UNIVERSITÄT PASSAU

Master Thesis in Computer Science

Enhancing Program Analysis with Git Metadata in VaRA

Julian Breiteneicher

2019-09-02

Supervisor: Prof. Dr.-Ing. Sven Apel
(Chair of Software Engineering I)

2nd corrector: Prof. Dr. Gordon Fraser
(Chair of Software Engineering II)

Advisor: Florian Sattler, M.Sc.
(Chair of Software Engineering I)

Breiteneicher, Julian:
Enhancing Program Analysis with Git Metadata in VaRA
Master Thesis, University of Passau, 2019.

Abstract

As the development of a software project progresses, it often becomes larger and more complex. Each new change that gets introduced into the project can have unintended interactions with different parts of the code. This can lead to bugs that are difficult to spot. To this end, the LLVM-based framework VaRA can perform control-flow and data-flow analyses to detect interactions between the commits of a software project. This can aid software engineers during development to assess the potential impact of a new change or during debugging. It can also be used by researchers to gain insights into how projects evolve over time and how changes interact with each other.

However, precise data-flow analyses are very expensive and can produce a lot of results. This makes it hard to use for certain use cases. During development, for example, long analysis times might be undesirable. Large result sizes can make it difficult to identify interesting findings, like potential bugs.

In order to improve VaRA's analyses in this regard, we work on reducing the number of false-positive results. Additionally, we integrate filter mechanisms in VaRA that allow the user to fine tune the analyses by filtering out unimportant changes, thereby speeding up the commit-interaction analysis. We show the impact of our approaches on a small, contrived software project and two real-world projects.

Contents

List of Figures

List of Listings

1 Introduction

Many software projects keep getting larger and more complex. For example, in 2017 alone, the size of the linux kernel increased by over 2.5 million lines of code [Lar18]. If a programmer introduces a new change it can be very hard to predict its impact on the functionality of the program, especially if multiple developers are involved in the project at the same time. If an interaction between different parts of the program causes a bug, it can be very hard to find the cause of that bug.

The LLVM-based framework VaRA (Variability-aware Region Analyzer) offers functionality to help the programmer analyze interactions between different changes. If a project's source code is stored in a Git repository, we can use the repository to gather information about which changes were introduced by the commits of the project. We can then use VaRA to find control- and data-flow interactions between the different commits of the analyzed project. However, for large projects this analysis can take a very long time, which makes it difficult to use in certain use-cases, like during development. Another potential difficulty is the size of analysis results. They can quickly become too large to be easily interpreted by the user to gain useful insights into the project. We believe that increasing the speed of VaRA's commit interaction analysis would make it more useful by enabling use-cases that were previously unfeasible, for example, analyzing very large software projects in a reasonable amount of time or helping software engineers during development or debugging. Additionally, limiting the size of the analysis result by not showing interactions that are unlikely to be interesting or useful could also make it easier to interpret the results of the analysis.

1.1. Goals

Our goal is to improve VaRA's commit interaction analysis, which can be used by researchers as well as software developers to detect control-flow and data-flow interactions between commits and to estimate the impact of a change. We aim to reduce the analysis's false positives to make it more precise. Furthermore, we also intend to increase the speed of the analysis and allow the user to remove uninteresting interactions from the analysis result to make it easier to interpret.

1.2. Contributions

In this thesis, we enhance VaRA's commit interaction analysis with two main contributions. We introduce BlameRegions as a new region type in VaRA to increase the accuracy of the analysis. Furthermore, we provide mechanisms to increase the speed of VaRA's analysis and to filter out uninteresting interactions from the analysis result to make it easier to interpret and to make its size more manageable. In order to achieve these goals, we added a Git metadata interface to VaRA that can be used to retrieve additional information from the Git repository of an analyzed software project.

1.3. Overview

In Chapter 2, we introduce the necessary background of this thesis. We describe the basics of control-flow and data-flow analysis, the Git revision control system and the LLVM compiler framework. We also introduce the reader to the VaRA analysis framework and the accompanying VaRA-Tool-Suite application. In Chapter 3, we present our improvements of VaRA's commit interaction analysis. This includes the implementation of a Git metadata interface, the development of the new BlameRegion type and the Interaction Filters that can be used to filter the interactions of VaRA's interaction analysis. These approaches are evaluated in Chapter 4 by performing VaRA's analysis on a set of case studies. In the last chapter, we summarize our work and present ideas for future improvements.

2 Background

In this chapter, we introduce control-flow and data-flow analysis. We then explain the revision control system Git and the LLVM compiler framework; particularly we focus on the C/C++ frontend clang and LLVM's intermediate representation. At the end, we introduce the LLVM-based analysis framework VaRA.

2.1. Control-Flow and Data-Flow Analysis

Static analyses can be used for many applications, such as program optimization, automatic code review or software verification. This section introduces the basics of control-flow and data-flow analysis. Both of these are static program analysis techniques, which means that the program is analyzed without being executed.

2.1.1. Control-Flow Analysis

The goal of control-flow analysis is to determine the order in which the operations of a program are executed. When running the program, (a subset of) these statements are executed in a certain order. Every sequence of statements, which are executed directly after each other, form a so-called *execution path*. By using control-flow analysis, we can find the set of possible execution paths of a program [DGS97].

Control-Flow Graph

The control-flow graph (CFG) of a program's method is a directed graph that represents all possible execution paths within that method. The nodes of the graph are the method's basic blocks (BB). A basic block is a sequence of instructions where the control-flow always enters the block at its first instruction—which is called *leader*—and exits it at its last instruction—called *terminator*. When we separate the instructions into two different kinds, where one kind can change the control-flow of the program and the other one can not, we can look at the definition of BBs in another way: A BB is a sequence of instructions where only the terminator is able to change the control-flow of the program, while all other instructions are not. If a BB can be executed directly after another BB, then the CFG contains a (control-flow) edge from the BB that is executed first to the BB that can be executed next. It is also possible that there are multiple BBs that could potentially be executed after another BB. In this case the BB has multiple outgoing edges. Similarly, a BB can also have multiple ingoing edges.

Figure 2.1 shows an example CFG. The control flow of this example begins

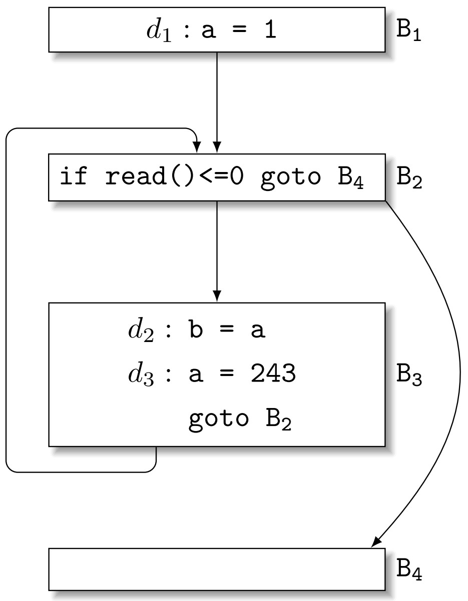


Fig. 2.1. Example of a control-flow graph (taken from [Aho+06])

with the basic block B_1, which defines the variable a. Afterwards, the control-flow continues at B_2 where the `read()` method is called. Depending on whether its return code is greater than zero or not, we continue the execution of the program at B_3 or B_4. If the program jumps to B_4, this particular BB is executed and the program ends. If we jump to B_3, however, the variables b and a are defined and we jump back to B_2, which gets executed again. This small example illustrates, for instance, how loops can be represented in the CFG [Aho+06].

2.1.2. Data-Flow Analysis

Data-flow analysis aims to obtain information about how data flows along the execution paths of a program. In this section, we explain the fundamentals of how data-flow analysis works.

In the previous section, we have already described that a running program can be viewed as a series of operations. These operations generally perform a transformation of the *program state*—the set of the values of all variables of the program. Each operation takes an input state—the program state before the operation is performed—and transforms it to an output state—after the operation is executed. In a data-flow analysis, we map a *data-flow value* to every program point, which represents all program states that are possible at that point in the program's execution. For each point in a program's execution path, the analysis takes the data-flow value at a specific point and evaluates how the next instruction modifies that value, which results in a new data-flow value after that instruction. By propagating that information along the execution paths of the program in this way, the analysis can collect knowledge about the data-flow of all points of the program [Aho+06].

IN and OUT Sets

For every statement s in a program, we can define the set of data-flow values before the execution of that statement as $\text{IN}[s]$ and after that statement as $\text{OUT}[s]$. There are two kinds of constraints on the IN and OUT sets of s. The first constraint is based on the semantics of the statement, while the second one is based on the control-flow of the program.

Transfer Function

The first kind of constraint on the IN and OUT sets of a statement s is based on its semantics, since the semantics of a statement determines how its IN set is transformed into the OUT set. The change of the data-flow values by a statement is denoted by the *transfer function* f_s. By evaluating the transfer function for every instruction, we can propagate the data-flow information along the execution paths to gradually extend our knowledge about the data-flow of the program. Determined by whether we want to propagate the data-flow information forward in the direction of the execution path or backward (depending on the application), the transfer function takes the IN or the OUT set as input and returns the other one [Aho+06]:

$$\begin{aligned} \text{OUT}[s] &= f_s(\text{IN}[s]) && \text{(forward propagation)} \\ \text{IN}[s] &= f_s(\text{OUT}[s]) && \text{(backward propagation)} \end{aligned}$$

The transfer function is a very central part of a data-flow analysis because it does the actual work. By analysing the data-flow information at one point in the program it generates information about another point, which extends our knowledge about the program's data-flow. We can define the transfer function as follows:

$$f(x) = gen \cup (x - kill)$$

It takes, for example, the IN set of statement s (forward propagation) and calculates OUT by using the *gen* and *kill* sets of s. *gen* contains the information that is generated by s, while *kill* contains the information that is destroyed by s. So by removing the killed information from the IN set of s and adding the information that is generated by s, we get the OUT set of s and expand our knowledge about the data-flow to another point in the program.

Data-Flow with Multiple Statements

So far, we have looked at how the data-flow values are changed by a single statement. When we want to determine how the values are changed between multiple statements, we need to consider the second kind of constrains on the IN and OUT sets, which are based on the control-flow of the program.

Within a basic block, this is simple. For a basic block with a sequence of statements $s_1, s_2, \dots, s_n$, the IN set of a statement is given by the OUT set of the statement that precedes it:

$$\text{IN}[s_{i+1}] = \text{OUT}[s_i], \text{for } i \in \{1, 2, \dots, n-1\}$$

To calculate the IN set of the first statement we simply define a virtual *entry* statement as predecessor of s_1 with $\text{OUT}[\textit{entry}] := \emptyset$ which leads to $\text{IN}[s_1] = \emptyset$.

Data-flow between multiple basic blocks is more complicated. A BB might have multiple predecessors, so we need to find a way to "combine" the data-flow values from all preceding blocks.

