it. Furthermore, we present an analysis that determines the impact of a certain change on the rest of the software.

1.3 OVERVIEW

This thesis is organized as follows. In Chapter 2, we introduce the basic concepts of the methodologies, frameworks, and software used throughout the thesis. In Chapter 3, we present how change regions are annotated, utilized by clang, and provided to the LLVM optimizer. Chapter 4, describes how the region information is used to determine correlations between different code regions and we present the results and insights generated. Finally, in Chapter 5, we draw a conclusion from our work with a final statement of the author, suggestions to further improve our approach, and discuss similar efforts made by other researchers.

2 BACKGROUND

In the following chapter, we give information regarding the technologies, theories, and software used in this thesis. First, we introduce the general methodologies of control-flow and data-flow analyses. Second, we introduce the compiler framework LLVM including the front-end clang in Section 2.2. Finally, we describe the required components of the RCS Git.

2.1 CONTROL-FLOW GRAPH AND DATA-FLOW ANALYSIS

In the following section, we introduce the concept of a control-flow graph, a graph representation of the program flow. Furthermore, we describe the basic concept of a data-flow analysis.

EXECUTION PATH Regarding to Aho et al.[1] the execution of a program can be seen as a chronological sequence of transformations to the program state, which represents the values of all variables at a given point in the program's execution. Each transformation converts its input state into an an output state. An execution path then represents a chain of transformations, where the output state of transformation T_i is the input state of the succeeding transformation T_{i+1}. Furthermore, the execution path may contain instructions that change the flow of control, as a result a program can have an infinite number of possible execution paths.

CONTROL-FLOW IN BASIC BLOCKS A program is organized in basic blocks. Each of theses blocks contain instructions, whereby none of them change the flow of control of the application, except the one terminating the block. Therefore, the execution path in a basic block can be denoted as follows: Every instruction $i_1, i_2, ..., i_n$ contained in a basic block is executed exactly in that order.

2.1.1 *Control-Flow Graph*

The control-flow graph (CFG) represents all possible execution paths through a program. Since the flow of control in basic blocks is straight forward the CFG describes only the relation between basic blocks. The example depicted in Figure 1 contains the CFG of an example program described as follows. At the start of the program the variable a is defined and initialized in basic block one denoted with BB_1. On the end of BB_1 the program jumps to the beginning of the next basic block

BB_2. The program requests a character from the user and decides, based on the value, of the character whether to jump to BB_3 or directly to the end of the program in BB_4. Since both execution paths are possible, we assume the condition is fulfilled and the control flow is directed towards the basic block containing the while body BB_3. In BB_3 the previously defined variable a is increased by one and the program jumps back to the while condition in BB_2. If the user input does not fulfill the condition the program jumps to BB_4 printing the current value of a to the console and the program exits.

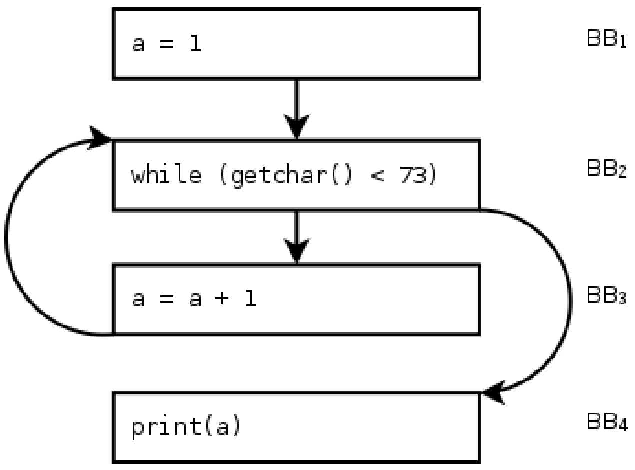

Figure 1: Control-flow graph of an example program.

2.1.2 *Data-Flow Analysis*

Data-flow analysis is a generic term for techniques that derive information about the flow of data along the program execution path [1]. In this section, we introduce the general concepts enabling these techniques.

TRANSFER FUNCTION The transfer function takes an input state and uses it to produce, according to the constraints given by the instructions semantics, a new output state. Based on whether the data flow is propagated forward [7] or backward [3] the state before an instruction is used to generate a new value after an instruction, or the state after an instruction is used to produce a new before value, respectively. We denote the transfer function for an instruction i with f_i.

IN AND OUT SETS The values of the states before and after instructions along the execution path are stored in an IN and an OUT set, whereby the IN set refers to the value of the state before an instruction, the OUT set refers to the value of the state after an instruction. The IN and OUT set for an instruction i is denoted with IN[i] and $\mathrm{OUT}[i]$. Lets assume we have a program with a single execution path $i_1, i_2, ... i_n$. The IN set for the first instruction i_1 is the empty set $\mathrm{IN}[i_1] = \emptyset$. The transfer function transforms the input set $\mathrm{IN}[i_1]$ to the output set $\mathrm{OUT}[i_1]$ for instruction i_1. The next instruction i_2 uses the output state from the first instruction as its input $\mathrm{IN}[i_2] = \mathrm{OUT}[i_1]$ and transforms it to the output set $\mathrm{OUT}[i_2]$ by using the transform function for instruction i_2. We can generalize this for all instructions in the program, for $j = 1, 2, ..., n-1$:

$$\mathrm{IN}[i_{j+1}] = \mathrm{OUT}[i_j]$$

$$\mathrm{OUT}[j+1] = f_{i+1}(\mathrm{IN}[i_{j+1}])$$

IN AND OUT OF A BASIC BLOCK Since the execution path of a basic block has no control-flow changes, we can generalize the IN and OUT set such that it applies to basic blocks. The OUT set of a basic block OUT[BB] is determined by a series of transformations based on the set IN[BB]. However, the control flow between basic blocks can be influenced by certain instructions, as a result the program contains a number of execution paths. Since we perform a static analysis we have to consider all of these paths. Base on whether we want to follow the data flow in forward or backward direction, we have to take all of the basic blocks predecessor or successor states into account. In order to join the resulting sets to a single one we need to define a specific meet operator $\bigwedge$, that is able to join the information.

In terms of a forward data flow the IN set for a basic block BB is determined by the OUT sets of all its direct predecessors P:

$$\mathrm{IN}[BB] = \bigwedge_{p \in P} \mathrm{OUT}[p]$$

If the data-flow analysis is done backwards the OUT set for a basic block BB is determined by the IN sets of all its direct successors S:

$$\mathrm{OUT}[BB] = \bigwedge_{s \in S} \mathrm{IN}[s]$$

SPECIFIC ANALYSES These concepts can be used for a specific analyses that are based on the data-flow of a program, e.g., reaching definitions and live variable analysis. Depending on their use case they either follow the control-flow path forward or backward. The transfer function and the meet operator are then used to specify which and how the information is propagated.

2.2 LLVM COMPILER FRAMEWORK

LLVM is a compiler framework designed to support life long program analysis and transformations to a software program and still provide transparency to their programmers [12]. Today LLVM is a collection of sub-projects introducing programming languages and bring support for various platforms and tools condensed in a single framework[1]. In the following section we introduce the architecture and important components of LLVM.

2.2.1 *Architecture*

The LLVM framework has a three-layer architecture depicted in Figure 2. The design consists of a front-end (left), an optimizer (middle), and a back-end (right).

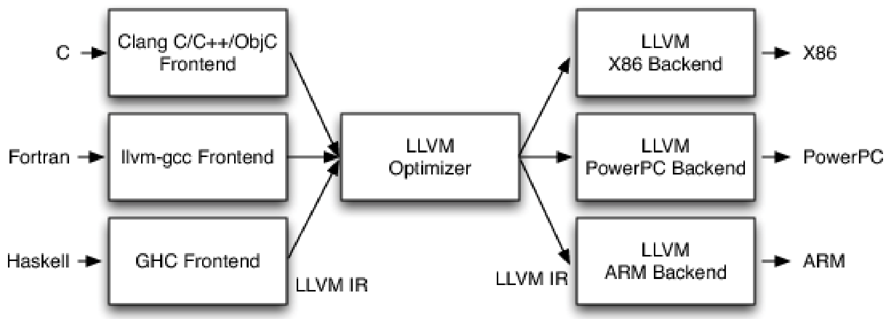

Figure 2: Three-layer architecture of LLVM [11].

FRONT-END A front-end supports a high-level programming language, e.g., C, C++, or Haskell. It is responsible for parsing, validating, and diagnosing errors in the input files [11]. Furthermore, the front-end may apply language-specific optimizations to the program before it generates and passes the LLVM-IR code on to the optimizer.

OPTIMIZER LLVM offers modules, so called passes, that can be used to analyze and alter the code of a software, as well as to create reports based on the gathered insights. They are categorized into three groups: analysis pass, transform pass, and utility pass[2]. Since every front-end translates its input to LLVM-IR the optimization passes can be applied to every piece of code, no matter the original input language. The language-independent optimizations are one of the greatest benefits of LLVM, since they only need to be written once in order to enhance numerous programs.

BACK-END The improved code from the optimizer is used by the back-end whose main purpose is to generate native machine code and perform architecture-dependent optimizations. Due to the mod-

1 *The LLVM Compiler Infrastructure Project.* `https://llvm.org/`. (Accessed on 10/02/2017)

2 *LLVM's Analysis and Transform Passes — LLVM 6 documentation.* `https://llvm.org/docs/Passes.html`. (Accessed on 10/02/2017)

ular three-layer design the program can be deployed to numerous platforms, providing a LLVM back-end, using the same input from the optimizer.

MODULARITY LLVM is basically a set of libraries that can be used to create a compiler, a virtual machine, or tools to analyze and optimize programs built with the framework. For example the static-value-flow analysis tool SVF [16], the Static Patch Extraction and Dynamic Insertion (SPEDI) [5], or the Variablility-aware Region Analyzer (VaRA) [15], which is described in more detail in Section 2.2.5. Furthermore, the modularized subsystem allows to easily extend the optimizer with new passes by loading them from external libraries.

2.2.2 *LLVM Intermediate Representation*

In the following section, we introduce the LLVM intermediate representation (LLVM-IR), its properties, and some of the design decisions made by the inventors. A detailed definition of the syntax and semantics of LLVM-IR is provided by the LLVM reference manual[3].

DESIGN LLVM-IR is designed to serve as a platform and language-independent representation of computer programs, allowing the compiler to perform transformations and analyses upon them. The lightweight and low-level design with an instruction set similar to a Reduced Instruction Set Computer (RISC) offers a simple type system with strongly-typed instructions. Furthermore, it provides an infinite set of registers in static-single-assignment (SSA) form. While not all language and platform-specific features are represented by LLVM-IR directly, it allows the front-end to map the high-level features to it and allows the back-end to derive platform specific instructions from it to use all components form the target platform. LLVM utilizes the code representation in three different forms: (a) As a representation in the compiler (b) in bitcode form, e.g., for just-in-time (JIT)-compiler, (c) and to be a human-readable assembly-language representation [11, 12].

EXAMPLE PROGRAM The example provided in Listing 1 is a program expressed in LLVM-IR writing the string `"Hello World!"` to the standard output stream. In LLVM, such a program as shown in the example is called a module. In the first line of the module we define a constant string in global scope with the value `"Hello World!"`. As mentioned earlier LLVM-IR preserves types and the type of this string is a sequence of 13 8-bit integers. In Line 3 we declare the external function `printf`, which is needed to print values to the standard out-

3 *LLVM Language Reference Manual — LLVM 6 documentation.* `https://llvm.org/docs/LangRef.html`. (Accessed on 10/08/2017)

put stream. The definition of the `main` function starts in Line 5. The function has no arguments and returns a 32-bit integer. In the body we first calculate a pointer to the string `@hello` stored in the variable `%s`. The first argument we define is the base type used for offset calculations, the second one is the base address of the string, the third and forth argument indexes the first character in the string. Now we can use the pointer to the character sequence to call instruction that prints our string. The `printf` function in Line 7 is executed using the `call` instruction, which, in our case, takes three arguments: **`i32`** is the return type a 32-bit integer, `(i8, ...)` is the signature of the function, and the last argument `@printf(i8* %s)` is the function pointer to `printf` followed by its parameters, the pointer variable pointing to the `"Hello World!"` string. In the last line of the function body we return a zero, the first parameter of the `ret` instruction is the return type and the second one the value.

```
1  @hello = constant [13 x i8] c"Hello wordl!\00"
2
3  declare i32 @printf(i8*, ...)
4
5  define i32 @main() {
6          %s = getelementptr [13 x i8], [13 x i8]* @hello,
            ↪  i32 0, i32 0
7          call i32 (i8*, ...) @printf(i8* %s)
8          ret i32 0
9  }
```

Listing 1: "Hello World" example in LLVM-IR.

METADATA LLVM offers a way to annotate instructions with custom information via metadata annotations. This additional information on the program can be utilized by the optimizer with only little effort. Each of these annotations is grouped in a so-called metadata-group and each record is identified by an incrementing number, starting with zero. We extend the example from Listing 1 with metadata, by adding two custom metadata-groups: (a) CustomInfo and (b) ExternInfo and three distinct metadata nodes. The updated example is depicted in Listing 2. The distinct nodes we added are appended to the end of the file, see Lines 11 to 13. We now are able to reference these nodes by appending the group and the ID after an instruction. We are also able to reuse the ID to annotate different instructions with the same metadata, see Line 6 and 7. The optimizer is now able to request the annotations for a given group and can take measures based on them.

```
@hello = constant [13 x i8] c"Hello wordl!\00",
  ↪ !CustomInfo !1

declare i32 @printf(i8*, ...), !ExternInfo !2

define i32 @main() {
        %s = getelementptr [13 x i8], [13 x i8]* @hello,
         ↪ i32 0, i32 0, !CustomInfo !3
        call i32 (i8*, ...) @printf(i8* %s), !CustomInfo
         ↪ !3
        ret i32 0
}

!0 = !{!"Contains a string."}
!1 = !{!"libc"}
!3 = !{!"Used to output string."}
```

Listing 2: "Hello World" example in LLVM-IR with meta-data annotations.