To make this task easier, from now on, we look at IN and OUT sets of whole basic blocks instead of individual statements. We denote this as $\text{IN}[B]$ or $\text{OUT}[B]$ with B being the basic block in question. This is possible because we know that in a basic block, by definition, all its statements are always executed from the first one to the last one, since the block does not contain any branching instructions (except the last one). For a single BB B with statements $s_1, s_2, \dots, s_n$ the OUT set is defined as

$$\begin{aligned} \text{OUT}[B] = f_B(\text{IN}[B]), \text{ with } \text{IN}[B] &= \text{IN}[s_1], \\ \text{OUT}[B] &= \text{OUT}[s_n], \text{and} \\ f_B &= f_{s_n} \circ \dots \circ fs_2 \circ fs_1. \end{aligned}$$

To combine the IN or OUT sets of multiple predecessor BBs, we introduce a new operator which we call the *join* operator. Sometimes it is also called the *meet* operator. Depending on the direction in which we want to propagate information, it combines the OUT sets of multiple predecessor blocks to calculate the IN set of a BB, or vice versa. In case we want to propagate the data-flow values forward (for example if we want to analyze which values *may* be assigned to a variable), we could use the set union operation as *join* operator and calculate the IN set of a BB as follows:

$$\text{IN}[B] = \bigcup_{P \text{ a predecessor of B}} \text{OUT}[P]$$

In case we want to use backward propagation, the OUT set could be calculated as follows [Aho+06]:

$$\begin{aligned} \text{IN}[B] &= f_B(\text{OUT}[B]) \\ \text{OUT}[B] &= \bigcup_{S \text{ a successor of B}} \text{IN}[S] \end{aligned}$$

Taint Analysis

Taint analysis is a concrete application of the general data-flow analysis concept described previously. It tracks how variables that have been tainted by a source propagate through the program's execution path and determines if they reach a sink. Sinks and sources can be functions or instructions depending on the use-case. Taint analysis is very frequently used in security research to find potential vulnerabilities in programs. For example, it can be used to determine if untrusted user input can reach certain critical functions. Functions that read untrusted input data are regarded as sources and all variables these functions write to are marked as tainted. Critical functions or instructions are called sinks. For example, the `printf()` method could be a potential sink because it might be used in a format string attack or leak secret information. The taint analysis propagates the tainted variables through the program. Every time such a variable reaches a sink, the analysis has found a potential vulnerability. However, the analysis does not only track variables that are tainted by a source. A tainted variable can also propagate its taint to another variable that was previously untainted. For example, if the value of a tainted variable a affects another variable b (for example, if a string a is appended to a string b), the b gets tainted as well. This way, the analysis can also find sources that can indirectly reach sinks.[1]

[1]The advantages and disadvantages of implicit flow tracking are described in [Kin+08].

Taint analysis can also be used for other use-cases, for examplei, to find code interactions. If we imagine, for example, two code regions that are linked to two different features A and B of a program then we could use a taint analysis to find interactions between the two features by defining the instructions of these two regions as sources and targets. If a taint can be propagated from a source to a target then we found a data-flow relation between these two features of the program [SHB19].

2.2. Git

In this section, we introduce Git, a distributed revision control system. According to multiple surveys, it is now the most used revision control system [Ske14; Sta18]. Git enables developers to efficiently work on a software project and solves or eases many challenges of the development process.

When developing a software without a revision control system, every time a developer makes a change to the project (e.g., by adding or deleting some code), the previous state of the project is lost. There is no stored history of the projects development. When a bug is introduced into the project it is very hard to undo the change and go back to the previous version. Git solves this problem by always storing the complete history of the project's development. Every time a change is introduced to the software, Git adds that change to its storage while still keeping the previous state so that a developer can always go back to any previous state.

Another big challenge of developing a software without a revision control system is to integrate the changes of multiple developers into a single "official" version of the project. This is not an issue if only one developer works on the project at a time. But often this is not a desirable solution since it is very inefficient, especially for bigger projects. When multiple developers work on the project simultaneously on their own copy of the project, then they would constantly have to manually integrate their changes into a single version, which is very error-prone and time-consuming, especially if the developers changes conflict with each other. Git makes this process a lot easier. It can be used to create a central instance (or *repository*) of the project with its current state. The developers work on their own copies of that repository. After they finished working on a particular code change, they can use Git to incorporate that change into the central repository. If the new code change conflicts with the change of another developer, Git also aids the developer in resolving that conflict.

2.2.1. Commits

A Git commit can be seen as a snapshot of a software project at a specific point in time. When a developer finishes working on a particular code change, he or she can use Git to make a new commit. The commit incorporates the code change into its repository and persists the new state of the software project at that point in time. Other developers can base their work on that state and make new commits with their code changes (in this case the old commit is the parent of the new commit). The evolution of a software project over time is reflected by Git by a series of commits. A developer can look at the code changes of the individual commits and even revert the project to a previous state (by going back to the commit of that point in time).

Internally, Git stores a commit as a commit object, which can be uniquely identified by its SHA1 checksum (also called the commit hash). Listing 2.1 shows an example of the content of such an object. The first line contains a reference to the tree of the commit (also in the form of a hash). The tree contains the already mentioned snapshot of all files within the repository at the time of the commit. If a commit has parent commits, their hashes are also stored in the object. Lines 3 and 4 contain author and committer information (name, email address, and time). The author of a commit is the person that made the source code change (the person that “created” the commit) while the committer is the person that added the commit to the repository. The last part of the commit object is the commit message, a textual description of the commits content.

```
1  tree 643c4d078e2ec99c79089b17e1486f7497d08416
2  parent cea583a77aeb7d3b1cf4cd216e6d843ffc5f1c8f
3  author Julian Breiteneicher <julian@example.com> 1562343972
    ↪  +0200
4  committer Florian Sattler <florian@example.com> 1562343972
    ↪  +0200
5
6  Commit message
```

Lst. 2.1. Example of a Git commit object

2.2.2. Blame

As a software project progresses, different authors add commits, which add, modify or delete different files of the project. If we want to know which authors created which parts of the project's files and when, we can use a feature of Git called *blame*. For each line of a file, Git blame can show which commit last modified that line. Listing 2.2 depicts an example of a blame output. It shows, for instance, the line number 113 was last modified by the commit c2e86addb86 and the author Stephen Boyd in the year 2011. Consecutive lines with the same commit hash can be grouped together to a

```
112    [...]
113    c2e86addb86 (Stephen Boyd      2011-03-22 00:51:05 -0700 113) void NORETURN usage(const char *err)
114    39a3f5ea7c0 (Petr Baudis       2006-06-24 04:34:38 +0200 114) {
115    64b1cb74f83 (Jonathan Nieder   2009-11-09 09:05:02 -0600 115)     usagef("%s", err);
116    39a3f5ea7c0 (Petr Baudis       2006-06-24 04:34:38 +0200 116) }
117    39a3f5ea7c0 (Petr Baudis       2006-06-24 04:34:38 +0200 117)
118    [...]
```

Lst. 2.2. Git blame output of a file from the git repository

single *hunk*. In the current example, this means that lines 113 to 115 each belong to a hunk with a single line and lines 116 to 117 belong to another hunk.

2.3. LLVM

In this section, we introduce LLVM, a modular and reusable compiler framework. We describe LLVM's basic architecture in Section 2.3.1. In the remaining sections, we explain some of its components—the frontend clang, the intermediate representation, and the pass framework—in more detail.

2.3.1. Compiler Architecture

The LLVM compiler consists of three major parts (shown in Figure 2.2): frontend, optimizer, and backend. The frontend takes the input source code,

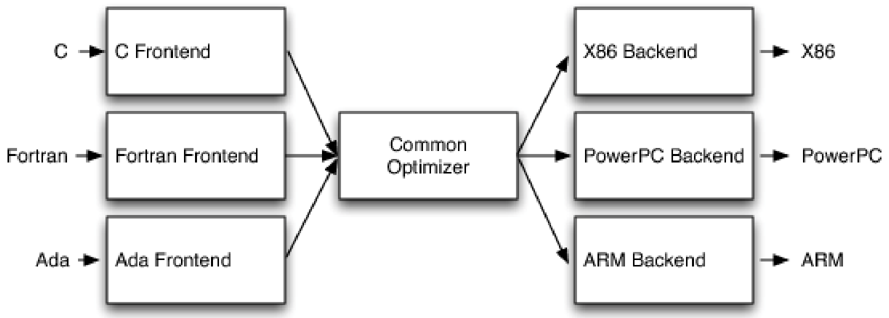


Fig. 2.2. Architecture of the LLVM compiler (taken from [Lat12])

parses it and checks it for errors. The parsing is usually done by first converting the source code to an abstract syntax tree (AST). An AST is a data structure that represents the code's hierarchical syntactic structure. [Aho+06] After the AST has been constructed and the code has been checked for errors like syntax or type errors, it is then converted into LLVM's intermediate code representation (LLVM-IR) and passed to the optimizer, the compiler's second major part [Lat12].

LLVM-IR is a typed, low-level code representation that is designed to be human readable as well as to enable efficient code analysis and transformation. All LLVM frontends transform the input source code into IR code. This common language enables the remaining components (optimizer and backend) to work independently of the input language. The frontend's IR code is passed to the optimizer which in turn passes it through one or more of its optimization passes. Every pass analyzes the IR code it receives and optionally optimizes it. Afterwards, it outputs (potentially modified) IR code again which can be given to the next pass. Different passes can perform different kinds of optimizations, for example, loop unrolling or dead code elimination. After the last optimization pass has run, its output is passed to the backend.

The different LLVM backends take IR code and convert it to machine code for the target architecture, e.g., X86 or ARM. In this step, the backend can perform target-specific optimizations, for example, by optimizing the register allocation or by selecting different instructions depending on the target platform [LLV19a].

2.3.2. LLVM Frontend clang

In this section, we look at clang in more detail, LLVM's frontend for the C language family (e.g., C, C++, Objective-C, Objective-C++). Clang is comprised of multiple stages which we describe in the following:

Driver The driver is called when clang is launched via the `clang` binary. It starts and controls the different tools, that are used during the compilation process, e.g., the compiler, the assembler, and the linker.

Preprocessing During this stage, the input source code is tokenized and the preprocessor statements (such as macros, `#ifdefs`, and `#includes`) are processed.

Parsing and Semantic Analysis This stage takes the output of the previous stage and parses the code into an AST. It then performs semantic analysis and error checking on the AST and outputs warnings or error messages if necessary.

Code Generation The last stage of the frontend is the generation of LLVM IR code based on the AST. This code is then passed to LLVM's optimizer pipeline and then to the backend for machine code generation and linking.

2.3.3. LLVM-IR

LLVM-IR is a static-single assignment (SSA) based language, which means that every variable has to be defined before it can be used and that a variable cannot be assigned more than once. Its virtual instruction set is similar to the RISC instruction set. It consists of a linear sequence of instructions that are in three address form, which means that they take some inputs and write a result into another register. LLVM-IR has a simple, strong type system which makes it easier for the compiler to optimize it.

Listing 2.3 shows a small example with the corresponding C++ source code. The first line of the IR code defines a method `main` with a 32-bit integer return

```
1   int main() {
2
3
4     int x;
5     int y;
6
7     x = 30;
8     y = 12;
9
10
11    return x+y;
12
13  }
```

(a) C++

```
1   define dso_local i32 @main() #0 {
2   entry:
3     %retval = alloca i32, align 4
4     %x = alloca i32, align 4
5     %y = alloca i32, align 4
6     store i32 0, i32* %retval, align 4
7     store i32 30, i32* %x, align 4
8     store i32 12, i32* %y, align 4
9     %0 = load i32, i32* %x, align 4
10    %1 = load i32, i32* %y, align 4
11    %add = add nsw i32 %0, %1
12    ret i32 %add
13  }
```

(b) LLVM-IR

Lst. 2.3. LLVM-IR example with corresponding C++ code

value (“`i32`”). Line 2 defines the single basic block `entry` of this method. The next three lines allocate memory for the `x` and `y` variables and the method’s return value. Lines 6 to 8 initialize the return value with `0`, and the `x` and `y` variables with `30` and `12`. Next, the values of the `x` and `y` variables are loaded (lines 9 and 10) and added (line 11). The result is stored in the `add` variable. Lastly, the content of the `add` variable is returned by the method [Lat12; LLV19a].

IR Metadata

LLVM-IR allows the developer to convey additional information by attaching metadata to IR instruction. One important use-case of this is to provide debugging information. If the compiler generates a warning or error message, the metadata can be used to map a location in the IR code to the original source code. There are two different kinds of metadata elements: strings and nodes. Both always start with an exclamation mark (“!”). An example for a metadata string is “`!test`” (including double quotes). A metadata node can have multiple values of arbitrary type. A named metadata can reference multiple metadata nodes that are stored separately in a special section of the

IR code. Listing 2.4 shows a small example. The named metadata "MDName"

```
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4, !MDName !23
  [...]
}
!23 = !{!"Very", !"important", !"metadata"}
```

Lst. 2.4. Example of Metadata Nodes in IR Code

references the metadata node 23 which contains the metadata strings "Very", "important" and "metadata" [LLV19a].

2.3.4. LLVM Pass Framework

In the overview section, we have already briefly described what a LLVM pass is. It receives IR code as input and, depending on the type of the pass, performs some analysis on the code or transforms it. Afterwards, it outputs the (possibly modified) IR code, which then gets passed to another optimization pass or to the backend. There are six different kinds of passes in LLVM:

- ModulePass
- CallGraphSCCPass
- FunctionPass
- LoopPass
- RegionPass
- BasicBlockPass

A ModulePass is the most generic type of pass. It receives the whole module as input and can perform arbitrary transformations. In C++, a module corresponds to a single translation unit (a single source file after all preprocessor statements have been processed). The CallGraphSCCPass is used to traverse the program's call graph from bottom to top. FunctionPasses receive the individual functions of the program as input, LoopPasses work on loops within functions, RegionPasses on single entry single exit regions within functions and BasicBlockPasses work on the individual basic blocks of a function.

In order to implement a new pass, one has to subclass the appropriate class, e.g., FunctionPass to implement a new function pass. The new pass can overload three virtual methods of the parent class. When the pass modified the program then they should return `true`, otherwise `false`. The most important method that has to be overloaded is the `runOnFunction(Function)` method. It can perform its analysis or transformation on each function of the program, which it receives as input. Additionally, the pass can overload the methods `doInitialization(Module)` and `doFinalization(Module)`, which can be used for initialization before the `runOnFunction` method is called or for cleanup afterwards. Finally, the newly created pass has to be registered with LLVM's PassManager. The PassManager is responsible to schedule the execution of the passes in such a way that they run efficiently and that all their requirements are fulfilled [LLV19b].

2.4. VaRA

In this section, we introduce VaRA, the "Variability-aware Region Analyzer", a software analysis framework built on top of LLVM. It is able to find different kinds of regions in the analyzed code, for example regions that are related to a specific run-time feature of the software or to a specific commit. Additionally, it can perform control-flow and data-flow analyses on the detected regions. The supported regions all share a common interface and their detection is separated from the analysis. This makes it easy for developers to add support for new region types which can be used by the existing analyses or to add a new analysis that can use existing regions [Sat17].

2.4.1. Regions

One of VaRA's central concepts is notion of an `IRegion`, which is defined as a section of the code that is of specific interest for an analysis. One example for an `IRegion` is the `FeatureRegion` which is defined as a section of the code that is related to a specific run-time feature of the software, e.g., logging. This concept is expressed in VaRA by the abstract class `vara::IRegion`, from which all supported region types are inherited. This class is used as a common type between the different region types by VaRA's analyses. It offers certain often-used convenience methods, for example to calculate the unique UUID of the region [Sat17].

2.4.2. Region Aggregation

Instead of inheriting from `vara::IRegion` directly, a region can alternatively inherit from the `llvm::IRegionAggregate` class. As already mentioned, a region is a single section of the code. By default, this includes only sections of code withing a basic block. However, in some use-cases, we may be interested in viewing multiple regions with a common property as a single entity, even though they span across multiple basic blocks, for example, if multiple regions belong to the same feature. This can be done via the `IRegionAggregate` class, which extends the `IRegion` class and manages a set of `llvm::InstBlock` objects. An `InstBlock` references a block of instructions within a `llvm::BasicBlock` by keeping track of three properties: a pointer to its basic block and a pointer to its first and last instruction within that block respectively. Since `IRegionAggregate` inherits from `IRegion`, we can still use the common IRegion interface for all types of regions.

When detecting regions in the code, it would be a correct solution to create only IRegions where each one only contains a single instruction. However, it is often desirable to group these regions together as much as possible if they are continuous in the CFG, even across basic block boundaries. One advantage of this is that it can increase the performance of VaRA's analysis, since fewer items have to be analyzed. During region detection, VaRA can automatically group continuous IRegions within a function into IRegionAggregates.

In the following, we describe how this process works. It is split into two stages: The first stage finds the regions per BB and the second one groups these regions together across BB boundaries.

The first stage performs the following algorithm per basic block of all functions: It iterates over all instructions of the BB and fetches the region metadata for the current instruction, for example, the set of features this instruction belongs to. We call this set `RRIDs`. In addition, a set of currently "active" instruction blocks (`ActiveInstBlocks`) is maintained for each basic block. It stores the instruction blocks we are currently collecting together with their region ID, for example, the name of a feature together with the block of instructions that belong to that feature. We are iterating over the basic block instructions in sequential order. For every region ID in `RRIDs`, we check if `ActiveInstBlocks` already contains that ID. If it does, we found an additional instruction that belongs to that region, so we can add it to the instruction block. If it does not, we create a new instruction block with the current instruction and store that together with the region ID in `ActiveInstBlocks`. Additionally, we check if `ActiveInstBlocks` contains region IDs that are not contained in `RRIDs` (the set of region IDs of the current instruction). If it does, that means we are finished with that instruction block because the previous instructions belonged to that region but the current instruction does not, so we cannot append it to that block. We are finished with that block because a region is a continuous set of instructions and the current block terminates that. If we iterate over all instructions and all basic blocks of a function in this way, we get a set of all instruction blocks that belong to a region together with their region ID.

In the second step, we group these instruction blocks together across basic block boundaries into `IRegionAggregate` objects. This is done in backwards order on the CFG. We iterate over the basic blocks of a function and get all the instruction blocks we found in the previous step together with their region identifier. If an instruction block starts at the beginning of a BB, we check if the predecessor BBs end with instruction blocks with the same region ID. For all BBs where this is the case, we merge the corresponding instruction blocks to an `IRegionAggregate`. Figure 2.4 on the facing page shows an example of this. It contains three instruction blocks that all have the same region ID (e.g., they belong to the same feature). Since `InstBlock_3` is at the beginning of its BB and `InstBlock_1` and `InstBlock_2` are located at the end of their BBs, we can merge them to a single `IRegionAggregate` [Sat17].

2.4.3. CommitRegions

In 2018, Niederhuber already extended VaRA with the ability to perform control-flow and data-flow analyses between different commits of a software project by adding `CommitRegions` as a new `IRegion` type. In his approach, the

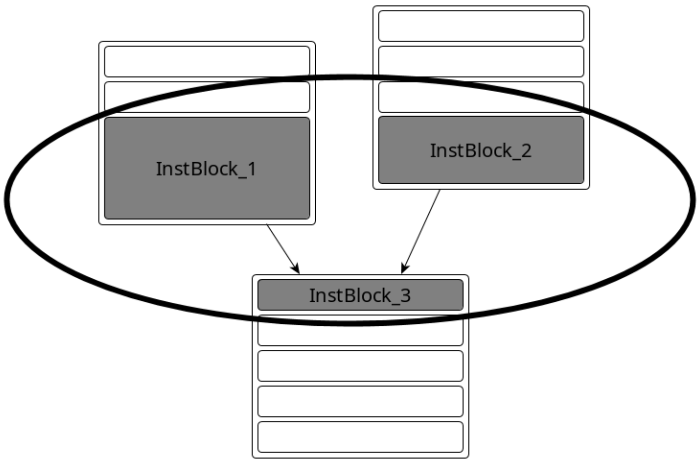


Fig. 2.4. Example of an `IRegionAggregate` consisting of three instruction blocks across three basic blocks

initial region detection is not performed by VaRA. Instead, a preprocessing step is used to annotate the source code with special markers that indicate to which CommitRegion a specific part of the source code belongs. This is done by a special script that uses the Git repository of the analyzed project to get Git's blame output for each source code file to determine for each line of code which commit added/last modified this line. Then, it determines the first and the last occurrence of each commit. It inserts a region start annotation at the first occurrence of a commit and a region end annotation at the last occurrence. An example of this can be seen in Listing 2.5. It shows a small C++ code

```
___REGION_START __RT_Commit "7a32a36e68[...]"
int main() {
  int x;
  int y;
___REGION_START __RT_Commit "5a095078d4[...]"
  x = 30;
___REGION_END __RT_Commit "5a095078d4[...]"
  y = 12;
  return x+y;
}
___REGION_END __RT_Commit "7a32a36e68[...]"
```

Lst. 2.5. Example of marker-based CommitRegion annotation

example with the CommitRegion markers. "`___REGION_START`" indicates the start of a region while "`___REGION_END`" indicates its end. The second string "`__RT_Commit`" encodes the type of the region, which is `CommitRegion` in our case, and the third string of each marker line contains the commit hash of the region. It can be seen that the regions can be nested. For each line, the innermost nested region marker shows which commit last modified that line. In our example, this means that line 6 was created by the commit “`5a095078d4[…]`” while all other lines were created by commit “`7a32a36e68[…]`”.

Niederhuber also extended the clang frontend so that it can detect and process these region markers. For each line it detects the commit hash that belongs to it and it adds this information to the generated IR code as metadata. VaRA’s analysis passes can then use this metadata to create `CommitRegion` objects for every region, which can then be used by the control-flow and data-flow analyses to find interactions between the commits.

However, this approach has some drawbacks. The first drawback is that it requires an additional preprocessing step which makes the analysis process more complex.

The second issue is the nesting of the regions. VaRA’s analysis creates IRegions for all nested annotations in the source code. In our small code example above, this would mean that line 6 is part of both `CommitRegions` “`7a32[…]`” and “`5a09[…]`”, even if it was added by commit “`5a09[…]`” and was never actually part of commit “`7a32[…]`”. It would be more reasonable if every line was only part of a single `IRegion`, namely the region that represents the commit that added that line.