2.2.3 *Compiler Front-End clang*

In the following section, we introduce clang, a front-end to support C and C++ for LLVM. The front-end aims for high performance and efficiency, it also offers expressive diagnostics and is rather easy to extend with new features[4].

SOURCE LOCATION The source location is used by clang to determine the line and column of a certain token. This feature is mainly used to generate more meaningful diagnostic reports. The location tracking relies on two components the `clang::SourceManager` and the `clang::SourceLocation`. The `clang::SourceManager` is created only once for every translation unit and manages the loading and caching of source files in general. The `clang::SourceLocation` on the other hand is attached to every token, in order to avoid a performance loss all the necessary information, like source file ID and the relative location in the code, is encoded in a 32 bit sequence. In combination with the information stored in the `clang::SourceManager` the exact position of the instruction in the source file can be calculated.

ABSTRACT-SYNTAX-TREE An AST represents the source code of a program in tree form, whereby the structure of the program is preserved and all unnecessary information is omitted [14, Section 3.17.2]. The clang AST specifically preserves, compared to those used in sim-

4 *Clang - Features and Goals*. `https://clang.llvm.org/features.html`. (Accessed on 10/11/2017)

ilar compilers, more information, e.g., source location and compile-time constants[5]. In clang the `clang::ASTContext` class manages all the information related to the AST, available during a translation of a module. Every node in the tree is expressed through either a `clang::Type` representing a single base type, a `clang::Decl` representing a single declaration or definition, a `clang::DeclContext` representing declaration types with context information, a `clang::Stmt` representing a single statement, or through a node class derived from one of these four classes. The corresponding AST for the following example shown in Listing 3 is depicted in Listing 4.

```c
        char s[] = "Hello World!";
```

Listing 3: Example C code.

We cropped internal clang declarations, marked with the three dots, for the sake of simplicity. The following node is a variable declaration of the 13 character array **`char`** `s[]`. The declaration has one child `clang::StringLiteral`, that is based on the `clang::Stmt`.

```
...
‘-VarDecl 0x55ccdd455248 <example.c:1:1, col:12> col:6 s
 ↪  'char [13]' cinit
  ‘-StringLiteral 0x55ccdd4aae30 <col:12> 'char [13]'
   ↪  lvalue "Hello World!"
```

Listing 4: Example code in AST form.

LEXING AND PARSING As one of the first steps after clang loaded the source code, is to interpret the character encoded parts of the program. The process is called tokenization and is performed by the lexer component. The lexer splits the source code, from the example program shown in Listing 3, in the following token:

(1) **`char`**
(2) `s`
(3) `[`
(4) `]`
(5) `=`
(6) `"Hello World!"`
(7) `;`

Each token is then passed on to the parser, which decides how to act on each and every token. In clang a recursive-descent parser is used, which uses in clang a top-down direction and a depth-first strategy. We do not describe recursive-descent parsing any further, more detailed information can be found in the compiler-design reading [14,

5 *Introduction to the Clang AST — Clang 6 documentation*. `https://clang.llvm.org/docs/IntroductionToTheClangAST.html`. (Accessed on 10/12/2017)

Section 3.12]. The parser starts with the first token and may request additional ones based on the underlying grammar. The grammar is used to validate the input source code whether it is well-formed or not. In order to perform these validations, clang has a semantic analysis component. If the input code is not well-formed the analysis reports the issues back to the developer, otherwise the program is translated into an AST.

ADD A NEW KEYWORD In order to introduce a new keyword to clang we need to adapt the lexer, the parser, and the semantic engine components so that they are able to handle the instruction properly. The first step in introducing a new keyword is to add a token-kind definition. These definitions are required so that the lexer is able to translate the character sequence into a token. To add a new token kind we need to adapt the *include/clang/Basic/TokenKinds.def* file in the clang project. The example depicted in Listing 5 introduces a new token named `example`. The `KEYALL` attribute defines in which C-family the keyword is available. In our case the token is recognized by all variations clang has to offer.

```
KEYWORD(example , KEYALL)
```

Listing 5: Example of a new keyword introduced to clang.

The lexer is now able to translate the token. In the next step we need to introduce the keyword to the parser and define its behavior on the occurrence of the token. Usually keywords expect some kind of successor token, like the type declarator **`int`** expects an identifier. We made sure that the syntax is correct in the parser component, now we need to make sure the semantic of the keyword fits our needs. In the last step we have to create an AST representation of our newly-created keyword and include it in the AST.

CODE GENERATION Lastly, clang uses the AST to generate LLVM-IR code. The code produced looks slightly different from the one we wrote by hand in Section 2.2.2. The example shown in Listing 6 contains on the left side a program written in C and on the right side the corresponding instructions in LLVM-IR code. The output is produced with clang in version *6.0.0*. In the example we start with the declaration and initialization of the global variable `a` and continue with the declaration of the function `foo` in Line 5, declaring variable `b` and `result`. We initialize `b` and add the global `a` and the local variable `b` together in Line 12. The result of the calculation is stored in the variable `res`. In Line 15, we exit from the function `foo` and return the value of the variable `res`.

We cropped the arguments and metadata generated by clang to focus on the translation of the code parts. In the first line, we declare and initialize the global variable `@a`, notice the explicit alignment set by clang, affecting the alignment of address to the value. In Line 5, clang defined the main function and assigned the set of attributes introduced in Line 18. These attributes are used to enhance the optimization process of the function. The following two lines in the function body allocate space for the variables `b` and `res` and the address is stored in a virtual register. The variable `b` is initialized in Line 9. The two required variables for the addition are loaded in a virtual register and added up in Line 12. The result of the addition is stored in the local variable `%res`. The last two lines in the function body load the result from memory and return the value.

```c
1
2  int a = 31;
3
4
5  int foo() {
6
7    int b;
8    int result;
9    b = 42;
10
11
12   res = a + b;
13
14
15   return res;
16 }
17
18
19
```

```llvm
1  ...
2  @a = global i32 31, align 4
3
4  ; Function Attrs: noinline ...
5  define i32 @foo() #0 {
6  entry:
7    %b = alloca i32, align 4
8    %res = alloca i32, align 4
9    store i32 42, i32* %b, align 4
10   %0 = load i32, i32* @a, align 4
11   %1 = load i32, i32* %b, align 4
12   %add = add nsw i32 %0, %1
13   store i32 %add, i32* %res, align 4
14   %2 = load i32, i32* %res, align 4
15   ret i32 %2
16 }
17
18 attributes #0 = { ... }
19 ...
```

Listing 6: C source-code and corresponding LLVM-IR Example.

2.2.4 *Optimization Pass*

In this section, we introduce the different types of LLVM passes, their capabilities, and how to write a certain optimization pass.

CATEGORIES In LLVM the following list of passes are available:

- ModulePass
- CallGraphSSCPass
- FuncationPass
- LoopPass
- RegionPass
- BasicBlockPass

All these passes are a subclass of the `Pass` class, which implement the functionality based on their area of application. Each pass of a certain area is executed on different parts of the input source-code. For example, the `ModulePass` is run upon a whole LLVM-IR module, whereby the `FunctionPass` is executed on every function in the LLVM-IR bitcode file. As already mentioned earlier, LLVM categorizes the actions made by passes in three different groups: Analysis, transform, and utility. Analysis passes do not alter the input, but their purpose is to compute information. The objective of a transformation pass is to apply optimizations by altering the code, e.g., the inline pass, which replaces the function call in their callees with the actual function body. Utility passes are used to aid the programmer, for example by printing information to the output stream. Another important feature is the capability to request results from other passes. In order to retrieve these results the required pass needs to be executed beforehand. The right execution order is ensured by the scheduling function of the `PassManager`.

PASS MANAGER Every pass in the LLVM optimizer is registered with the `PassManager`. This component of LLVM manages the active passes and schedules their execution, based on the dependencies they require. A pass can invalidate the results from another one. It is the responsibility of the `PassManager` to reschedule these invalidated passes. Furthermore, it is responsible for sharing the results among the passes. In LLVM version 6 two different `PassManager` exist the legacy version and the replacement for it. We described and used the legacy `PassManager`, since it is still default in LLVM version 6.

MODULE PASS We now take a closer look on the former mentioned `ModulePass`. This pass is the most general of all the passes, which operates on the entire program as a unit. The main advantage is, in the pass we have access to every bit of information that is available for a program, e.g., from the `Module` we can request the list of contained `Functions`, each of these contains a list of `BasicBlocks` where we can request `Instructions` from. Quite contrary to the `FunctionPass` or `LoopPass`, which operate only on small parts of the program. The downside of operating on a whole `Module` is that the `PassManager` has little possibilities to optimize the execution of the passes, since the scheduler has no knowledge about the operations that are performed in the `ModulePass`.

DEFINING A NEW MODULE PASS First, we have to define the constructor where we need to pass an identifier to the base-class constructor. The value of the identifier is not important, because LLVM initializes the ID by using the address of the parameter to identify the pass. The constructor definition looks like this:

```
        static char ID;
        ExampleModulePass() : ModulePass(ID) {}
```

Second, we need to override the `runOnModule` function. It provides the following parameter `Module &M` referencing the module that is currently processed. In the function body follows the implementation, which performs,e.g, analyses or transformations upon the given module `&M`. To signal the `PassManager` whether we modified the LLVM-IR, and thus invalidated other analyses, we need to return `true`, otherwise `false`.

```
        bool runOnModule(Module &M) override {
                /*
                ...
                */
                return false;
        }
```

Before we can use the pass in the optimizer we have to register it in the `PassManager`. To register the pass we need to add the following line after the function definition:

```
static RegisterPass<ExampleModulePass> X("example",
 ↪  "Example pass", false, false);
```

The first argument of the registration function `X` states the command-line parameter to enable the pass. The second one is the name of the pass. The third states whether we modified the CFG and the last one indicates if it is an analysis pass or not. The full code is available in Listing 21 in the appendix.

RUN THE EXAMPLE-MODULE-PASS After we compiled the new pass we can use it to apply the `ExampleModulePass` on LLVM-IR code. First, the optimizer command `opt` needs to load the library containing our pass with the `-load` argument and activate the pass with the `-example` flag, as we provided in the register function. With the last argument we define the input file. The command with all parameter is shown in Listing 7.

```
opt -load lib/ExampleModulePass.so -example
 ↪  example_hello_world.ll
```

Listing 7: LLVM optimizer command example.

2.2.5 *Variability-Aware Region Analyzer*

In the following section, we introduce the Variablility-aware Region Analyzer (VaRA), a framework built into the LLVM compiler infrastructure. This framework aims to enable software-engineering research and aid engineers by providing the capability to extract and use software metrics within the compiler pipeline. Furthermore, it provides reusable data structures and debugging utilities to develop analyses [15].

INTEREST REGIONS VaRA provides the `llvm::IRegion` data structure, which represents a list of instructions from a program grouped together. These interest regions are used in analyses provided by the framework. The creation of a region is managed through the framework and stored in the `llvm::IRegionStore`. The main purpose of this database is to store all different kinds of interest regions in a single optimization run. The store also ensures that each region is unique. Furthermore, the framework extends the base type `llvm::Value`, that is the base class of the types used in the LLVM-IR language, including `llvm::BasicBlock`, `llvm::Function`, and `llvm::Instruction`, with a reference to a `llvm::IInfo` object, which maintains a list of references to `llvm::IRegions`. This allows us to assign a single instruction to an interest region or utilize the ones already created in other detection passes.

FLOW ANALYSIS The `vara::FlowAnalysis` is a generic component managing the traversal through a graph-based data structure to gather information based on the concept of a data-flow analysis. It requires a `llvm::GraphTrait`, which describe how to traverse through the graph, and a `vara::DefaultAnalysisTrait`, which specifies how the analysis information is propagated. The `llvm::GraphTrait` and the `vara::DefaultAnalysisTrait` serve as base classes for an actual analysis. In order to implement a specific analysis we need to provide a `llvm::GraphTrait` template specialization for targeted data structure and a specialized version of the `vara::DefaultAnalysisTrait`. This specialized analysis trait must offer a `join` and `update` method, representing the analysis specific implementation of the meet operator and the transfer function. Furthermore, we have to provide an initial list of nodes and optionally we can provide an initial lattice state, and a filter to omit unwanted nodes from the analysis. These additional information we have to provide heavily depends on the graph trait and the analysis trait we are using.

FUNCTION DEF-USE-GRAPH The `vara::FunctionDefUseGraph` is a data structure offered by VaRA to use in combination with the `vara::FlowAnalysis`. The data structure provide the ability to iterate over the def-use chain of the encapsulated `llvm::Function`. A def-use chain consists of a definition of a variable and all the uses that are reachable from that definition.

TAINT-FLOW-ANALYSIS TRAITS
The vara::TaintFlowAnalysisTraits class is a specialized form of the vara::DefaultAnalysisTrait, which implements a simple context-insensitive taint flow analysis on LLVM-IR. The main purpose of the taint flow analysis is to handle the propagation of taint information based on the type of instruction.

The analysis can be combined with the vara::FunctionDefUseGraph in a FlowAnalysis to perform a data-flow analysis based on taints, as we will see in Chapter 4.

2.3 GIT A REVISION CONTROL SYSTEM

This section introduces the most important parts of the version control system Git to be able to understand how to automatically determine regions by tracking changes applied to the source code of a program versioned via Git.

COMMIT A commit in Git is a snapshot of one or more files in the repository. Furthermore, it is the only way to introduce alterations to a project managed with Git. This fact is exploited in a later chapter regarding the region detection. Additionally, a commit is identified by a SHA 1 hash and serves as a unique identifier for these specific alterations [13, pp. 65,66].