The third drawback is that preprocessor statements cannot be handled correctly during the annotation. If the source code contains, e.g., `#ifdef` directives to conditionally include or remove statements, the region annotations can become inconsistent. Listing 2.6 demonstrates this effect with a small example. If `FOO` is undefined then lines 8 and 9 will be removed by the

```
1   ___REGION_START __RT_Commit "7a32a36e68[...]"
2   int main() {
3     int x;
4     int y;
5   ___REGION_START __RT_Commit "5a095078d4[...]"
6     x = 30;
7   #ifdef FOO
8     x += 1;
9   ___REGION_END __RT_Commit "5a095078d4[...]"
10  #endif
11    y = 12;
12    return x+y;
13  }
14  ___REGION_END __RT_Commit "7a32a36e68[...]"
```

Lst. 2.6. Example of CommitRegion markers with C preprocessor statements

preprocessor. This makes the region annotations inconsistent because the region “`5a09[…]`” is never closed. [Nie18].

2.4.4. *BlameRegions*

BlameRegion is a new type of region that was added to VaRA in order to remove the disadvantages described in the previous section. Contrary to

CommitRegions, BlameRegions do not require a preprocessing step and their annotations can never be nested. They also work as expected in the presence of preprocessor statements. We describe details of their implementation later in Section 3.2 on page 24.

2.4.5. *Analysis Pipeline*

In this section, we explain how the first step of VaRA's analysis pipeline works, namely the annotation of the commit information that can be used by VaRA's interaction analysis in a later step.

vara-clang Blame Annotations

The BlameRegion approach does not require a preprocessing step. Instead, clang is extended with the ability to automatically detect if the compiled source file is stored in a Git repository. If it is, clang loads the repository and gets Git's blame output for the file. This information is then used when clang generates the IR code. Every instruction gets annotated (via IR metadata) with the commit that added the source code line that generated this instruction. This feature is implemented in a fork of the clang project called *vara-clang* and it can be enabled via the "`-fvara-GB`" command-line flag.

Listing 2.7 shows an example of LLVM-IR with added blame metadata. The

```
 1  [...]
 2  define dso_local i32 @readVariable() #0 !Region !0 {
 3  entry:
 4    %res = alloca i32, align 4, !Region !0
 5    [...]
 6  }
 7  [...]
 8  !0 = !{!1}
 9  !1 = !{!"Blame", !2,
     ↪  !"60e791d8d3c792741ea6e9d19546bd269ee75ff5"}
10  !2 = !{!"calculator", !"/home/jb/calculator/.git/"}
11  [...]
```

Lst. 2.7. Example of Blame Metadata LLVM-IR

instruction in line 4 of the example is annotated with a named region metadata that references the metadata node `!0`. This node then references the blame metadata node `!1`, which contains the hash of the commit that last modified the source code line that generated the IR instruction. It additionally contains a reference to another metadata node with the name of the Git repository of that commit as well as a filesystem path. This is necessary because software projects can consist of multiple Git repositories and a commit hash may not be sufficient to uniquely identify a commit across multiple repositories. The indirection of the metadata node `!0` has two reasons. A single IR instruction might be annotated with multiple regions (potentially of different type). In this case, node `!0` would link to multiple region metadata nodes. Additionally, the indirection saves space because only a single metadata node has to be stored for every commit and multiple instructions can reference the same blame metadata node.

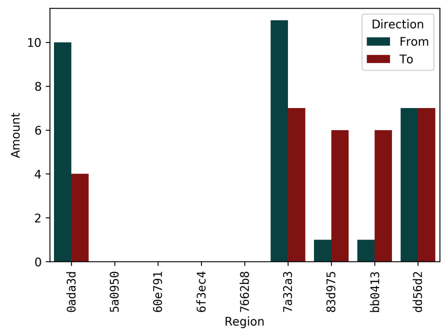
With this annotation approach, dealing with preprocessor statements is not an issue. Previously, the region annotation script added the region marker to the source code before it was processed by clang. When using BlameRegions, however, clang processes the original source code that was not modified by a preprocessing script. Clang first resolves all preprocessor statements and then builds the AST. When the AST is then used to create the IR code, the preprocessor statements have already been processed and we can load the region information from the Git repository. This can easily be done because every AST node stores its `SourceLocation`, which can be used in clang to get the line number in the currently processed file that corresponds to that AST node. With the line number and the path to the source code file (which can also be obtained via the `SourceLocation`), clang retrieves the blame information from the Git repository to determine which commit added the line. This information is then added to the generated IR instruction as metadata.

BlameRegions also do not suffer from the nesting issue because, as we have seen, we only retrieve a single commit hash for every AST node, namely the hash of the commit that last modified the corresponding line in the source code.

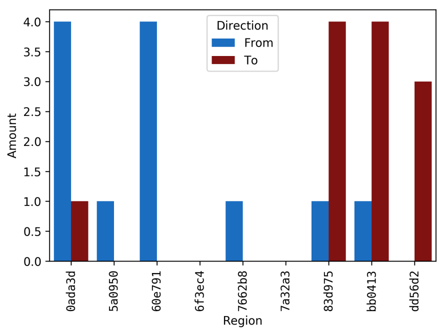
2.5. VaRA-Tool-Suite

The VaRA-Tool-Suite (or VaRA-TS) is an application that aids VaRA's users during multiple tasks. It can automatically download, build and install the latest version of VaRA on the user's system. An existing installation can also be easily updated. The tool suite can perform VaRA's analyses on a range of preconfigured software projects. Result graphs can also be generated to visualize the analyses' outcomes. For example, Figure 2.5 on the next page shows the results of VaRA's commit interaction analysis. For each commit of the analyzed project, it shows the number of ingoing and outgoing commit-flow and data-flow interactions.

The `GitBlameAnnotationReport` experiment is one of the analyses that are supported by the tool suite. It analyzes the control-flow and data-flow interactions between the different commits of a project. For example, VaRA-TS could run the `GitBlameAnnotationReport` experiment on gravity (one of the preconfigured software projects). It automatically downloads the project and compiles it. After that, it starts VaRA's analysis and it collects the result. VaRA-TS can not only analyze the current state of the project, it can also automatically sample an arbitrary number of revisions from the projects history and perform the analysis on each one of them. For example, a user could let VaRA-TS sample 10 revisions from each year of a projects history and perform the `GitBlameAnnotationReport` on each one of them. The result



(a) Visualization of control-flow interactions



(b) Visualization of data-flow interactions

Fig. 2.5. Example of VaRA's visualization of control-flow and data-flow interactions between commits

can be automatically plotted, so we can examine how the control-flow and data-flow interactions develop over time.

The analyzed projects, the used experiment and the sampled project revisions can also be persisted in a `CaseStudy` file. This makes the experiments easily reproducible [VaR19].

2.5.1. *BenchBuild*

BenchBuild is a large-scale empirical-research toolkit that can be used by researchers to repeatedly perform compile-time or run-time experiments on a range of software projects. It minimizes the necessary effort to prepare and execute the experiments by automatically downloading, configuring, and compiling the projects that are analyzed by the chosen experiment. A list of preconfigured and ready to use projects is already supplied by BenchBuild.

However, it is very easy for the user to add own projects. The user only needs to implement a class that inherits BenchBuild's `Project` class and implement the two methods `compile()` (that compiles the project) and `run_tests()` (optionally, for run-time tests). An example can be see in Appendix A on page 53, which shows the definition of the `gravity` project, one of VaRA's current case studies. The project is defined by implementing the `Gravity` class. BenchBuild's `with_git` decorator is added to the class so that BenchBuild automatically clones the project's Git repository before the compilation step is started. It is important to note that the definition of the project is independent from the definition of the experiment. This allows the user to arbitrarily combine the available projects with the configured experiments.

The VaRA-Tool-Suite uses BenchBuild to configure its projects and run its experiments. For example, to run VaRA's `GitBlameAnnotationReport` on the `gravity` project, the user only needs to run the command "`benchbuild run -E GitBlameAnnotationReport gravity`" [Sim+16].

3 Improvements to VaRA's Analyses

In this chapter, we explain how we extended the VaRA analysis framework to combat the disadvantages of the previous CommitRegion implementation. First, we describe the design of VaRA's new Git metadata interface that allows clang and VaRA to extract information from the Git repository of an analyzed software project. Second, we introduce the new BlameRegion type, which we added to VaRA. Third, we explain our BlameRegion filter interface that can be used to filter BlameRegions before and after the analysis process based on different criteria, in order to enhance the analysis.

3.1. VaRA Git Metadata Interface

We already briefly mentioned the Git metadata interface that we implemented in VaRA in previous sections. Its purpose is to allow VaRA to extract metadata from the Git repository of an analyzed software project, for example information about the commits of a project. It is used by clang to annotate the IR code with the blame information and we explain in a later section how we additionally use it to improve VaRA's analysis. To extract the needed metadata from Git repositories, we added the libgit2 library[1] to VaRA.

[1]Available at `https://libgit2.org`

libgit2 is a library implementation of Git. Many features and methods that are offered by the official Git command-line tool are also available via libgit2. In order to programmatically interact with a Git repository, programmers often parse the output of the Git command-line interface, which can be very time-consuming and error-prone. Using libgit2 makes this a lot easier because it offers a solid API to interact with a repository.

However, using libgit2's C API can also be challenging, especially for programmers that are not experienced in programming in C. For example, the programmer often has to manually allocate and free memory for libgit2's data structures.

To remove this burden from the programmer, we implemented a wrapping layer in VaRA that translates libgit2's C API into an object-oriented and easier to use interface. This interface also automatically manages libgit2's memory for the user.

VaRA's Git interface is split into two major parts: the `GitExtraInfo` interface and the `LibGit2Wrapper` interface. The `LibGit2Wrapper` interface is the part that directly interacts with the libgit2 library and handles the memory management. It is not intended to be used by the programmer directly. Instead, the programmer should use the `GitExtraInfo` interface which offers a higher-level abstraction of the `LibGit2Wrapper` interface. Another reason for this separation is that it allows the high-level Git interface to not be fully dependent on libgit2. If a user wants to compile VaRA without the libgit2 library, he or she can still use the `GitExtraInfo` interface, although with less functionality.

The following are the most important classes of the `LibGit2Wrapper` interface:

- `LGGitRepository`
- `LGGitTime`
- `LGGitSignature`
- `LGCommit`
- `LGBlame`

The `LGGitRepository` class opens a libgit2 repository and keeps a pointer to it. `LGGitTime` stores a single time point, e.g., the time of a commit. `LGGitSignature` contains the metadata of either the author or the committer of a commit, e.g., name and email address. `LGCommit` references a specific commit in the repository and `LGBlame` contains the blame information of a commit.

The `GitExtraInfo` interface consists of the following classes:

- `GitExtraInfo`
- `Repository`
- `RepositoryStore`
- `Commit`
- `GitSignature`

The `GitExtraInfo` class allows for the separation between the two interfaces. If the libgit2 library is available, it holds a pointer to a `LGGitRepository` object. A `Repository` object references a single Git repository. It can, for example, be used to get a commit object for a specific commit hash from that repository. Since a single LLVM module can have BlameRegions from multiple repositories, the `RepositoryStore` keeps a list of all of them. The `Commit` class represents a commit from a repository. It can be used to get information about its author or committer. This information is returned as `GitSignature` objects that store a name, an email address, and a time point.

To better understand the relationship between the two interface layers, we look at a small usage example: A user has a pointer to a `Commit` object and wants to determine the name of the commit's author, so he or she calls the commit's `getAuthor()` method, which is implemented as follows: Getting the author name of the commit requires access to the Git repository via libgit2 so we need to use the `LibGit2Wrapper` interface. It is not used directly, however. The `Commit` object calls the `getExtraInfo()` method of its `Repository` which returns an optional of a `GitExtraInfo` object (which is empty if libgit2 support is not available). The `GitExtraInfo` object can be used to access the `LibGit2Wrapper` functionality. It offers a `getAuthor(string CommitHash)` method. The `Commit` object calls this method, which now uses the `LibGit2Wrapper` interface to get the author information from the libgit2 library and returns it as a `GitSignature` object. The `Commit` object then returns the `GitSignature` to the user, allowing convenient access to the requested information.

3.1.1. Extending libgit2

During testing of the new BlameRegions, we discovered an issue with clang's blame metadata annotation. When the Git repository of the analyzed project contained uncommitted source code changes, the LLVM-IR blame annotation of clang was incorrect. The reason for this issue was that libgit2's blame feature does not report blame information for changes that have not yet been committed to the repository. libgit2's blame information is based on the most recent commit of the repository. If, for example, a line was added to a source code file without committing it, then libgit2 would have no information about that line.

This is not an issue when using CommitRegions. The preprocessing script that is used in this approach uses the Git command-line program to obtain the blame information and parses its output. It uses the `git annotate` command, which does include blame information about uncommitted changes. Listing 3.1 shows an example of this. The second line was added but not

```
a5479197371e    (Teresa Johnson     2016-11-11 05:34:58 +0000       10)#include "llvm/Bitcode/BitcodeReader.h"
000000000000    (Not Committed Yet  2019-08-08 19:13:52 +0200       11)// new line
8a4442a342e3    (Mehdi Amini        2016-12-12 19:34:26 +0000       12)#include "MetadataLoader.h"
8a4442a342e3    (Mehdi Amini        2016-12-12 19:34:26 +0000       13)#include "ValueList.h"
```

Lst. 3.1. Example output of the `git annotate` command

committed yet. Git reports it as having the commit hash "00000[...]" and the author "Not Committed Yet".

To solve this issue when using BlameRegions, we created a fork of the libgit2 library and extended is so that its blame feature also includes information about uncommitted code changes.

Git Blame Uncommitted Algorithm

libgit2 stores the blame information of a file in a list of individual hunks. Instead of modifying how libgit2 builds that list, we decided to instead modify the list afterwards and insert the information about the uncommitted code changes into it. The first step of the algorithm is to let libgit2 create the hunk list with the blame information. Second, we use the libgit2 method `git_diff_tree_to_workdir` to get the differences between the repository's latest committed state and the current (uncommitted) state of the working directory. In the third step, we iterate over these differences and include them into the blame hunk list. For each diff entry, we determine whether it removes or adds source lines. If it removes lines, we find the corresponding hunk in the list and remove the lines from it. If the hunk becomes empty in the process, we delete it from the list. The subsequent hunks in the list are shifted by the number of deleted lines. If the diff entry adds lines, we create a new hunk with these changes and insert it into the hunk list. If necessary, we split an existing hunk to be able to insert the new one. Again, all subsequent hunks of the list are shifted by the number of added lines.

3.2. BlameRegion Support in VaRA

In Section 2.4.4 on page 16, we already briefly described BlameRegions as an alternative to CommitRegions. Remember, CommitRegions have the disadvantage that they need a preprocessing script to annotate the region information in the source code. Due to the way the preprocessing script annotates the nested regions, it can falsely mark source code lines as belonging to a commit. An additional drawback is the fact that C preprocessor statements cannot be handled correctly and can lead to inconsistent region annotation. We described these issues in detail in Section 2.4.3 on page 14.

To solve these issues, we implemented `vara::BlameRegion` as a new `IRegion` type. In Section 2.4.2 on page 13, we explained the `IRegion`, `IRegionAggregate`, and the `InstBlock` classes and how they relate to each other. Instead of inheriting from `IRegion` directly, `BlameRegion` interhits from the `IRegionAggregate` class, which aggregates multiple `InstBlock` objects into a single region.

3.2.1. vara-clang Blame Annotations

In order to add the region metadata to the generated IR code, clang retrieves the Git blame output of a project's repository. It is able to get that information from the repository by using the Git metadata interface we added to VaRA. For each line in the compiled source code file, it fetches the blame information from our interface and adds that information to the IR metadata. Listing 2.7 on page 17 shows an example of the blame IR metadata.

3.2.2. VaRA BlameDetection Pass

VaRA's analyses are implemented as LLVM optimization passes. VaRA receives the LLVM-IR code from clang with the embedded BlameRegion metadata. In order to use that information, the first necessary step is to extract the region metadata and use it to create the BlameRegion objects. This is done by the `vara::BlameDetection` pass that we added to VaRA. It can be run by supplying the `-vara-BD` command-line flag to the LLVM optimizer. The `BlameDetection` class inherits `llvm::FunctionPass`, and thus runs on all functions in the current module. The pass's `runOnFunction()` method calls `llvm::IRegionAggregate::detectAggregateRegion(llvm::Function)` with the currently processed function as argument to do the actual region detection. We have already described this method in Section 2.4.2 on page 13. It extracts the IR metadata and aggregates the regions to create `IRegionAggregate` objects, in our case `BlameRegion` objects. After the region detection is performed, the `BlameDetection` can optionally output debugging information about all detected `BlameRegions`. The user can enable this feature by passing the -vara-print-IRegions flag. After all BlameRegions have been created, they are ready to be used by VaRA's analyses.

3.2.3. BlameRegion Implementation

The vara::BlameRegion class extends the general IRegionAggregate type and additionally stores a reference to the region's Commit and Repository object. Listing 3.2 shows the class's public API. The factory method create-

```cpp
class BlameRegion : public llvm::IRegionAggregate {

public:
  static unsigned createID(llvm::StringRef RegionID,
                           int RepoID);

  static bool classof(const llvm::IRegion *IR);

  static BlameRegion *createNewBlameRegion(
          llvm::StringRef RegionID,
          Commit *RegionCommit, int RepoID);

  explicit BlameRegion(llvm::StringRef RegionID,
                       Commit *RegionCommit,
                       int RepoID);
  ~BlameRegion() override = default;

  llvm::StringRef getRegionID() const;
  int getRepoID() const;
  llvm::Optional<Repository *> getRepository() const;
  llvm::Optional<Commit *> getCommit() const;

  std::string getRepresentation() const override;
  std::string getHighlightColor() const override;
};
```

Lst. 3.2. Public API of the BlameRegion class (shortened)

NewBlameRegion() can be used to create a new BlameRegion. It gets passed the RegionID, the RegionCommit, and the RepoID as arguments. The RegionID is simply the region's commit hash as a string, which uniquely identifies the commit within a repository. Every BlameRegion stores a reference to its commit in the form of a vara::Commit object and to the repository of that commit (in the form of a vara::Repository object). This information is passed to the factory method via a Commit pointer and a repository ID. The BlameRegion offers getters for all these properties. The Commit and Repository classes are part of VaRA's Git metadata interface which we explained in Section 3.1. Every region is uniquely identifiable via an ID, which is generated by the createID() method based on the commit hash, and the repository ID. It creates a 45-bit integer in which the first 5 bits are the `RepoID` and the remaining 40 bits are created by the first 10 characters of the commit hash. A commit hash consists of only hex characters, so we can encode a single character in 4 bits. Using both the `RepoID` and the commit hash to create the regions ID ensures that two `BlameRegions` from two different commits will always have different IDs. The `getRepresentation()` method returns the region's commit hash and the `getHighlightColor()` method returns a pseudorandom RGB color value based on the region's ID that can be used for debugging purposes to make different regions easily distinguishable by printing them in different colors.

BlameRegion Aggregate Traits

The general strategy of the IRegion aggregation algorithm described in Section 2.4.2 on page 13 works independently of the concrete IRegion type. However, there are some details of its implementation that depend on the type. This is why the `detectAggregateRegion()` method is implemented as a function template with the region type as template parameter. The parts of the aggregation algorithm that depend on the region type are extracted into three methods in the `IRegionAggregateTrait` trait class. The `IRegion` type has to provide a template specialization of this trait class that implements these three methods:

```
using IRegionTy = vara::BlameRegion;
using RawRegionIDTy = std::pair<llvm::StringRef, vara::Repository *>;

static SetVector<RawRegionIDTy> getRawsRegionIDsFromMD(Instruction &Inst)}
static IRegionAggregate *createNewRegion(RawRegionIDTy MD)}
static bool isSame(IRegionTy *R, RawRegionIDTy MD)}
```

`RawRegionIDTy` and `IRegionTy` are type aliases that also have to specified by the region type. `IRegionTy` is simply an alias to the region type, e.g., `BlameRegion`. `RawRegionIDTy` specifies a type that can uniquely identify a region and is used to create an `IRegion` object. In the case of BlameRegions, it is a pair of the region's commit hash and a pointer to the repository of that commit.

By implementing the `getRawsRegionIDsFromMD` method, the user specifies how the region metadata is extracted from an instruction. The method returns that metadata as a set of RawRegionIDs. The `createNewRegion` method is called by the aggregation algorithm to create a new region of the appropriate type. The region information that was returned by the `getRawsRegionIDsFromMD` method is passed as an argument. `isSame` is used as a comparator between the created region and the RawRegionID. This is used by the aggregation algorithm to determine if an `IRegion` object has already been created for a specific `RawRegionIDTy`. With the fully specified `IRegionAggregateTraits` VaRA's detection algorithm can now create BlameRegions.

Analysis Support for BlameRegions

To allow the analysis to use the new BlameRegions in addition to the old CommitRegions, some modification in VaRA are needed. We implemented a `vara::BlameTaint` class that extends `vara::Taint`. This class represents the taint of a single `BlameRegion` and is used by VaRA's taint analysis.

VaRA's control-flow and data-flow analyses are separated in different LLVM passes. To support BlameRegions the following two passes require modifications:

- CommitTaintFlowAnalysis
- CommitFlowReport

The `CommitTaintFlowAnalysis` pass performs the actual analysis. We extended it so that it creates `BlameTaint` objects for all found BlameRegions. We additionally had to extend the `CommitCFGAnalysisTraits` trait class to also support BlameRegions since it is used by the `CommitTaintFlowAnalysis` pass.

The `CommitFlowReport` is responsible to collect the result of the control-flow and data-flow analysis and write them to a YAML file. We had do modify it to not only fetch analysis results for CommitRegions, but also for BlameRegions.