ANNOTATE Git provides an annotate function (git annotate), which annotates every line in a file of a repository with detailed information about the commit that introduced the line. An example is shown in Listing 8. The listing depicts a snippet of the output generated by the annotation function. Every line of the target file is annotated with the unique identifier of the commit followed by the author and date. Finally, the respective source code is displayed.

```
...
a6a87b59 (Chandler Carruth  2015-01-31 03:43:40 +0000
 ↪  18)#include "llvm/CodeGen/BasicTTIImpl.h"
f35ce237 (Hal Finkel        2014-05-08 09:14:44 +0000
 ↪  19)#include "llvm/Analysis/LoopInfo.h"
be04929f (Chandler Carruth  2013-01-07 03:08:10 +0000
 ↪  20)#include "llvm/Analysis/TargetTransformInfo.h"
a6a87b59 (Chandler Carruth  2015-01-31 03:43:40 +0000
 ↪  21)#include
 ↪  "llvm/Analysis/TargetTransformInfoImpl.h"
...
```

Listing 8: Output of the Git annotate function on a file from the LLVM repository.

TAGGING A tag points to an important change in the history of a Git project and is often used to mark release points, as described in the documentation[6]. Tags are managed with the command `git tag`. The example shown in Listing 9 depicts how to create the tag "NewTag" in Line 1. This tag would point to the currently checked-out commit ID. The other command shows how to print a list of all tags in the project, in our case the newly created one, in Line 3.

```
1  > git tag NewTag
2
3  > git tag
4  NewTag
```

Listing 9: Add and list tags in the Git repository.

6 *Git - Tagging*. https://git-scm.com/book/en/v2/Git-Basics-Tagging. (Accessed on 12/09/2017)

3 REGION DETECTION AND ANNOTATION

3.1 INTRODUCTION

In the following chapter, we define how our regions are extracted from common repositories and how they are used to extend the original software with region information. Furthermore, we explain how the information passes the clang compiler front-end and how it is processed in the optimizer of the compiler.

REGION DEFINITION We define a region as a set of program instructions combined in an organizational unit that does not alter the behavior of the program itself. They may not change the output of the binary directly, but they can be utilized by the compiler in order to optimize the output binary.

For our approach, we particularly need two kinds of regions at different layers. On source-code level we introduce source-code region (SC-R), these are then linked to their LLVM-IR counterparts the IR-meta region (IR-R), to introduce a mapping between source code and LLVM-IR.

A SC-R is used to add region information to a software in its original form. It is organized as a group of consecutive statements and declarations surrounded by a begin and an end annotation. Every SC-R has an identifier, which has to be unique throughout the whole software. Furthermore, a SC-R can overlap or be nested within other regions.

The IR-R is used to add region information to a software program that was translated to LLVM-IR. An IR-R is identified by a unique string. This identifiers are used to annotate single functions, basic blocks and instructions in order to map them to the corresponding IR-R. In order to express nested regions each one can have a list of identifiers.

APPROACH We begin with a software repository, which provides the necessary information needed to detect regions, in particular the source code of the program and the commit hashes, e.g., a Git repository. In the next step, the source files are annotated with SC-R based on the changes tracked by RCS. Next we execute the compiler front-end with the annotated source files, in order to translate the source-code and its annotations into LLVM-IR. Finally, the front-end passes the information to the optimizer, which is able to extract and use this information for further analyses and optimizations. The approach and the steps are depicted in Figure 3.

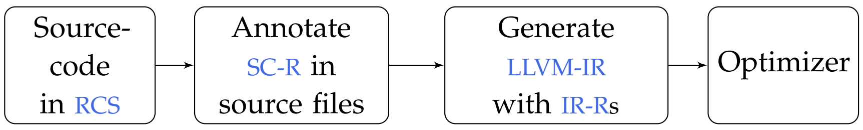

Figure 3: Region detection and annotation work-flow.

In the following sections, we describe every step in more detail.

3.2 INTRODUCING SC-R TO CLANG

In order to enhance a software program with regions the compiler front-end needs to be capable to process the provided information. In this section we describe how to enhance clang with this capability. Fist we present our general approach before we go further into detail by describing how SC-R are introduced in the programming languages C and C++. Furthermore, we explain the parsing phase and its challenges, the processing stage, and how the information is emitted and passed along to the LLVM optimizer.

3.2.1 *General Approach*

The work-flow depicted in Figure 4 represents the general approach how the region detection works in clang. In the first stage the compiler reads the input file and passes the source code to the second stage, which extracts the SC-R information. In the processing phase the data is used by the front-end to create a mapping of the region information from statements and declarations to instructions. These are then facilitated in the third stage, where they are annotated as IR-R and forwarded to the last phase where the emitted instructions are either written to an output file or passed on to the optimizer.

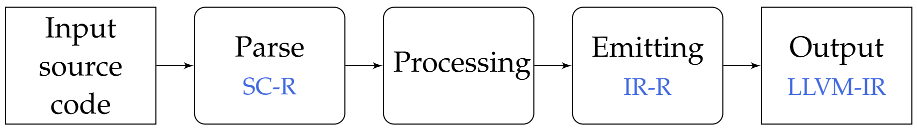

Figure 4: The work-flow depicting the general approach to convert SC-R to IR-R.

3.2.2 *SC-R Keywords*

In order to describe a SC-R two keywords are needed. One keyword to start and another keyword to enclose a SC-R. For this, we extended the clang front-end with **`__REGION_START`** `<ID>` to open and **`__REGION_END`** `<ID>` to enclose a SC-R. Each keyword expects an argument named `ID`, which serves as unique identifier over the whole project. An example written in C is depicted in Listing 10. The example shows three regions identified by `R1`, `R2`, and `R3`. `R1` has two nested regions `R2` and `R3`. `R2` overlaps with `R3` in Line 8.

```
1  ___REGION_START R1
2  #include <stdio.h>
3
4  int main() {
5          ___REGION_START R2
6          int a = 1;
7          ___REGION_START R3
8          int b = 0;
9          ___REGION_END R2
10         int result = a / b;
11         ___REGION_END R3
12         printf("Result: %d\n", result);
13 }
14 ___REGION_END R1
```

Listing 10: C example with SC-R.

3.2.3 *Parse SC-R*

The first attempt to process the newly introduced keywords, from Section 3.2.2, was to extend the language with a new kind of scope, the region scope. Similar to a block scope in C and C++, the `___REGION_START` keyword is used to open a new scope and the `___REGION_END` to enclose it. The idea was to process and annotate all instructions surrounded with these keywords. Unfortunately this attempt did not cover all desired applications. The following paragraph describes why this attempt brought no success.

SYNTAX PROBLEM One of the main goals is that the keywords defining a SC-R are not depending on the position in the source code. Unfortunately the lexer and the recursive-descent parser used in clang are not designed to support context-independent keywords.
The recursive-descent parser technique constructs a tree from top to bottom by parsing the input provided by the lexer beginning with the left most symbol and ending with the right most. For every terminal and non-terminal symbol exists a procedure handling a single production rule from the C++ language grammar[8].

The problem is that the lexer is reading every token from the source buffer and decides what the respective token means and how it has to be processed. The token is passed on to the responsible parser function, which validates the syntax by requesting additional tokens from the lexer. The following example, depicted in Listing 11, illustrates the problem. The example shows an if-statement and a SC-R annotation identified by `IF_1`. The lexer reads the **`if`** token and passes it on to the parser, which performs a syntax check. In the validation process further tokens are requested from the lexer. As soon as the parser tries to validate the **`__REGION_START`** token an error is raised. The reason causing this error is that the function handling the if-statement is not aware of the keyword at this point in the source code. In order to solve the problem the SC-R annotations need to be introduced to every function in the parser performing a similar syntax check, making this technique hard to implement, maintain, and thus error-prone.

```
if (true)
__REGION_START IF_1 // Error
{
        printf("Hello World!");
__REGION_END IF_1
}
```

Listing 11: Syntax validation issue.

CONTEXT-DEPENDENT KEYWORD PROBLEM Similar to the challenge described above, the parser used in clang reveals another problem concerning context-dependent keywords. The fact that keywords depend on the position in the source code makes the implementation much more complex. In order to make the SC-R annotation keywords context-independent the parser needs to be extended with additional production rules. The example in Listing 12 illustrates the problem. In the first line a SC-R start annotation is added, which is parsed as a top-level declaration. In Line 2 the parser enters the function `func1`, which is not handled by the top-level declaration routine. In order to parse the annotation in Line 5 the keyword needs a separate implementation in the statement parser function.

```
1  __REGION_START C_1 // Top-Level-Decl
2  void func1 ()
3  {
4          printf("Hello World!");
5          __REGION_END C_1 // Statement-Or-Declaration
6  }
```

Listing 12: Context-dependent keyword issue.

SOLUTION Since these two challenges imply major changes in the parser component of clang we propose another approach. In order to circumvent problems similar to the ones mentioned earlier the SC-R implementation is done in the lexer. The `Lex` function is visited every time the parser requests a token from the stream. The current implementation utilizes this bottleneck by calling its own custom parsing function for every token appearing in the source code. As soon as either a `__REGION_START` or `__REGION_END` occurs the keyword and the following string literal are consumed. The consumed string literal, the ID of the region, is added to a list of active regions if the start annotation is consumed and removed from the list if an end annotation is consumed. For all other token a mapping between the active regions and the current processed token is created.

3.2.4 *Processing*

The extended lexer is now able to handle SC-R annotations and use the information they provide to further process it in the internal structure of clang. In the following, we first introduce the region store, which maintains a mapping between tokens and the regions it is contained in, then we describe how we use the component to propagate information to the code generation.

REGION STORE The `RegionStore` class provides functionality to lookup region information for a specific instruction based on its source location. The component is available throughout all compilation stages in the front-end. Its main purpose is to allow the region handler to store region information and make it available to the code generation.

INSTRUCTION ANNOTATION The main goal regarding the region detection and annotation in clang is to transfer the region information from the source code to the LLVM-IR, so we can analyze it later on. In order to do that a SC-R is converted into a IR-R. Since every instruction in a IR-R contains information about the regions it is a part of, the compiler front-end needs to ensure that the information is accessible during the code generation. We achieve that by storing the regions for every statement and declaration and provide access to the information while emitting the corresponding instructions. The implementation in clang works as follows. For every occurrence of a `__REGION_START` annotation in the source code, the compiler appends the identifier of the SC-R to the `activeRegion` set. An entry is removed as soon as the `__REGION_END` annotation with the specific identifier is processed. The source location of all instructions, handled during the module compilation, is used to generate a mapping between the entries in the `activeRegion` set and the currently processed instruction. These mappings are then stored in the `RegionStore` database.

3.2.5 *Emit IR-R*

The final step to complete the conversion from SC-R to IR-R is to annotate the instructions in LLVM-IR with region information. As we already mentioned in Section 3.2.3 we do not want to adapt every function in clang. Therefore, the general idea is to circumvent all the following stages in the compilation process after the parsing phase and identify the instructions right before they are emitted by the code generation. We then use the gathered information from the last section to annotate the affected instructions.

IDENTIFY INSTRUCTION IN CODE GENERATION In order to identify instructions we facilitate the information offered by the `SourceLocation`. It serves as a unique identifier and is assigned to every token parsed in the front-end. The ID is passed along with the source-code fragment no matter how the information is transformed during the process. In the code generation we exploit this fact by looking up the ID in the `RegionStore`, which provides us with region information of the currently processed instruction. Since there are three different kinds of instructions the implementation has to be done in all three. This affects the `CodeGenModule::EmitTopLevelDecl`, `CodeGenFunction::EmitStmt` and `CodeGenFunction::EmitDecl` methods.

TACKLING INFORMATION LOSS The code generation in clang uses functions from LLVM to generate LLVM-IR instructions. Since LLVM is language independent, it has no access to data structures specific to the front-end like the `SourceLocation`, which is needed to access the information in the `RegionStore`, though these insights are needed to generate the IR-R annotations. We solve this problem by introducing the `llvm::RegionAnnotation` class to LLVM, which caches a string containing the information for the currently processed instruction. The clang front-end is now able to pass the output from the `RegionStore` to the `llvm::RegionAnnotation`, which enables LLVM to request the necessary information.

LLVM-IR METADATA The instructions in LLVM, similar to the one in the code generation of clang, are processed on three different choke points, namely when constructing a `llvm::Function`, `llvm::Instruction`, or a `llvm::GlobalVariable`. In order to annotated each value that is emitted we need to adapt all of these constructors. Fortunately, the VaRA version of clang offers the `MDProvider` and the `MDAnnotater`, which provides an interface to add metadata without the need of any modification to one of the constructors. These VaRA components work as follows: Our class, to handle the information conversion and annotation, `RegionAnnotation` extends the `MDProvider` class and overrides the `getCurrentMetadata` function, which provides the region information as `MDNode` object. The `MDNode` class is an attachment representing the metadata for emitted instructions containing the metadata-group and data, in our case the region information. During the construction of our class it introduces itself to the `MDAnnotator`, that requests and adds a `MDNode` to an emitted instruction. This annotated information is identified by the metadata-group *Region*, such that the optimizer is able to recognize that the metadata is containing IR-R information.

3.2.6 *Process IR-R*

In order to facilitate the information provided by the IR-R we create an optimizer pass, which extracts and stores the information. The `vara::MarkerDetection` pass enhances the VaRA framework with region detection capabilities and provides compatibility to its analysis functions.

EXTRACTION In order to extract the information stored in the LLVM-IR files we have to create a new module pass in the optimizer. The pass is supposed to run on a whole module because we want to store information for every function, basic block, or instruction separately. Storing it separately allows us to run the generic analysis functions available in VaRA not only on instructions but on basic blocks, and functions, too. The module object provided by the pass enables us to request every component, which are nested within one another. To extract the information we iterate through all the interleaved components and parser their metadata and build an internal database based on the following set. Let F be a set of functions, BB a set of basic blocks, I a set of instructions, R a set of regions, and $g(c)$ a function returning the assigned regions of the provided component c. For every instruction $\forall i \in I$ that is part of a basic block $bb \in BB$, the regions assigned to the instructions $g(i)$ are also assigned to the parent basic block bb. This also applies in a similar manner to every basic block $\forall bb \in BB$ that is contained in a function $f \in F$, the regions assigned to the basic block $g(bb)$ are also assigned to the parent function f. We can express the resulting set as follows: $g(i) \subseteq g(bb) \subseteq g(f)$.