3.3. Interaction Filters

Depending on the analyzed software project, VaRA's analysis can require a lot of time and the generated analysis results can be very large. This can make it difficult to interpret the results and to gather useful insights into the control-flow and data-flow interactions of a project. To solve these issues and to allow VaRA's user to better control the analysis, we added support for *Interaction Filters* in VaRA.

Interaction filters allow the user to exclude certain BlameRegions from VaRA's analysis based on different filter criteria. Depending on the specific question the user wants to answer with the analysis, there can be many BlameRegions that are of no interest for the result. By using Interaction Filters, VaRA can ignore these regions in the analysis. This can be used to improve the speed of the analysis and to reduce the size of the generated result, which can make it easier to interpret.

3.3.1. Filter Types

An interaction filter specifies which interactions between two commits the analysis should keep and which it should remove because they are of no interest. Every filter offers the following three functions that can be used to evaluate it:

- `filter(SourceRegion, TargetRegion)`
- `filterSource(SourceRegion)`
- `filterTarget(TargetRegion)`

The `filter` method takes two BlameRegions as arguments, the source and the target of an interaction. It then returns either `KEEP` or `REMOVE`, depending on whether the filter criteria match the passed regions or not. `REMOVE` means that the interaction should not be included in the analysis result, while `KEEP` means the opposite. Since the `filter` method requires both the source and target region to make a filter decision, it can only be used after an interaction has been found. Because of this, it can only be used to reduce the size of the analysis result, but it can not improve its performance since all interactions are always computed regardless of whether or not they will be removed afterwards.

The `filterSource` and `filterTarget` methods take only one BlameRegion as argument. They allow the user to filter out regions from the analysis before the interaction—between a source-target pair—is found. If regions can be removed before the analysis, they do not have to be considered by the analysis, which can increase its performance. `filterSource` checks if, according to the filter, the passed region can occur in an interaction as the source region. `filterTarget` checks if the region can occur as a target. If both `filterSource` and `filterTarget` return `REMOVE` for a region, the region can be ignored by the analysis.

Some types of filters can have one or more child filters. This can be used to arrange multiple filters in a tree structure and build more complex filters by combining different filter criteria. Figure 3.1 on the next page shows the class hierarchy of all available filter types. `InteractionFilter` is the common type of all filters. It can be divided into `FilterOperators` and `ConcreteInteractionFilters`. A `ConcreteInteractionFilter` performs the actual filtering while a `FilterOperator` can be used to combine filters or to define the scope of a filter. The `ConcreteInteractionFilters` can be further divided into `UnaryInteractionFilters` and `BinaryInteractionFilters`. They can not have child filters, they can only be used as leaf nodes in the filter tree.

`FilterOperator`

`FilterOperators` can be used to combine filters or to restrict the scope of the evaluation of a filter.

The filters `SourceOperator` and `TargetOperator` can have a single child filter and restrict how that filter is evaluated. For example, if we imagine a `SourceOperator` with a `UnaryInteractionFilter` (e.g., `AuthorFilter`) as its child, then the `SourceOperator's` `filter` method does not return the result of the `filter` method of its child, but instead returns the result of the child's `filterSource` method. Without the `SourceOperator`, the `filter` method of a unary filter evaluates the source region as well as the target region. Adding the `SourceOperator` as parent ensures that only the source region is checked against the filter criteria.

If the user specifies a filter tree with filter types that produce a semantic conflict (e.g., specifying a `SourceOperator` with a `TargetOperator` child) then all three filter methods always return the safe `KEEP` decision that does not

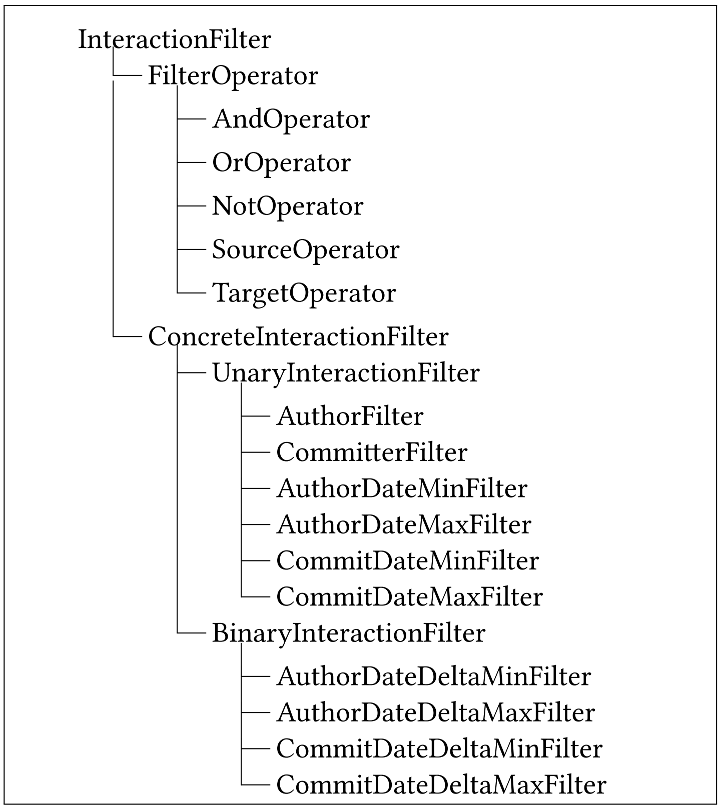


Fig. 3.1. `InteractionFilter` class hierarchy

remove anything. Similarily, calling `filterTarget` on a `SourceOperator` filter also returns `KEEP`.

The remaining filter types `AndOperator`, `OrOperator`, and `NotOperator` represent the basic operators of Boolean algebra and are used to combine different filters. `AndOperators` and `OrOperators` can have two or more children, while `NotOperators` can only have one child. The `AndOperator's` only returns `KEEP` if all of its children return `KEEP`, otherwise it returns `REMOVE`. The `OrOperator` returns `KEEP` if at least one of its children returns `KEEP` and the `NotOperator` returns the opposite of its child.

`BinaryInteractionFilter`

A `BinaryInteractionFilter` can only evaluate a complete interaction. It requires both the source and the target region of an interaction to make a decision. This means that the `filterSource` and `filterTarget` methods of these filters always return `KEEP`, because they do not have the second region to make a decision whether to remove it or not.

The `AuthorDateDeltaMinFilter` and `AuthorDateDeltaMaxFilter` can be configured with a time span and only keep an interaction if the time difference between the author dates of the source region's commit and the target region's commit is bigger (min filter) or smaller (max filter) than the configured time span. For example, imagine that VaRA's analysis found an interaction between two BlameRegions `BR1` and `BR2`. `BR1's` commit has an author date of January 1, 2017, while `BR2's` commit has an author date of January 3, 2017. A `AuthorDateDeltaMinFilter` with a configured time span of one day keeps this interaction, while a configured time span of five days removes it from the result. The filters `CommitDateDeltaMinFilter` and `CommitDateDeltaMaxFilter` are very similar, but they compare the commit time instead of the author time.

`UnaryInteractionFilter`

A `UnaryInteractionFilter` can make a filter decision based on only one region (source or target). When using the `filterSource` or `filterTarget` methods, the filter checks if its filter criteria matches the passed region. If it does, it returns `KEEP`. It is, however, also possible to use the `filter` method with both regions of an interaction (source and target). In this case, the unary filter only returns `KEEP` if both regions match the filter criteria.

The `AuthorFilter` bases its filter decision on the author of the commit of the passed region. If the configured author name and email address match, it returns `KEEP`, otherwise `REMOVE`. The `AuthorDateMinFilter` returns `KEEP` if the author time of the commit is at least the filter's configured date and the `AuthorDateMaxFilter` returns `KEEP` if the author time is at most the configured date. The commit filters work similarly, but they evaluate the committer instead of the author of the commit.

During the evaluation of this thesis, we observed an unexpected behavior when using unary filters. When using, for example, an `AuthorFilter`, to prevent the taint creation for certain BlameRegions, we expected the analysis results to not contain any interactions with these BlameRegions (as either source or target). We observed, however, that while the result did not contain any interactions with the filtered BlameRegions as the source of the interaction, it still contained interactions with these regions as targets. The reason for this behavior is the following: The interaction filter successfully prevents the taint creation before the analysis, so the result cannot contain interactions with these regions as source. It is, however, possible that taints from other BlameRegions (that have not been removed by the filter) have flows to one of the filtered regions. After the analysis, all detected interactions are filtered again, but the unary filter only removes interactions where both regions (source and target) fulfil the filter criterion. In this case, though, only the target region matches, but the source region does not. Because of this, these interactions are not removed by the filter. Fortunately, it is simple to also remove these interactions from the result by extending the used interaction filter. This can be seen in Figure 3.2 on the facing page. The first subtree of the `AndOperator` is the initially used filter. It prevents the taint creation of the filtered regions before the analysis. The second subtree is added to remove interactions after the analysis where the target region matches the filter.

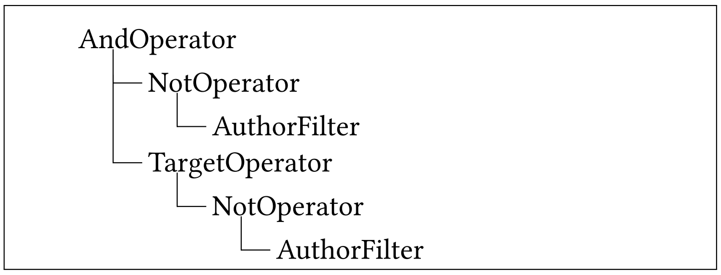


Fig. 3.2. InteractionFilter with workaround to entirely remove certain regions from the analysis result (as source and target)

3.3.2. Filter Editor in VaRA-Tool-Suite

To allow the user to easily configure an interaction filter, we implemented a graphical editor in the VaRA-Tool-Suite. Figure 3.3 on the next page shows a screenshot of the editor with an example filter. It uses the two filters `AuthorFilter` and `CommitDateDeltaMinFilter`. Since they are combined by an `AndOperator` the analysis result contains only interactions that fulfil both of these filters. In this case, the source and the target commit of an interaction must both be authored by "Jane Doe" and the commit times of the two commits must be more than one year apart.

The usage of the editor is very simple. In the top half, the user can select a parent filter and add a child filter to it by using the "+" button, which opens a menu with the different filter types to choose from. Filters can also be deleted and moved up or down. After selecting a filter it can be configured in the lower half of the window. An arbitrary comment string can also be added to every filter.

After the user finished configuring the filter, he or she can export it to a YAML file. This file can also be loaded again to modify it and save it again.