INTERNAL REPRESENTATION Regions are represented as a `vara::MarkerRegion` in LLVM. The internal representation is derived from `llvm::IRegion` interface, which is part of the VaRA framework. VaRA provides generic analysis functions, which require to implement the former mentioned interface. This interface ensures that the analyzer is able to obtain the relevant data gained in the previously described extraction process.

MAPPING Every `vara::MarkerRegion` identified by its `ID` is created once to keep the overhead low. However, a region can be defined by more than one basic block, function, or instruction. Hence, we have to create a mapping between instructions and regions. The VaRA compatible version of LLVM provides the `llvm::IInfo` object that is capable of describing such a mapping. Furthermore, the `llvm::LLVMContext` provides a separate mapping assigning an `llvm::IInfo` object to a `llvm::Value`, which is the most common base class of every basic block, function, and instruction. This allows us to add a reference in the `llvm::IInfo` object to the corresponding `vara::MarkerRegion`. The creation and requests of regions are managed by the `llvm::IRegionStore`, a component available in the VaRA compatible version of LLVM.

3.3 AUTOMATED ANNOTATION OF SC-R USING GIT

In the previous sections we ensured that the compiler is able to utilize SC-Rs from a provided input file. We now want to annotate whole projects with SC-R automatically. In order to do that we make use of the commit information provided in the RCS Git to find and annotate regions.

IDENTIFY RESPONSIBLE COMMIT HASHES In order to annotate the files in the repository with region information we need to identify which lines are affected by a certain commit. We can achieve this by using the `git-annotate` command. We process the output generated by `git-annotate` by tracking down the first an the last change made by a certain commit. A SC-R is defined by all the code that is contained between the first and the last change. For example, the output depicted in Listing 13 contains a simplified version of the output `git-annotate` would produce. As we can see the first line affected by commit with ID *9aa9da* is in Line 1 and the last line affected is in Line 10. Hence the resulting SC-R contains all lines of the source file. In comparison with the commits affecting Line 5 and Line 7, where the first is the same as the last line modified by the responsible commit. Therefore, only the single line is contained in the resulting SC-R.

```
9aa9da  1)  #include <stdio.h>
9aa9da  2)
9aa9da  3)  int  main() {
9aa9da  4)      int a = 1;
d8527c  5)      int b = 0;
9aa9da  6)
45dcb1  7)      int result = a / b;
9aa9da  8)
9aa9da  9)      printf ("Result: %d\n", result);
9aa9da 10)  }
```

Listing 13: Simplified output of `git-annotate`.

In the next step, we generate a new version of the source file that contains the SC-R annotations. As identifier for these SC-R we use the commit hashes. They are especially suitable because it is unlikely a repository yields two identical hashes, as stated in the official Git documentation[1]. Furthermore, they can be useful if we want to combine the insights generated by an analysis and the information provided by Git. The result of the transformation from the example used in Listing 13 is provided in Listing 14. For every detected commit we added a begin annotation before the first line, see Line 1, 6, and 10 and an end annotation after the last affected line, see Line 8, 12, and 16.

```
1  ___REGION_START "9aa9da"
2  #include <stdio.h>
3
4  int  main() {
5  int a = 1;
6  ___REGION_START "d8527c"
7  int b = 0;
8  ___REGION_END "d8527c"
9
10 ___REGION_START "45dcb1"
11 int result = a / b;
12 ___REGION_END "45dcb1"
13
14 printf ("Result: %d\n", result);
15 }
16 ___REGION_END "9aa9da"
```

Listing 14: Resulting source file after annotation is finished.

1 *Git Tools - Revision Selection*. `https://git-scm.com/book/en/v2/Git-Tools-Revision-Selection`. (Accessed on 12/05/2017)

TACKLE MULTI-LINE PREPROCESSOR CODE The clang front-end handles the SC-R annotations at compile time after the preprocessing is finished. The detection and annotation process we just described does not concern itself whether the lines wrapped are preprocessor instructions or C/C++ code. Based on these facts an error arises as soon as an annotation is between a multiline preprocessor definition. An example causing a compiler error is shown in Listing 15. Processing the example with the clang preprocessor is starting to parse the definition in Line 1, since the definition is not finished by the end of the line, expressed with the \ (backslash), the parser consumes the next one. Unfortunately the `___REGION_START ERROR_1` is not a keyword the preprocessor can act on, therefore it raises an error.

```
1 #define F(x) (x) \
2 ___REGION_START ERROR_1
3         +  \
4 ___REGION_END ERROR_1
5         (x)
```

Listing 15: Annotated multi-line macro in C.

We can solve the problem by converting multi line to single line preprocessor code before we add the annotations to the output-file. The downside of this solution is that we loose accuracy, since the region shown in the example affects only the plus operator. After reducing the macro to a single line the whole macro is contained in the SC-R. However, omitting the information has little effect on the outcome, since the automatic annotation considers only line changes. For example, the example depicted earlier in Listing 14 in Line 11. If we change the operator from a division to addition the corresponding annotation affects the whole line, which is inaccurate since we only changed the operator. Since we accept this kind of inaccuracy the impact of converting multi-line to single-line macros is negligible.

3.3.1 *Tag-group-based Annotation*

The organization of commits is the responsibility of the project. A single commit can introduce a simple change or contain a whole feature. However, Git offers another organizational feature that we can base our annotations on. Tags in Git refer to a certain commit ID in the working tree and they are used to highlight important points in history as stated in the Git documentation[2]. We exploit the fact that tags mark important changes to group them together, to be able to analyze specifically these sets of important changes.

2 *Git - Tagging*. `https://git-scm.com/book/en/v2/Git-Basics-Tagging`. (Accessed on 12/09/2017)

To list the available tags in a project Git offers the command `git tag`. In order to get the corresponding commit ID and its predecessors we use the `git rev-list` command. We combine the two commands to get a set of IDs newly introduced in the given tag. The example in Listing 16 shows the execution of the `git rev-list` command in Line 1. The command is provided with two tags "v1.7" and "v1.8". The three dots separating these tags affect the result of the command to yield all changes made after the first tag `v1.7` up to the change referenced by the second tag `v1.8`. The resulting list is ordered top-down, the commit ID in the first line is the newest.

```
1  > git rev-list v1.7...v1.8
2  3022fdb32287089d2201e715b7beb133a47ff608
3  d21616b9a9479331d721c07c6ad56bd72f4edd1f
4  f879a0511c034a21b0e9a15caf866006f9d1bed5
5  3557cd57906915eb9c990b5f386e25c395592643
6  a420fbafe812f9584f4e71cf3bd42f222bae74c8
7  525cc2dd42483bbdb4e4e60cec659fd286ead55f
```

Listing 16: Grouping git tags for annotation.

The annotation process is performed similar to the one described previously with the difference that all IDs yield by the `git annotate` command are substituted with the corresponding tag name. In this manner we are able group together small changes to larger organizational units.

4 EVALUATION

In the last chapter, we introduced an approach to detect change-regions from a Git repository and propagate them forward to LLVM. In this chapter, we describe how we utilize commit-region information to analyze software. We introduce approaches based on control-flow and data-flow in order to determine interactions between regions. We visualize and then discuss our results in order to provide an answer to our research questions.

IMPACT OF A CHANGE By our means, the impact of a change, introduced by a commit, is measured by the effect it has to the behavior of a program.

RESEARCH QUESTIONS The following three questions are evaluated in the upcoming chapter:

- **RQ1.1**: Can we use LLVM-IR based **control-flow** to detect interactions between different commit regions?
- **RQ1.2**: Can we use LLVM-IR based **data-flow** to detect interactions between different commit regions?
- **RQ2**: Can we measure the impact of a change introduced by a commit region through its interactions to other commit regions?

4.1 CHANGE-REGION BASED ANALYSIS

In this section, we introduce the LLVM passes we created to evaluate our approach. Furthermore, these passes should give an idea how the change-region information can be utilized for the analysis of software. At first we describe the control-flow and data-flow analysis passes, which uses the regions provided by the detection pass introduced earlier. The results generated in the analysis are then aggregated in the report pass, which serializes the information to YAML and stores the content to an output file.

4.1.1 *Control-Flow Analysis*

The purpose of the analysis is to generate a graph from the extracted IR-R based on the control-flow. The result generated in the process is an ordered set of visited IR-R gathered while traversing along the control-flow edges. In addition, the analysis gathers call information from source to destination IR-R. The pass is based in the VaRA framework, where it utilizes the `vara::FlowAnalysis` component to generate insights. In order to be able to use this component, we need a compatible data structure and a flow-analysis trait that fits our need.

MARKERINFO Before we can introduce the flow-analysis trait we need to define an object capable of storing information that is propagated throughout the analysis. The `MarkerInfo` object stores three different attributes: (a) Predecessor IR-R, the IR-R visited in the previous node, in order to detect a control flow between two regions. (b) Control-flow edges, to track the flow from one region towards an other. The structure is organized as a pair, whereby the first precedes the second regarding the control flow. (c) Called functions, a map which uses the source region as a key and a list of called functions as value.

MAKER-FLOW-ANALYSIS TRAITS The `MarkerFlowAnalysisTraits` gathers control-flow information form the `llvm::Instructions` contained in `llvm::BasicBlocks` and propagates it within `MarkerInfo` objects. The flow-analysis trait is derived from the `vara::DefaultAnalysisTrait` that requires to implement the `join` and `update` function to work. A description of how they are implemented follows.

UPDATE The `update` function is called by the underlying flow-analysis algorithm and provided with the following three parameter:

1. The corresponding `llvm::BasicBlock`, since we perform the `vara::FlowAnalysis` on a `llvm::Function`.
2. A `MarkerInfo` storage object as IN set, which provides the results from the predecessor.
3. A `MarkerInfo` storage object as OUT set, which stores the intermediate results gained in the current update step.

We start our update routine by merging the previous obtained control-flow edges and function-call mappings with the empty OUT set, in order to preserve the results already obtained.

We use the predecessor regions from the IN set and copy them to an intermediate set of predecessor regions, since the IN set is immutable and we need to update the predecessor region set nearly after every instruction.

In the next step, we iterate through the instructions contained in the basic block node. In every iteration we check whether the instruction is a call to a function, and if so, for every region assigned to this instruction we store a mapping in the OUT set associating those regions with the targeted function of the call instruction.

In the same iteration step we use the intermediate predecessor regions to create control-flow edges by combining every region from this set with all those assigned to the current instruction. Each newly created edge is added to the OUT set and the intermediate predecessor region store is emptied and refilled with the regions assigned to the currently processed instruction. If an instruction has no regions assigned to it the algorithm will simply use those from last predecessor that had regions assigned to it. If there are no preceding instructions with region entries nor entries from the IN set, no control-flow edges will be generated.

After all instructions are processed the predecessor region store in the OUT set is filled with all regions assigned to the basic block. Since we propagate the regions upward in the detection pass, all regions assigned to any instruction in the basic block are also assigned to the parent. The exact behavior how regions are propagated is described in Section 3.2.6.

Finally, the function returns `true` if the OUT set differs from the IN set, otherwise `false` is returned. This information is crucial to the underlying `FlowAnalysis` component, e.g., to handle recursion.

JOIN The `join` function has a rather simple responsibility, it merges the results from two different paths in the flow. The function is provided with four parameters:

1. The preceding node.
2. A `MarkerInfo` storage object as IN set, which provides the results from the preceding node.
3. The currently processed node that needs joining.
4. A `MarkerInfo` storage object as OUT set, in which the changes are joined into.

In our case the join function just needs to copy the contents from the IN set to the OUT set.

Similar to the `update` function the return value is either `true` if the OUT set is different to the IN set, otherwise `false` is returned.

ANALYSIS WRAPPER-PASSES We have to create a new pass to actually run the flow analysis. Since we built our control-flow based analysis on functions we create a function pass. In the `runOnFunction` method we perform the actual analysis based on the algorithm introduced in Section 4.1.1. First, we filter whether the function is defined externally or not. We can not access the definition of functions that are introduced via a library. If the function fulfills the prerequisites we continue to initialize the analysis. Second, we have to provide a set of entry blocks, which define the basic blocks used to start the traversal through the flow graph, and the targeted function. Furthermore, the `MarkerDetection` pass, which handles the extraction of IR-R from LLVM-IR, is required as a dependency. Finally, we run the analysis and store the results in the `Result` field of the `MarkerFlowDetection` pass.

4.1.2 *Data-Flow Analysis*

In the last section, we introduced our implementation to analyze the control flow of a program in order to find relations between IR-R. In this section, we describe how we determined the data-flow relations between regions. Fortunately, VaRA already offers an implementation to analyze the data-flow of a program, which we are able to utilize for our evaluation.

FLOW-ANALYSIS COMPONENTS Again we use the `vara::FlowAnalysis` component in combination with the `vara::FunctionDefUseGraph`, which provide paths based on def-use chains, and the `vara::TaintFlowAnalysisTraits`. The flow analysis manages the traversal through the `vara::FunctionDefUseGraph` and the propagation of gathered insights with the `vara::TaintFlowAnalysisTraits`. However, the analysis trait operates on taints, which are represented by the `vara::Taint` structure. We need to extend the implementation such that nodes in the graph can be tainted with their corresponding IR-R.

The `MarkerTaint` class inherits from the `vara::Taint` class and essentially wraps a reference to the corresponding IR-R offering wrapper functions in order to be compatible with the `vara::TaintFlowAnalysisTraits`.

ANALYSIS WRAPPER-PASSES Again we create a function pass actually executing the analysis. Since the data-flow analysis is also based on functions, similar to the analysis based on control flow, however, the function is wrapped in the `vara::FunctionDefUseGraph` structure. In addition, we need to provide a list of entry blocks containing all instructions from the currently processed function and a lattice consisting of a preset taint to instruction mapping. Again, the `MarkerDetection` pass is required as a dependency. We can access the results via the function pass.

4.1.3 *Report Pass*

The results gained in the analysis need to be aggregated an exported in order to use the insights for external processing. For this, we created the `MarkerFlowReport` pass that runs on the whole `llvm::Module`. The aggregation process is performed as follows. We start by iterating over all functions in the module, except those we were not able to analyze, external functions and functions with no annotations. The results are copied to an intermediate data structure used to generate a YAML file.

YAML GENERATION We export the generated insights in YAML format using the YAML traits provided by LLVM. These traits describe how to map from an internal data structure to its corresponding YAML formatted string and backwards. Our implementation, however, only supports export of results. A more detailed explanation can be found in the corresponding LLVM documentation[1].

In order to save the results to disk, we need to set the path to the output file with the `-yaml-out-file` parameter. The example shown in Listing 17 and 18 illustrates how the report is organized. The file is extracted as a whole, though it contains two different YAML documents.

The first one, shown in Listing 17, contains a mapping of functions and IR-R from their unique ID to entry-specific information.

The first list in this document represents all functions contained in the module, even those not providing results. The reason why we added them as well is that function calls can call an external function and we want to be able to resolve these functions. Each function representation in the list has a unique ID, the first region appearing in the body, if available, and its textual representation.

The second list in this document contains a mapping from internal used unique IDs to their string representation of IR-R. The second

```yaml
---
function-info:
  - id:               11
    region-id:        21
    function-name:    foo
  - id:               12
    region-id:        22
    function-name:    bar
region-mapping:
  - id:               21
    representation:   'Region_1'
  - id:               22
    representation:   'Region_2'
  - id:               23
    representation:   'Region_3'
...
```

Listing 17: Example report in YAML format, containing string mappings.

1 *YAML I/O — LLVM 6 documentation*. `https://llvm.org/docs/YamlIO.html`. (Accessed on 12/11/2017)

document in the report contains a list of functions and their corresponding results. These results are identified by the textual representation of the function and followed by a list of called functions identified by `call-graph-edges`. Each of these entries describe which ones are called from a certain region, whereby the regions are referenced by their ID and the functions by their string representation. Furthermore, for each distinct region a unique entry is created and each of these entries can contain a list of targeted functions. The list is in the same order in which they were analyzed. The second list `control-flow-edges` in the report describes the control flow between regions as an edge, from the source region `from` to the destination region `to`, whereby both values represent a region by their unique ID. The `data-flow-relations` contains a similar list, however, the entries represent data-flow edges between regions.

```yaml
---
- function-name:   foo
  call-graph-edges:
    - from-region:     21
      to-functions:
        - bar
        - foo
    - from-region:     23
      to-functions:
        - bar
  control-flow-edges:
    - from:            21
      to:              22
  data-flow-relations:
- function-name:   bar
  call-graph-edges:
  control-flow-edges:
    - from:            21
      to:              23
    - from:            23
      to:              21
  data-flow-relations:
    - from:            21
      to:              23
    - from:            23
      to:              22
...
```

Listing 18: Example report in YAML format, containing analysis information.

4.2 TOOLS

The following section describes all tools used to facilitate the evaluation process. We start by introducing the graph-plot generation tool, which creates a graphical representation of the relations between regions. We continue with the interaction-plot tool, which offers a quantity-based look on the results. Further, we introduce Whole Program LLVM (WLLVM)[2], a wrapper for the LLVM front-end clang to produce a binary and its corresponding bitcode version. Finally, we describe a tool used to ease the result generation of larger projects.

4.2.1 *Graph Visualization*

The graph plot generation tool utilize the report we introduced in the last section. The tool is designed to serve only for evaluation purposes, therefore, the settings are limited to process a given input file and output the resulting plots. The input must have the format introduced in Listing 17 and 18, otherwise the tool will not be able to process the data. If provided with a valid report it will write the plots in Scalable Vector Graphics (SVG) format to the subdirectory `plot` in the current working directory.

CONTROL AND DATA FLOW IN FUNCTIONS The first kind of plot generated depicts the control and data flow between regions regarding a function. An example, base on the report shown Listing 17 and 18, is shown in Figure 5. The plot is structured as follows. The function name is depicted on the bottom center. The nodes in the directed graph represent different regions, they are shown regardless whether a flow exists or not. The solid directed lines describe the control-flow edges, e.g., our analysis detected a control flow in function `foo` from `Region_1` to `Region_2`. Similar, dashed directed lines represent a data-flow edge, e.g., in function `bar` from `Region_1` to `Region_3` and from `Region_3` to `Region_2`. A mutual control or data flow is depicted with two separate lines, as we can see in the example function `bar`. The plot is generated for every function we were able to analyze.

REGION-BASED CALL GRAPH The next plot generated describes the call relations between regions. We take a look at the example depicted in Figure 6. Again every node in the graph represents an independent region and every solid line describes one or more calls toward a function located in the targeted region. The source region of the edge is determined by the call instruction and depicted with the plain end of the line. If the call instruction is located in multiple regions, an edge will be drawn for every source region. The destination region is defined by the first one appearing in the control flow of

2 https://github.com/travitch/whole-program-llvm

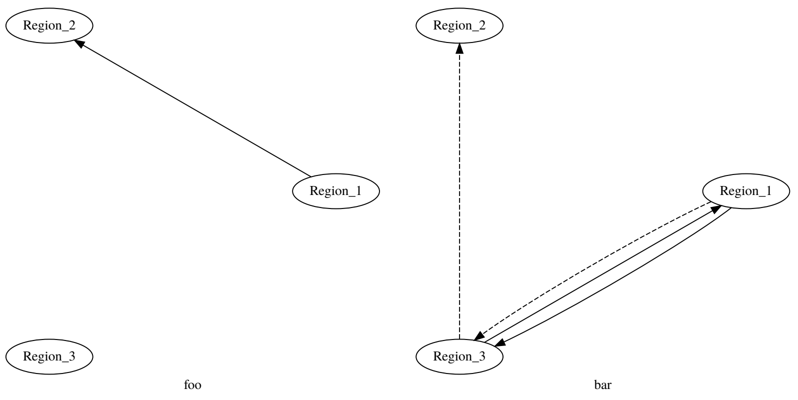

Figure 5: The graph visualization of the functions "foo" and "bar".

the targeted function, if the target has no region in the control flow a new node with the function name is generated and used instead. Furthermore, the edges targeting such functions are drawn with dashed lines. In the example the external function `printf` is drawn as separate node identified by its name since it is defined external and we have no region information. In the graph visualization the destination region is denoted with an arrow head pointing towards it.

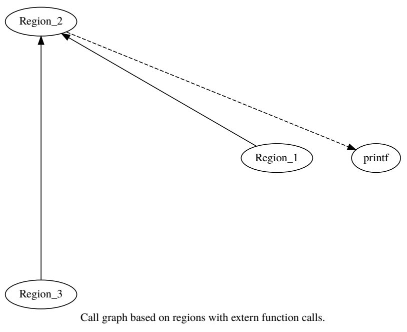

Figure 6: The call-graph visualization based on regions.

OVERVIEW The graph-plot generation also offers a visualization depicting a combination of both variants, flow graph with call edges. The example depicted in 7 illustrates the combination of the above introduced plots. Each graph visualizing a function is located in a circle around the zero point. The function graph is similar to the one introduced above. However, the call-graph edges are drawn from the function where the call instruction is located towards the target function. Furthermore, the edges are connecting the corresponding regions directly, in the source function the region, in which the call instruction resides in, is connected with the destination functions entry region.

However, in our opinion, this type of visualization is only useful for graphs with a small amount of nodes, otherwise the illustrations gets very complex. For automated annotation it is possible to reduce the region size by enabling tag based annotation or a commit-ID filter in the annotation tool.

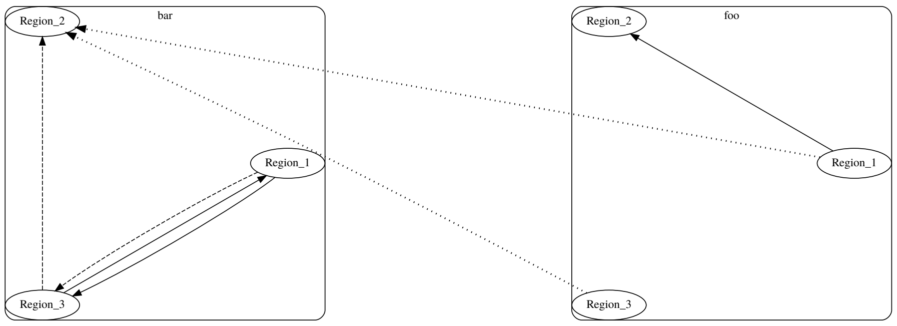

Figure 7: An overview depicting the control flow, data flow, and call relations between functions and regions.

4.2.2 *Interaction Plot*

The interaction-plot tool utilizes the data provided in the YAML report to generate a bar chart depicting the quantity of interactions based on the control and data flow between regions. The example in Figure 8, depicts two interaction plots. Both are based on the example we used earlier in Listing 18. The plot on the left is generated with the data from the control-flow analysis. The plot on the right is based on the data flow. In both plots for every region a set of two bars is displayed, the left one with a loose mashed red and orange texture represents the amount of interactions originating from the region. The other one on the right with the tight mashed texture in blue and yellow describes the amount of interactions targeting the region.

This kind of visualization allows us to reason about the complexity of a region regarding the size and impact.

However, the report does not contain duplicates, regarding the control and data flow edges, therefore we can not fully deduce the impact of a region in a single function with this technique. Still, we are able to reason about the impact beyond these function boundaries.

4.2.3 *Whole Program LLVM*

Whole Program LLVM (WLLVM) is a tool set for building a binary and a LLVM bitcode version from C or C++ projects. We use these generated bitcode files to perform our flow analyses, described in Section 4.1, upon them. WLLVM works as a wrapper for the clang front-end, that can be used as a drop-in replacement for the compiler. We use the tool to compile our sample projects that are introduced in Section 4.3. The tool processes the project in two stages. In the first one it invokes the compiler to build the object files. The second stage uses these object files to generate LLVM bitcode. Finally, the

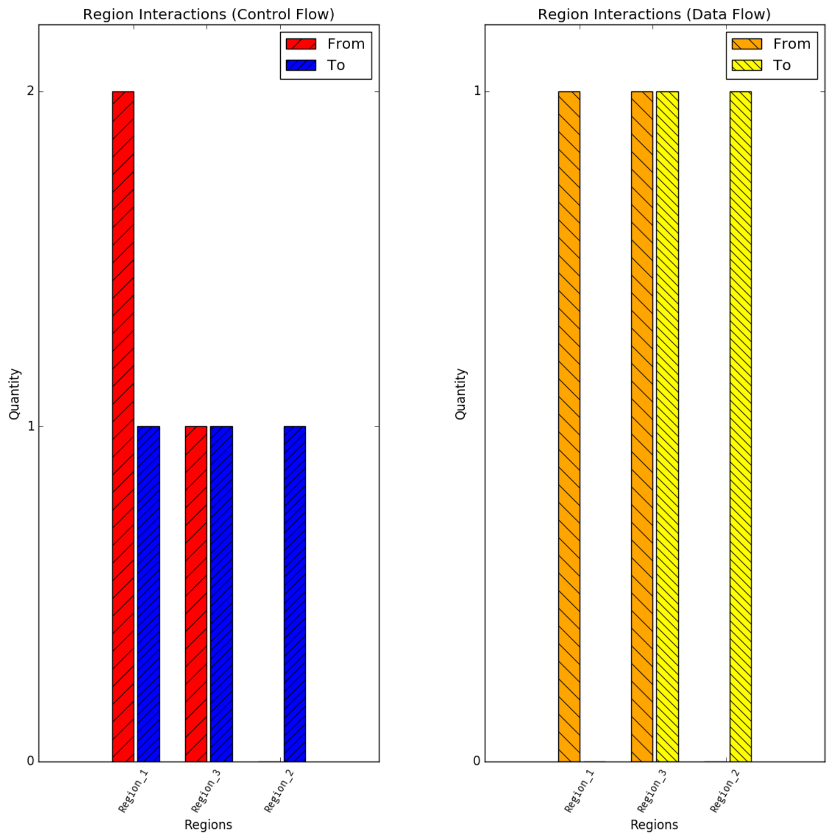

Figure 8: Interaction-quantity bar charts, control-flow based on the left and data-flow based on the right.

result is linked together providing a binary and a bitcode version of the program. WLLVM project provide a detailed explanation in their documentation[3].

4.2.4 *Repository Preperation*

In order to speed up the evaluation process, we created a preparation tool which utilizes the graph generation and WLLVM to provide an environment that is ready to start a build. The preparation script works as follows, it creates a copy of the chosen repository in a new subdirectory called `out`. Afterwards the annotation tool annotate all C and C++ files in the `out` directory. Furthermore, the tool sets up the WLLVM environment variables in order to use clang as compiler. Lastly, a series of user-defined commands are executed sequentially or, if no commands are provided, a new shell is started with the

3 *Whole Program LLVM: A wrapper script to build whole-program LLVM bitcode files*. `https://github.com/travitch/whole-program-llvm`. (Accessed on 12/13/2017)

proper settings. We use this tool to build our sample projects later in this chapter.

4.3 samples

In this section, we first introduce a hand-crafted program written in C and how we use it identify errors in the annotation process. Furthermore, we introduce a Git based software project, where we perform our evaluation on.