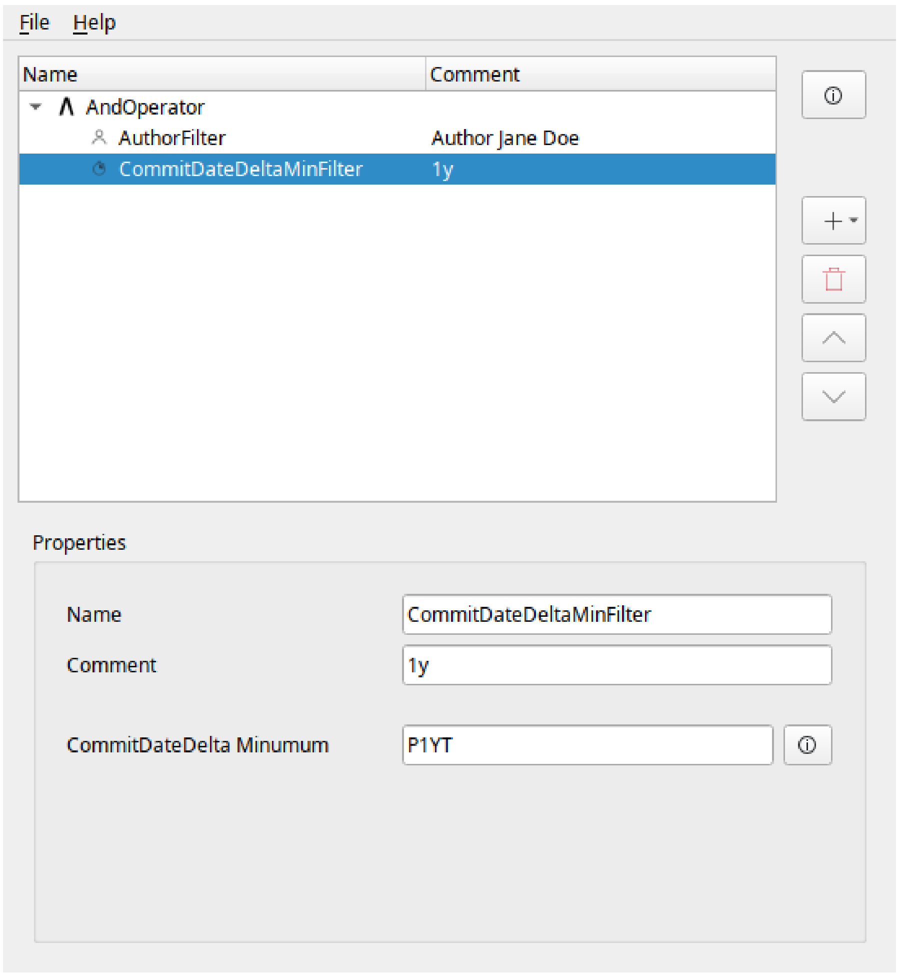
3.3.3. Filter Interface in VaRA

After the user has configured a filter and saved it to a file, it can be used to configure VaRA's analysis. We added the command-line parameter `-vara-cf-interaction-filter` to the LLVM optimizer which can be used to load a filter file. By using LLVM's "YAML I/O" library[2], we parse the file, construct the used filter objects and store them in a tree data structure. The filter objects offer the `filter`, `filterSource`, and `filterTarget` methods that we already described in Section 3.3.1 on page 27. The analysis pass can then use the filter tree to remove unwanted interactions or ignore certain BlameRegions during the analysis.

[2]Documentation available at `https://llvm.org/docs/YamlIO.html`

3.4. Integration of Git Metadata Filters in VaRA's Analyses

The last remaining step is to actually use the interaction filters in the analysis. We want to achieve two different goals by doing this. The first goal is to remove unwanted or uninteresting interactions from the result and to reduce



Name	Comment
AndOperator	
AuthorFilter	Author Jane Doe
CommitDateDeltaMinFilter	1y

Properties

Name	CommitDateDeltaMinFilter
Comment	1y
CommitDateDelta Minumum	P1YT

Fig. 3.3. Filter Editor of VaRA-Tool-Suite

its size. Since a filter's decision on whether to remove an interaction or not might depend on both the source and target region of the interaction, we can only make that decision after that information is known, that is after the interaction analysis has finished. The `CommitFlowReport` pass (see Section 3.2.3 on page 27) runs after the analysis and exports the found interactions into the result file. During that pass, we iterate over all found data-flow and control-flow interactions and use the `filter` method to determine whether to keep an interaction or whether to remove it from the result.

This, however, does not achieve the second goal, which is to reduce the execution time of the analysis. In order to achieve this we need to remove BlameRegions or, more specifically, their taints before running the interaction analysis. In the beginning of the `CommitTaintFlowAnalysis` pass, we iterate over all detected BlameRegions to create taints for the analysis. During this step, we call the filter methods `filterSource` and `filterTarget` for every BlameRegion. If both these methods return `REMOVE` for a region, we know that the analysis result should not contain an interaction with this region as either source or target. In this case, we can completely ignore the region in the analysis (and thus increase the performance) by simply not generating a taint for it.

3.5. Complete Analysis Pipeline

After individually explaining the different parts of VaRA's analyses in the previous sections, we now explain how the parts work together by going through the entire analysis pipeline from start to end. In this example, we analyze a fictitious software project. To simplify the example, we assume that the project is implemented in a single C++ source file called `my-project.cpp`.

Interaction Filter

The first step is to define an interaction filter. In our example, we imagine that we are only interested in control-flow and data-flow interactions between commits whose commit dates are at least 6 months apart but not more than 12 months. Using the graphical editor of the VaRA-Tool-Suite, we configure the following filter tree and save it to the YAML file `interaction_filter.yaml`:

```
AndOperator
 ├── CommitDateDeltaMinFilter (“P6MT”)
 └── CommitDateDeltaMaxFilter (“P12MT”)
```

Clang Frontend

The next step is to use the clang frontend to convert the project's source code to LLVM-IR. During this phase, clang also uses VaRA's Git metadata interface to get the blame information for every source code line. For example, assume clang is currently converting line 42 of `my-project.cpp` with the statement "`int i;`" into LLVM-IR, which was added to the file by the commit `36cf1a6`. Clang converts the declaration of `i` into the IR instruction `%i = alloca i32, align 4` and uses the Git metadata interface to get the blame information for line 42. That information is then added to the generated IR instruction as metadata. After converting the entire source file to LLVM-IR, clang saves the generated code to the file `my-project.ll`:

```
[...]
%i = alloca i32, align 4, !Region !0
[...]
!0 = !{!1}
!1 = !{!"Blame", !2, !"36cf1a6d373332571189a66cfe3d19a[...]"}
!2 = !{!"my-project", !"/path/to/my-project/.git/"}
[...]
```

Interaction Analysis

We can now use the annotated IR file to perform VaRA's commit interaction analysis. Since VaRA's analyzes are implemented as LLVM optimization passes, we need to execute the LLVM optimizer by using the following command:

```
opt -vara-BD -vara-CFR -vara-init-commits
 ↪  -vara-cf-interaction-filter=interaction_filter.yaml
 ↪  -yaml-out-file=my-project_analysis-results.yaml
 ↪  my-project.ll
```

We pass `my-project.ll` as input file to the optimizer as well as the interaction filter file via the `-vara-cf-interaction-filter` parameter. The `-vara-BD` flag instructs VaRA to load the blame metadata from the input file and create the BlameRegion objects so that they can be used by the analysis. The `-vara-CFR` flag enables VaRA's commit interaction analysis. Since we use an interaction filter, we also need to pass the `-vara-init-commits` flag so that VaRA loads the project's Git repository. This is necessary for the interaction filter to work because it needs additional information about the BlameRegions (in our case the time of the BlameRegion's commit) to make its filter decision, which it can get via the Git metadata interface. The `-yaml-out-file` parameter specifies the output file in which VaRA stores the analysis result.

In the first step of VaRA's analysis, it loads the LLVM-IR file and parses the blame metadata for every IR instruction. It then runs VaRA's region aggregation algorithm (see Sections 2.4.2 on page 13 and 3.2.3 on page 26) to group instructions with the same commit hash into regions and creates the `BlameRegion` objects (see Section 3.2.3 on page 25).

The second step is the actual interaction analysis. At first, VaRA loads the interaction filter file that we provided via the command-line. VaRA then iterates over all BlameRegions that were created in the previous step and passes them to the interaction filter to determine, which regions should be ignored by the analysis and which ones should not. If a region should be ignored by the analysis according to the filter, then VaRA does not generate a taint for that region, otherwise, it does. In our example, VaRA cannot ignore any BlameRegions in this step because we use only `BinaryInteractionFilters`. The filter can only decide to remove an interaction when both the source and the target are known because it filters based on the difference between the commit dates of the two commits. When all taints have been created, VaRA runs the control-flow and taint (data-flow) analysis to find the interactions between the BlameRegions. After the analysis, VaRA iterates over all found interactions and uses the interaction filter to remove all unwanted interactions from the result. Finally, all remaining interactions are stored in the result file that was specified via the `-yaml-out-file` parameter.

4 Evaluation

In the previous chapter, we described how we added analysis support for the new BlameRegion type and how we extended the analysis with the ability to filter BlameRegions based on Git metadata. In this chapter, we evaluate whether these approaches are able to improve VaRA's commit interaction analysis by performing the analysis on a set of case studies. Hence, we evaluate the following research questions in this chapter.

Research Questions

- **RQ1:** Does the use of BlameRegions lead to more accurate analysis results compared to CommitRegions?
- **RQ2.1:** Can interaction filters be used to remove unwanted interactions from the analysis result and reduce its size?
- **RQ2.2:** Can interaction filters increase the speed of VaRA's analyses?

4.1. Cluster System

Since part of our evaluation involves time measurements, we run all of these experiments on a dedicated test system so that we are able to produce reliable and comparable results. We use 16 nodes of a cluster system where each node is equipped with an Intel Xeon E-5 2690v2 CPU with 10 cores and 64GB RAM. The used operating system is Debian 10. We exclusively reserve the nodes during our evaluation and run only a single experiment on each node at a time. Furthermore, we try to reduce noise by disabling turbo boost and setting the kernel scaling governor to performance, to always scale CPU cores to their maximum frequency.

4.2. Case Studies

In this section, we present the different case studies that we use to evaluate our research questions. The first case study is a small, synthetic example that implements a simple calculator. The second and third case studies are the real world software projects gravity and gzip.

4.2.1. *Niederhuber's Calculator*

Niederhuber developed a very small calculator program to evaluate his CommitRegion-based commit interaction analysis [Nie18]. It is written in C and consists of only 47 source code lines that were created by 9 commits in total. The small size of the example has the advantage that the analysis result is small enough so that we are able to compare the results by hand. Appendix B on page 55 shows the calculator's source code and a description of each of its 9 commits.

4.2.2. *Gravity*

Gravity[1] is a lightweight programming language written in C. At the time of this writing, the project consists of 46 source files and 17 818 lines of code. Its Git repository contained 575 commits. From the project's history from 2017 to 2019, we sampled 76 revisions from a uniform distribution for our evaluation.

[1] `https://github.com/marcobambini/gravity`

4.2.3. *GNU Gzip*

GNU Gzip[2] is a popular data compression tool for UNIX or UNIX-like operating systems. At the time of our evaluation it consisted of 22 source files with a total of 6 059 lines of code. The project's Git repository contained 580 commits. Gzip uses the gnulib library, which is distributed in a separate Git repository with 20 124 commits in total. On our evaluation system, we were not able to compile gzip revisions older than July 2018 because earlier revisions can not be built with recent versions of the glibc. Therefore, we took all revisions since that date for our evaluation which are 23 revisions in total.