4.3.1 *Hand-Crafted Example Program*

Before we apply our analysis on preexisting software we developed a small example that serves as a sanity check, to verify if the analysis works as expected. The example is written in C and consists of a single file, 5 functions, and 59 lines of code in 9 commits. It introduces a very basic calculator that offers addition and subtraction, as well as input and output functionality. The commits are small, about 6 lines per commit, and contain only essential changes, e.g, a single commit introduces the `add` function and another the implementation in the `main` function. With small commits we intended to get a better look on the impact of faulty changes. A list of all commits applied to the example is shown in Table 2 and 3 in the appendix. These tables consist of the following four columns: *Nr.* the order in which the commits were submitted. We use the number later on in the analysis to refer to single commits. The commit *Message* tell how we describe the changes, and *Diff* denotes the code changes introduced with the commit.

Furthermore, the full example in its latest state is provided in the appendix in Listing 22.

4.3.2 *GNU Zip*

Our second example is gzip a compression utility maintained by the GNU project[4]. At the time of evaluation the program consisted of 17 C source files and 4 header files containing a total of 5700 lines of code without documentation and dependencies. The project is organized in a Git repository containing 556 commits in the master branch. On average a commit introduces 91 and removes 57 lines to files in the repository. These measurements concern all files in the repository not only the source code files. Base on these commits we could identify a maximum of 119 individual regions alone in the source files.

4 `https://www.gnu.org/`

4.4 EXPERIMENT 1 - CALCULATOR

In the following section, we answer our research questions by analyzing our hand-crafted example. We already generated the results and visualized these in graph plots.

OVERVIEW GRAPH The graph, depicted in Figure 9, provides an overall view of the program from the perspective of the analyses. The depicted figure does not require that we understand the nodes in detail, we use it just to get an impression of the results, since most of the insights generated are visualized in the overview.

Before we take a look on single functions we examine the information provided by this graph. We recall from earlier: Dotted lines are function calls, dashed represent a data-flow relation, and solid lines describe control-flow a relation. Our first observation reveals that all functions originate from the `main` function. Furthermore, the called functions correlate with the commits introducing the functionality as expected, e.g., Commit C7 calls the function `add` according to the graph. The `add`, `sub`, `listOperations`, and `readVaraiable` functions are helper functions serving the `main`. For this example, we can derive the commit that introduced the functionality to the program by looking at the control and data flow. For example, the only control and data flow revealed in the `add` function is the one between Commit C1, which is the initial commit, and Commit C2, which introduces the function. However, we can not conclude that this is true for all examples, since it highly depends on how the commits are organized. Another observation shows that the majority of interactions are located in the `main` function, which we expected, since the function orchestrates the programs behavior and is part in most of the committed changes.

INTERACTION PLOTS Next, we take a look at the interaction plots to expand our knowledge about the program. We start with the control-flow based plot in Figure 10. The first thing we noticed is that the *From* and *To* ratio is about even. That is because of the annotation strategy we used: For every changed that is wrapped within another, potentially an edge from the wrapper region to the inner region and one from the inner region to the wrapper region is created. However, the region with most interactions is Commit C1, since it introduced the initial program and all other changes are made within the initial code. The Commit C2, C3, C4, and C5 just add a function upon the initial commit, which is reflected by the count of interactions in the plot, whereby Commit C4 and C5 have more interactions, since they were modified later on.

Let us now look at Commit C6. Compared to the other commits the amount of interactions is significantly higher. The reason for that can

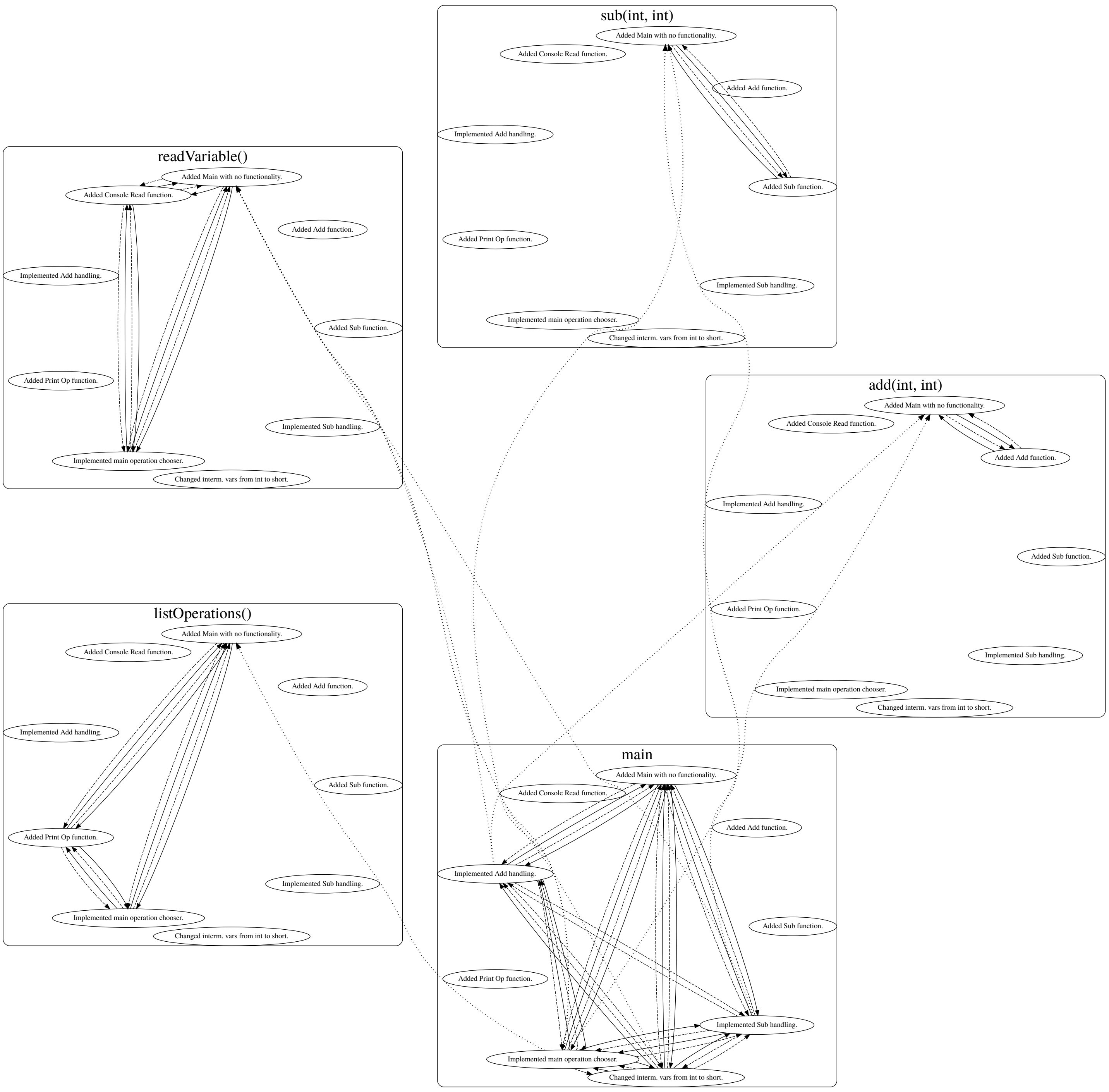
sub(int, int)
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.
readVariable()
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.
add(int, int)
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.
listOperations()
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.
main
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.

Figure 9: Overview graph of the sample application "Calculator".

be observed by taking a look at the changes introduced by the commit. We notice in addition to the changes stated in the commit message it also resolves a reference error and add changes the console output.

Lets focus on the uneven commits. For every interaction counted to *From* exists one that counts to *To*, again owed to the annotation strategy we use. Based on the plot we can reason whether Commit C9 influences Commit C7 and C8, because the interactions seem to originate from the one commit influencing the others. We take a look at the changes made by Commit C9, it changes the type of the variables used by both other commits leading to a possible error.

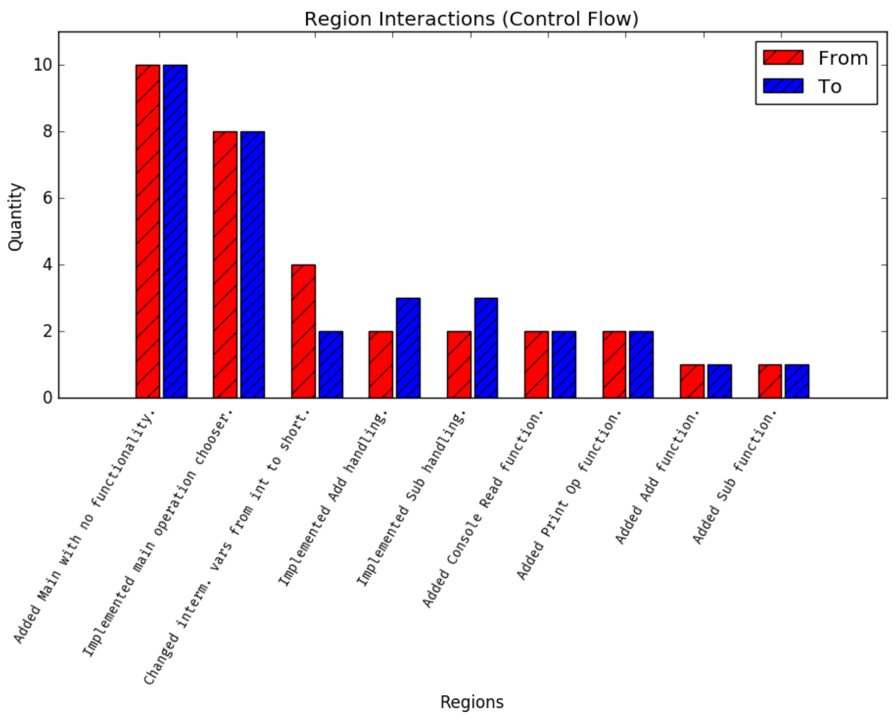

Figure 10: Interaction plot based on the control flow of the hand crafted example.

Next, we take a look at the interaction plot based on data flow depicted in Figure 11. We observer that Commit C1 has the highest interaction count, which was also the case in the plot based on control flow. However, if we take a look at the source code, we can not find evidence proving a data flow originating from Commit C1. Similar to the problem encountered previously the data is highly influenced by the annotation technique we are using. Since new changes might get wrapped in older ones causes the older commit to appear as a data source. This behavior can be observed for the variable `op`, which is not contained in Commit C1. However, Commit C1 encapsulates Commit C6, which does actually introduce the `op` variable. Hence, the data flow originating form `op` will count for Commit C1 as well.

Another thing we notice is that the *From* and *To* ratio is even for all regions in this example. One reason is that, similar to the control-flow plot, if the data-flow passes a region that is fully contained in another, then always two edges are created for each region involved. Furthermore, a certain imprecision results from a small implementation trick the analysis uses to circumvent the case where it cannot determine the successor memory access of a `store` instruction.

However, in case of our example program we can gain useful information from this illustration. In our example we can determine the impact a region has to the program by the amount of interactions. For example, Commit C1 introduces the `main` method and provides dependencies for the upcoming changes, Commit C6 changes the behavior of three different functions, and Commit C9 just changes the type from **`int`** to **`short`** in a single line, though it influencing the whole input behavior in three regions.

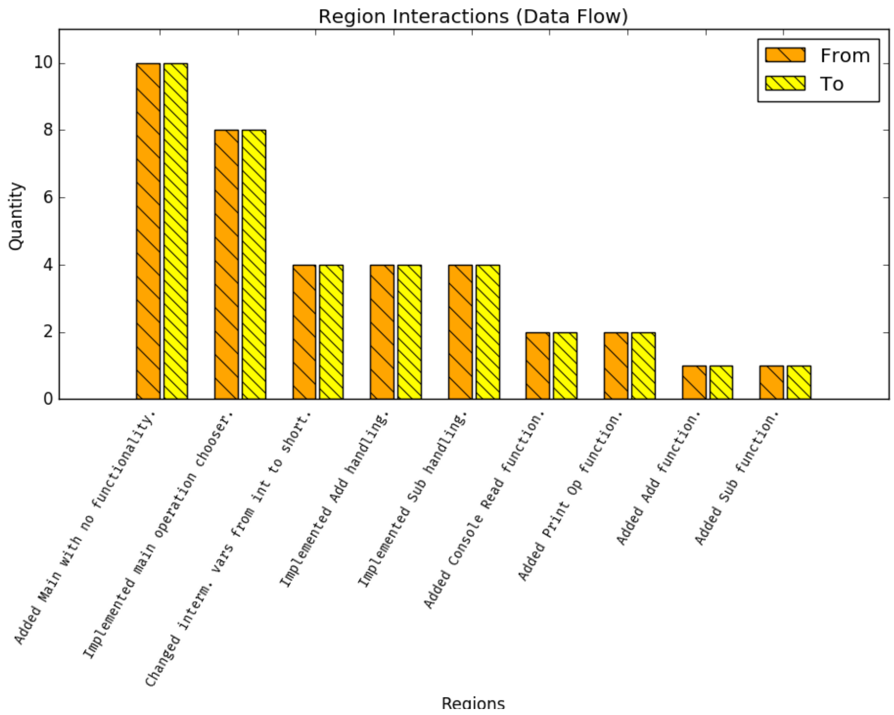

Figure 11: Interaction plot based on the data flow of the hand crafted example.

The interaction plots provide us with a general overview about how entangled certain commits are in a program. In our case we were able to identify commits introduces worthwhile to investigate.

CONTROL AND DATA FLOW OF THE MAIN FUNCTION We now focus on the graph plots especially on the one of the `main` function, which, as already noted, is responsible for the principle behavior of our program. The graph illustrating the results for the function is depicted in Figure 12. We notice that in most cases the data-flow and control-flow results are similar. This confirms the observation we made by looking at the interaction plots. However, the Commit C7 and C8 are operating on the same data, whereby they do not interact by their flow of control. What we observe here is an inaccuracy problem, the one we mentioned earlier. By looking at the changes introduced in both commits, the variables are initialized and used in the same commit region, therefore there should not be a data flow between those regions at all.

Aside the inaccuracy problem, in most cases the flow of data is depicted correctly. Now we take a look at a more problematic commit. The changes introduced in Commit C9 to the `main` function could lead to possible errors. For this example, we can identify affected regions by looking at the graph nodes targeted by the malicious commit.

The graph representation for the example offers a good view on how different commits are interacting with each other.

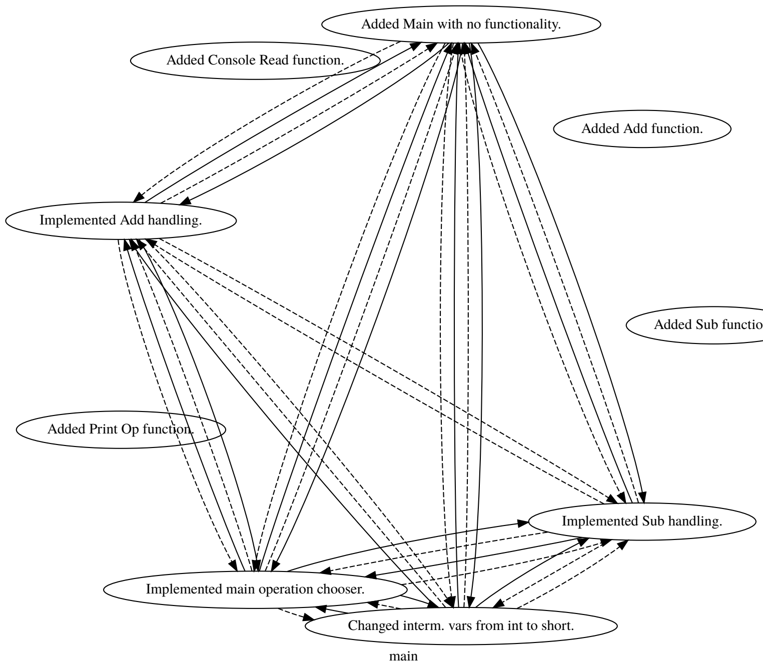
Added Main with no functionality.
Added Console Read function.
Added Add function.
Implemented Add handling.
Added Sub function.
Added Print Op function.
Implemented Sub handling.
Implemented main operation chooser.
Changed interm. vars from int to short.
main

Figure 12: Graph illustrating the analyses results for the `main` function.

4.4.1 *Addressing RQ1.1*

To address **RQ1.1** we focus on the control-flow based results of the `main` function in Figure 12. Regarding the graph, Commit C7 has control-flow relations with Commit C1, C6 and C9. We validate the these interactions by looking at the snippet depicted in Listing 19. The listing contains a cropped version of the calculator example including region annotations.

We can confirm the bidirectional interaction between Commit C7 and C1, since the control-flow passes from Commit C1, containing the whole program, to C7 and back to C1. This is also true for the interactions between Commit C7 and C6. Furthermore, we can validate the interaction from C9 towards C7, the control-flow passes C9 in Line 38 and enters Commit C7 in Line 42.

Hence, we can confirm **RQ1.1** that we are able detected interactions between regions based on LLVM-IR control-flow.

```
__REGION_START "C1"
...
__REGION_START "C6"
...
  __REGION_START "C9"
  short v1, v2, op = readVariable();
  __REGION_END "C9"
  switch (op) {
  __REGION_START "C7"
    case 0:
      printf("Var 1: ");
      v1 = readVariable();
      printf("Var 2: ");
      v2 = readVariable();
      printf("%d",add(v1, v2));
      break;
  __REGION_END "C7"
  __REGION_START "C8"
    case 1:
      printf("Var 1: ");
      v1 = readVariable();
      printf("Var 2: ");
      v2 = readVariable();
      printf("%d",sub(v1, v2));
      break;
  __REGION_END "C8"
...
__REGION_END "C6"
...
__REGION_END "C1"
```

Listing 19: Part of the `main` function from the hand-crafted example.

4.4.2 *Addressing RQ1.2*

Now we address **RQ1.2** by focusing on the data-flow based results of the `main` function for Commit C7. The commit has data-flow interactions with Commit C1, C6, C8, and C9 according to the graph depicted in Figure 12.

We can confirm that Commit C1 and C6 interact with Commit C7, since the instructions contained in Commit C7 are also contained in Commit C1 and C6. This is owed to the annotation technique we use, but, more importantly, we can identify data-flow relations for the variables `v1`, `v2`, and `op`, as we can see in Listing 19, that are responsible for the resulting edges. We can confirm a data-flow relation between Commit C7 and C9 based on the variables `v1` and `v2`. Furthermore, we can verify the interactions between C7 and C8 using the same variables. However, some edges result from overapproximation of the data-flow analysis.

Nevertheless, we detected data-flow interactions based on LLVM-IR, therefore we can answer **RQ1.2** positively.

4.4.3 *Addressing RQ2*

To address **RQ2** we take a look at the control-flow based interaction plot depicted in Figure 10, as we already observed, more relations are originating form Commit C9 than targeting it. We use this clue and explore the corresponding interactions in the graph of the `main` function depicted in Figure 12. The graph illustrates the interactions to three prior commits, whereby we will ignore Commit C1, since we try to focus on exceptional relations originating form Commit C9. Therefore, we further investigate the interactions with Commits C7 and C8. We conclude that Commit C9 has an impact on Commit C7 and C8, which we can verify by looking at the introduced changes in Listing 19. In Line 38 the type of the variables `v1` and `v2` was changed from **`int`** to **`short`** by Commit C9, however, the calculations introduced in Commit C7 and C8 operate still on **`int`**. This change has in our point of view a major impact on the behavior of the program.

Therefore, we are able to confirm **RQ2** that the impact of a change can be measured through the interactions to other regions.

4.5 EXPERIMENT 2 - GZIP

We now focus on analyzing the preexisting software gzip and use the results to address our research questions. The visualization for larger projects tend to be more complex regarding the amount of nodes and edges in the graph. Therefore, we aggregate the results based on the tags form the repository. By doing so we are able to reduce the amount of commit regions from 116 to 19. Furthermore, the project does not consist entirely of C source or header files, hence, another 6 commits are omitted. The downside of aggregating commits in the analysis is that we may miss interesting observations, the upside on this is that we can focus on certain commits in the repository. In our case we focus on the newer unversioned ones. We think analyzing them this is especially useful in order to identify possible unwanted interactions in the development process, e.g., before releasing a new version.

Nr.	Message
G1	* NEWS: Version 1.3.12 released.
G2	version 1.3.13
G3	version 1.4
G4	version 1.5
G5	version 1.6
G6	version 1.7
G7	version 1.8
G8	maint: change "time stamp" to "timestamp" globally
G9	gzip –no-name: avoid spurious warning
G10	gzip: drop mentions of Amiga, VMS
G11	gzip: fix some Y2038 etc. bugs
G12	gzip: minor time stamp cleanups'
G13	gzip: fix bug in unpack EOB check

Table 1: Table mapping regions to an abbreviated version.

TABLE OF COMMITS Table 1 contains all resulting regions gained from the analysis, as well as their abbreviations that we use to reference certain region.

In this experiment we focus directly on the research questions, since we use the same approach we did with the hand-crafted example. We closely observe the visual results and try to confirm our findings by looking at the changes introduced by the region.

4.5.1 *Addressing RQ1.1*

We further reduced the amount of regions by joining the commits from Region G1 to G7 together. To answer **RQ1.1** we focus on the interactions originating from single-commit regions. Therefore, the relation between condensed regions in not important in this section.

We now focus on the control-flow based analysis results of the `zip` function depicted in Figure 13. Based on the results depicted in the graph we can conclude Region G9 interacts with G7, G8, and G11. In order to verify the information we observed, we take a look at Listing 20 containing parts of the `zip` function including region annotations. We can confirm that Region G9 interacts with G7, since the instructions enclosed in Region G9 are also enclosed in Region G7, which results in a relation between these two regions. We can also verify the interaction from Region G9 to G8 and G11 by following the control flow originating from Region G9 towards them.

Hence, we can confirm **RQ1.1** that we can successfully detect commit interactions based on LLVM-IR control-flow.

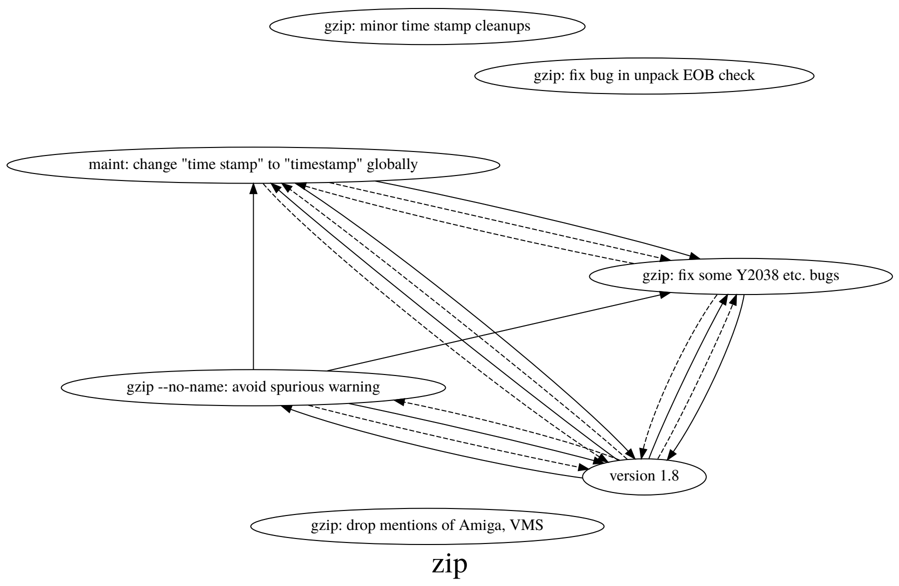
gzip: minor time stamp cleanups
gzip: fix bug in unpack EOB check
maint: change "time stamp" to "timestamp" globally
gzip: fix some Y2038 etc. bugs
gzip --no-name: avoid spurious warning
version 1.8
gzip: drop mentions of Amiga, VMS
zip

Figure 13: Graph representation of the analyses results for the `zip` function.

4.5.2 *Addressing RQ1.2*

Again we take a look at the results of the `zip` function depicted in Figure 13, but now we focus on the data flow. We take a look on the data flow originating from Region G11, which interacts with Region G7 and G8 based on the graph illustration. We verify our findings based on the source code depicted in Listing 20, by focusing on the `stamp` variable. In Line 65, 69, and 75 a value is stored to the variable in Region G11, the value is than loaded in Line 78 in Region G7. Therefore, we can confirm a data-flow interaction between Region G11 and G7.

However, we can not verify the data flow towards Region G8 based on the source code. If we investigate this issue further, we notice that Region G8 is wrapped in Region G11. In the perspective of the analysis both regions affect the data flow of the instruction shown in Line 73 of the source code. We will discuss this problem and a possible solution later in Chapter 5.

Nevertheless, we can identify interactions between commit regions based on LLVM-IR data flow. Hence, we can answer the **RQ1.2** positively, but the interactions found represent an overapproximation.

4.5.3 *Addressing RQ2*

In order to answer **RQ2** we take a look at the control flow based interaction plot depicted in Figure 14. We focus, like we did with the handcrafted example, on the regions with a high amount of interactions. We notice that the Region G10 has the highest amount of control-flow relations compared to the other single-commit regions. The commit

```
___REGION_START "G7"
...
___REGION_START "G9"
if (time_stamp.tv_nsec < 0)
  stamp = 0;
else if (0 < time_stamp.tv_sec && time_stamp.tv_sec <=
 ↪  0xffffffff)
  ___REGION_START "G11"
  ___REGION_END "G9"
  stamp = time_stamp.tv_sec;
else
{
  ___REGION_START "G8"
  warning ("file timestamp out of range for gzip
   ↪  format");
  ___REGION_END "G8"
  stamp = 0;
}
___REGION_END "G11"
put_long (stamp);
...
___REGION_END "G7"
```

Listing 20: Part of the `zip` function from gzip.

affects 5 files, whereby it introduces 18 new lines and removes 103. Compared to Region G8, altering 2 files with 3 additions and 3 deletions and G12, changing 1, file adding 21 lines and removing 19, the amount of changes made by Region G10 is significantly higher. However, as we showed in our first experiment, quantity of changes is not necessarily relevant when measuring the impact of these changes to the software. Furthermore, we can not argue based on the changed source code, since we do not know all the implications these changes have to the program. Therefore, we look up the official statement in the changelog[5] for the upcoming release published by Jim Meyering, which states:

> "Support for VMS and Amiga has been removed. It was not working anyway, and it reportedly caused file name glitches on MS-Windowsish platforms."

Based on our observations and the statement of the developers, can we confirm **RQ2** that the impact introduced by a change can be measured through its interactions to other commit regions.

5 *GNU gzip - News: gzip-1.9 released*. `https://savannah.gnu.org/forum/forum.php?forum_id=9049`. (Accessed on 28/01/2018)

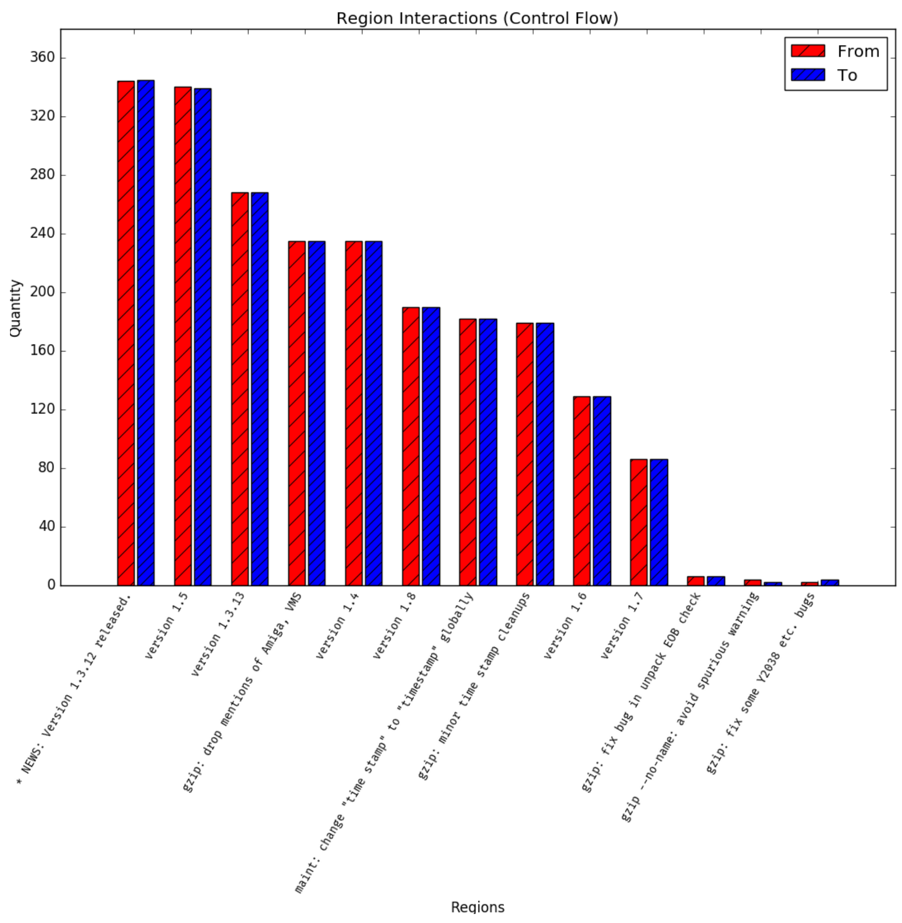

Figure 14: Interaction plot based on the control flow of gzip.

5 CONCLUSION

We conclude our work with a summary of our findings and contributions. Furthermore, we discuss similar efforts in the related work section and our ideas to further improve our work in the future.

5.1 SUMMARY

We presented an approach to detect changes based on the information provided by the repository. Based on these changes we organized the source code in regions using special keywords. Furthermore, we extended the C and C++ front-end clang with the ability to utilize these annotations and propagate the information forward to LLVM. We extended the VaRA framework to support our regions in its dataflow analysis and implemented an additional analysis, which is based on the control flow. Moreover, we used these analyses to analyze an example program we wrote and the open-source software gzip. We showed in the experiments, that we are able to detect interactions based on extracted commit information. These results can aid developer to better understand the implications and relations of certain changes. In addition, we showed that our approach is also applicable to software outside of our test environment.

Our goal was to provide developers with a tool that facilitates the maintenance of software. We introduced a technique that supports this procedure by enabling analyses during the compilation process. We think even more software projects written in different languages can benefit from this concept, since the analyses itself are language independent. Moreover, as the VaRA framework improves, additional analyses become available, which allows new researchers to answer different questions based on our regions.

5.2 RELATED WORK

To our best knowledge, up to now no publication worked on an approach to detect and extract change-regions using Git and LLVM.

SOURCE-CODE ANNOTATION However, Joy et. al. [9] propose a novel hybrid automated source-code annotated memory-leak detection approach for embedded system software. They use classical parsing techniques to identify basic blocks and for each identified block they insert one annotation in the beginning and one in the end. Unlike our annotations the ones used by Joy et. al. are implemented with

C macros, which, on disassembling, create basic blocks with unique identifiers. Based on the control flow between these basic blocks they extract leak candidate information in dynamic memory allocations.

They used a similar annotation technique in an earlier publication [10] where they performed a timing analysis for embedded software based on these annotations.

CHANGE IMPACT VISUALIZATION Gómez et. al. [6] preset a tool called *Torch* that characterizes changes based on information provided by an RCS. Their aim is to support the developers in comprehending changes, by providing visualizations and change summaries. Compared to our work they analyze changes based on the source code and information from repository alone. Their approach does not utilize data provided during the compilation process.

5.3 FUTURE WORK

SOURCE-CODE ANNOTATION We introduced an annotation technique that nest changes in regions. As we observed in the evaluation this might not be the best strategy for every type of analysis. We highly recommend to implement and evaluate another annotation method. In fact, we have already a working prototype, which annotates each change with a single region representing the latest commit that influences the corresponding part in the source code. We believe, this method would remove the inherited edges from regions, which could have a major effect on the results gained from the control-flow and data-flow analysis. We think, adapting the annotation procedure with respect to the analyses will improve the value of the generated results, significantly. Unfortunately, we had not the time to evaluate this approach in depth.

SEMATIC ENGINE AND AST Currently the parsing of annotations is located in the lexer component. However, this implementation causes errors regarding correct annotations in LLVM-IR, e.g., when using forward declarations. We believe that adding region information directly to the each AST node in the parse tree will eliminate such flaws. Furthermore, the current implementation lacks support for syntax and semantic checks. Adding region annotations to the semantic engine and the AST could prevent unwanted behavior, like a missing end-region annotation. However, the overhead to implement and maintain a solution that is located in all clang components is much higher than using the already existing implementation.

ANALYSES IMPROVEMENTS The analyses are currently only capable of analyzing single functions. Enhancing the analyses to be able to identify interactions throughout function boundaries would increase the accuracy of the results. Regarding the control-flow analysis, one could partially achieve this by preprocessing the interaction results for all functions and retrieve the information for every function call. However, solving this problem might raise further challenges, like how to handle recursion and indirect function calls. Nevertheless, we think its worth putting more effort in enhancing the analyses since one might gain results of higher quality.

VISUALIZATION TECHNIQUE The visualizations we used to present our results tend to get more complex with the size of the analyzed project. We solve this issue by reducing the participating regions. For example, in our gzip experiment we joined commits up to a certain version, in order to be able to investigate the changes that were applied afterwards. However, this also reduces the informative value of the results. A dynamic visualization providing the user with the capability to decide which information is relevant to him and which is not, would, in our eyes, increase the quality of the insights.

FRONT-END SUPPORT Furthermore, the analyses are performed language independent, adding change-region support to additional LLVM front-ends would increase the number of projects that can benefit from analyses provided by VaRA.

A APPENDIX: CODE

The following appendix contains example code and the commit history of the hand crafted example.

A.1 MODULE PASS EXAMPLE

Listing 21 shows the basic components needed to create a new LLVM module pass.

```cpp
#include "llvm/Pass.h"

class ExampleModulePass : public llvm::ModulePass {
        static char ID;
        ExampleModulePass() : ModulePass(ID) {}

        bool runOnModule(Module &M) override {
                /*
                ...
                */
                return false;
        }
};

char ExampleModulePass::ID = 42;
static RegisterPass<ExampleModulePass> X("example",
 ↪  "Example pass", false, false);
}
```

Listing 21: Example of a `ModulePass` in LLVM.

A.2 HAND CRAFTED EXAMPLE

Listing 22 shows the source code of the hand-crafted example we analyzed in Chapter 4. The program fulfills the task of a simple calculator. The Tables 2 and 3 contain the list of commits submitted during the development of the hand-crafted example. The first field is the number of the commit. In the second field the commit message is presented and the third field contains the changes introduced with the commit. Furthermore, they are ordered beginning by the latest commit down to the first one.

```c
#include <stdio.h>

int add (int a, int b) {
  return a + b;
}

int sub (int a, int b) {
  return a - b;
}

int readVariable() {
  int res;
  scanf("%d", &res);
  return res;
}

void listOperations(){
  printf("0 - Addition\n");
  printf("1 - Subtraction\n");
}

int main() {
  printf("Welcome to the Calculator!\n");

  listOperations();
  printf("Choose operation: ");
  short v1, v2, op = readVariable();
  switch (op) {
    case 0:
      printf("Var 1: ");
      v1 = readVariable();
      printf("Var 2: ");
      v2 = readVariable();
      printf("%d",add(v1, v2));
      break;
    case 1:
      printf("Var 1: ");
      v1 = readVariable();
      printf("Var 2: ");
      v2 = readVariable();
      printf("%d",sub(v1, v2));
      break;
    default:
      printf("Unknown operation: %d", op);
  }
  return 0;
}
```

Listing 22: Implementation of the hand-crafted example used in the evaluation.

Nr.	Message	Diff
C9	Changed interm. vars from int to short.	`@@ -24,7 +24,7 @@`
		`int main() {`
		`listOperations();`
		`printf("Choose operation: ");`
		`-  int v1, v2, op = readVariable();`
		`+  short v1, v2, op = readVariable();`
		`switch (op) {`
		`case 0:`
		`printf("Var 1: ");`
C8	Implemented Sub handling.	`@@ -33,6 +33,13 @@`
		`int main() {`
		`v2 = readVariable();`
		`printf("%d",add(v1, v2));`
		`break;`
		`+    case 1:`
		`+      printf("Var 1: ");`
		`+      v1 = readVariable();`
		`+      printf("Var 2: ");`
		`+      v2 = readVariable();`
		`+      printf("%d",sub(v1, v2));`
		`+      break;`
		`default:`
		`printf("Unknown operation: %d", op);`
		`}`
C7	Implemented Add handling.	`@@ -26,6 +26,13 @@`
		`int main() {`
		`printf("Choose operation: ");`
		`int v1, v2, op = readVariable();`
		`switch (op) {`
		`+    case 0:`
		`+      printf("Var 1: ");`
		`+      v1 = readVariable();`
		`+      printf("Var 2: ");`
		`+      v2 = readVariable();`
		`+      printf("%d",add(v1, v2));`
		`+      break;`
		`default:`
		`printf("Unknown operation: %d", op);`
		`}`
C6	Implemented main operation chooser.	`@@ -10,16 +10,24 @@`
		`int sub (int a, int b) {`
		`int readVariable() {`
		`int res;`
		`-  scanf("%d", res);`
		`+  scanf("%d", &res);`
		`return res;`
		`}`
		`void listOperations(){`
		`-  printf("0 - Addition");`
		`-  printf("1 - Substraction");`
		`+  printf("0 - Addition\n");`
		`+  printf("1 - Substraction\n");`
		`}`
		`int main() {`
		`printf("Welcome to the Calculator!\n");`
		`+  listOperations();`
		`+  printf("Choose operation: ");`
		`+  int v1, v2, op = readVariable();`
		`+  switch (op) {`
		`+    default:`
		`+      printf("Unknown operation: %d", op);`
		`+  }`
		`return 0;`
		`}`
C5	Added Print Op function.	`@@ -14,6 +14,11 @@`
		`int readVariable() {`
		`return res;`
		`}`
		`+ void listOperations(){`
		`+  printf("0 - Addition");`
		`+  printf("1 - Substraction");`
		`+ }`
		`int main() {`
		`printf("Welcome to the Calculator!\n");`
		`return 0;`

Table 2: List of commits 9 to 5 from the hand-crafted example.

Nr.	Message	Diff

C4	Added Console Read function.	`@@ -8,8 +8,13 @@`
		`int sub (int a, int b) {`
		`return a - b;`
		`}`
		
		`+ int readVariable() {`
		`+  int res;`
		`+  scanf("%d", res);`
		`+  return res;`
		`+ }`
		
		`int main() {`
		`printf("Welcome to the Calculator!\n");`
		`return 0;`
		`}`

C3	Added Sub function.	`@@ -4,6 +4,10 @@`
		`int add (int a, int b) {`
		`return a + b;`
		`}`
		
		`+ int sub (int a, int b) {`
		`+  return a - b;`
		`+ }`
		`int main() {`
		`printf("Welcome to the Calculator!\n");`

C2	Added Add function.	`@@ -1,6 +1,11 @@`
		`#include <stdio.h>`
		
		`+ int add (int a, int b) {`
		`+  return a + b;`
		`+ }`
		`int main() {`
		`printf("Welcome to the Calculator!\n");`
		`return 0;`
		`}`

C1	Added Main with no functionality.	`@@ -0,0 +1,6 @@`
		`+ #include <stdio.h>`
		`+ int main() {`
		`+  printf("Welcome to the Calculator!\n");`
		`+  return 0;`
		`+ }`

Table 3: List of commits 4 to 1 from the hand-crafted example.

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[2] Robert S. Arnold. *Software Change Impact Analysis*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1996. ISBN: 0818673842.

[3] Edsger W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: http://doi.acm.org/10.1145/360933.360975.

[4] Institute of Electrical, Electronics Engineers, International Organization for Standardization, and International Electrotechnical Commission. *Norma ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*. IEEE std. IEEE, 2006. ISBN: 9780738149608.

[5] Brian Fahs. "SPEDI: Static Patch Extraction and Dynamic Insertion." In: (Jan. 2008).

[6] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. "Visually characterizing source code changes." In: *Sci. Comput. Program.* 98 (2015), pp. 376–393.

[7] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[8] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization, Feb. 2012, 1338 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[9] M. M. Joy, W. Mueller, and F. J. Rammig. "Source Code Annotated Memory Leak Detection for Soft Real Time Embedded Systems with Resource Constraints." In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing* (2014), pp. 166–172.

[10] Mabel M. Joy, Markus Becker, Wolfgang Müller, and Emi Mathews. "Automated source code annotation for timing analysis of embedded software." In: *2012 18th International Conference on Advanced Computing and Communications (ADCOM)* (2012), pp. 12–18.

[11] Chris Lattner. *The Architecture of Open Source Applications: LLVM*. http://aosabook.org/en/llvm.html. (Accessed on 10/02/2017). 2015.

[12] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, 2004.

[13] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.", 2012.

[14] Torben Ægidius Mogensen. *Basics of Compiler Design*. Torben Ægidius Mogensen, 2009.

[15] Florian Sattler. "A Variability-Aware Feature-Region Analyzer in LLVM." MA thesis. University of Passau, 2017.

[16] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-flow Analysis in LLVM." In: *Proceedings of the 25th International Conference on Compiler Construction*. CC 2016. Barcelona, Spain: ACM, 2016, pp. 265–266. ISBN: 978-1-4503-4241-4. DOI: 10.1145/2892208.2892235. URL: http://doi.acm.org/10.1145/2892208.2892235.

DECLARATION

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, Germany, February 7, 2018

Florian Niederhuber