[2] `https://www.gnu.org/software/gzip`

4.3. Tools

To run our case studies and evaluate the results, we need to make additions to the VaRA-Tool-Suite, which we describe in this section.

4.3.1. *VaRA-Tool-Suite Experiment*

We already described the tool suite's `GitBlameAnnotationReport` experiment. It compiles the case study project and runs VaRA's commit interaction analysis on it. For our evaluation, we implemented a new experiment, called `GitBlameFilteredAnnotationReport`, which performs a very similar task. The only difference is that it can take a YAML file that configures an interaction filter and pass that to VaRA's analysis. This way, we can run both experiments on our case studies and compare their results.

In addition, we extended both experiments with the ability to measure how long each analysis runs. We use the GNU *time* command to measure the wall-clock time of the analysis.

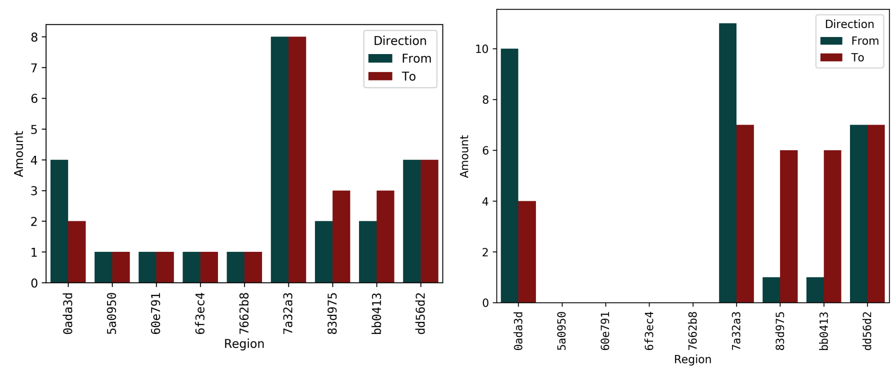
4.3.2. VaRA-Tool-Suite Visualization

To evaluate the results of our case studies we also implemented four new plot types in the tool suite. They compare the results of a case study when it is analyzed without interaction filters to the results when it is analyzed with them. The four different plots compare the four metrics result size, analysis time, number of control-flow interactions, and number of data-flow interactions.

4.4. Experiment 1 — Calculator

We use the calulator case study to answer RQ1. We run VaRA's commit interaction analysis on the calculator example by using the CommitRegion approach and the new BlameRegion approach. When comparing the two results, we immediately spot a difference. The result file of the CommitRegion approach is 12 KiB large while the BlameRegion result is only 9 KiB large.

Figures 4.1a and 4.1b show the found control-flow interactions of both approaches. For each commit, they show the number of interactions from that commit to other commits and vice versa. The first difference we observe is



(a) Control-flow interactions when using CommitRegions

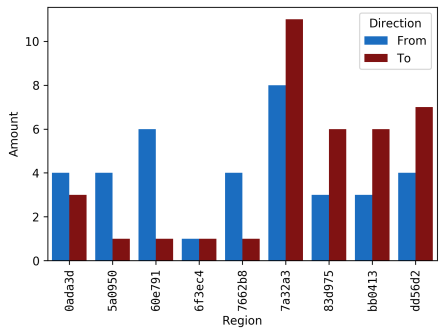
(b) Control-flow interactions when using BlameRegions

Fig. 4.1. Detected control-flow interactions in calculator case-study when using CommitRegions (4.1a) or BlameRegions (4.1b)

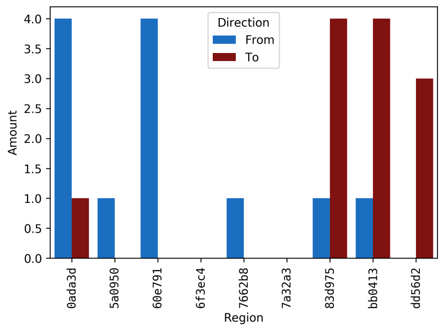
that the CommitRegion approach detected interactions with the four commits `5a09`, `60e7`, `6f3e`, and `7662`, while the BlameRegion approach did not. These four commits added the four methods `add`, `readVariable`, `listOperations`, and `sub` to the program. The CommitRegion-based analysis detected control-flow interactions between these commits and the commit `7a32`, the very first commit of the calculator. When inspecting the source code, we observe that these interactions are not correct. The program does not contain any control-flow interactions between these four methods and any line that was added by the `7a32` commit. `7a32` adds the first line of the file (the `#include` statement) and the last line (the closing brace). The CommitRegion approach adds the region information by adding start and end markers to the source code that indicate the start and the end of a region. The start marker is placed at the first occurrence of a commit and the end marker at the last occurrence of a commit. In the calculator example, it places the start marker for the `7a32` commit before the first line and the end marker after the last line of the file. Because of this, VaRA detects every source code line between these to markers as belonging to the region `7a32` even if it was added later by a different commit. This is the reason why the CommitRegion-based analysis wrongly finds many interactions with this commit. The BlameRegion approach does not detect these interactions because it annotates the region information differently. A source code line is only annotated with a single BlameRegion, namely with the commit that last modified the line. In the calculator example, this means that only the lines that were actually added by commit `7a32` are annotated with that region.

The other difference we find is that in some cases the BlameRegion approach seems to detect more interactions. Inspecting the result file reveals that these are duplicates and the analysis did not actually find more interactions. These duplicates can be created because multiple BlameRegions might belong to a single commit, for example, when a commit changes code in two different places. It can also happen if a commit contains only a single continuous change, because the BlameRegion annotation happens on IR-level and clang might rearrange the code when converting it to LLVM-IR. The result file of the analysis contains only commit hashes, so interactions between different BlameRegions with the same commits lead to duplicate entries in that file.

Figures 4.2a and 4.2b show the detected data-flow interactions. In total,



(a) Data-flow interactions when using CommitRegions



(b) Data-flow interactions when using BlameRegions

Fig. 4.2. Detected data-flow interactions in calculator case-study when using CommitRegions (4.2a) or BlameRegions (4.2b)

the CommitRegion approach detected 37 data-flow interactions, while the BlameRegion approach detected only 12. Of the 25 interactions that were only detected by the CommitRegion-based analysis 23 are incorrect. Again, the reason for this is the marker-based annotation of the CommitRegion approach. The BlameRegion approach did not detect these 23 interactions. The two remaining interactions were correctly detected by the CommitRegion approach but not by the BlameRegion approach. These two interactions are implicit data flows. They were not detected because VaRA's taint analysis does currently not propagate indirect taints to detect such flows. The CommitRegion-based approach only detected these interactions by chance because the target commit inserted code above the end marker of the source region so the code of the target region was annotated with both the source and target commit. In general, the CommitRegion approach is also not able to detect implicit flows.

4.4.1. Addressing RQ1

To answer **RQ1** we look at how many correct and incorrect control-flow and data-flow interactions were detected by CommitRegion-based and the BlameRegion-based analysis. We observed that both approaches detected the same number of correct interactions. We do not count the two indirect data flows since this is a limitation of VaRA's taint analysis that is independent of the used region type. We determined that the CommitRegion approach detected several incorrect interactions. The BlameRegion approach detected none of these. Overall, we found that both approaches found the same number of correct interactions. The CommitRegion approach detected several incorrect interactions while the BlameRegion approach did not. This leads us to accept **RQ1**.

4.5. Experiment 2 — Gravity

We use the gravity case study to answer **RQ2.1** and **RQ2.2**. Both of these research questions cover the use of the interaction filters. We run VaRA's commit interaction analysis on gravity with and without interaction filters and compare the results.

4.5.1. Used Interaction Filters

In two different experiment runs, we use the following interaction filters:

- **IF2.1:** In the first scenario, we are only interested in interactions where Marco Bambini, the maintainer of the gravity project, is not involved. More precisely, we want the analysis to ignore all BlameRegions whose commits were authored by Bambini. For this, we use an `AndOperator` with two children. The first child is a `NotOperator` with an `AuthorFilter`. It will prevent the creation of taints for commits authored by Bambini. The second subtree is a `TargetOperator` with a `NotOperator` as child, which in turn has an `AndOperator` as child. This will remove all interactions from the result where the target commit was created by Bambini.
- **IF2.2:** The second interaction filter removes the interactions based on the time between their source and target commits. Hence, the result contains only interactions where the time between the two commits

of the interaction lies into a configured time span, in our case if it is larger than 12 months. For this we use a `CommitDateDeltaMinFilter`.

Since all our interaction filters either filter based on time or on people, we chose the two evaluation scenarios to cover these types. The first scenario filters based on author and the second one on time. Another reason is that we expect a lower analysis time with the first filter but not with the second one. The time filter (time delta between source and target region) can only be evaluated after the interaction has been found. This means that the analysis still has to process all BlameRegions. With the author filter, however, VaRA's analysis can ignore all regions that do not match the filter, so we expect an increased analysis speed in that case.

In addition to the two described filters, we use a third interaction filter to create a baseline for our comparison. We configure an interaction filter to remove all BlameRegion taints before the analysis, which results in the fastest possible analysis with the smallest possible result. It indicates how much we can theoretically reduce the analysis time and the result size by using an interaction filter. We create this baseline by using an `AuthorFilter` with a random string that does not match any author in the project's Git history.

4.5.2. *Results*

For each of the two interaction filters, we perform VaRA's commit interaction analysis with and without using the filter. Our baseline experiment is also run. We compare result size, analysis time, number of data-flow interactions and number of control-flow interactions for each case.

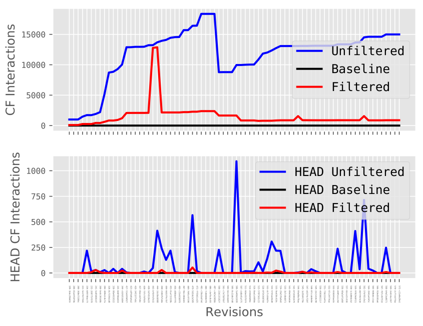
IF2.1

Figure 4.3 on the facing page and Table 4.1 show the results of the gravity analysis when using IF2.1. The used interaction filter ensures that the result contains only interactions where neither the source nor the target commit was created by Marco Bambini, the maintainer of gravity. The control-flow

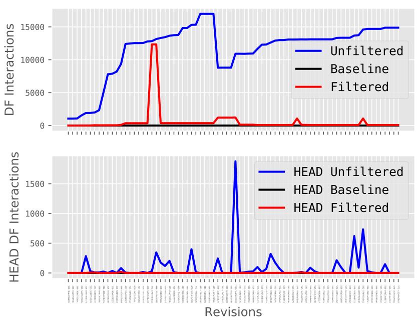
Tab. 4.1. Gravity IF2.1 Results (mean values across all tested revisions)

	Unfiltered	Baseline	Gravity IF2.1
Control-Flow Interactions	11 800	0 (−100.0%)	1 511 (−87.1%)
Data-Flow Interactions	11 622	0 (−100.0%)	569 (−95.3%)
Analysis Time (s)	58.0	49.6 (−12.9%)	49.5 (−13.1%)
Result Size (MiB)	3.59	0.73 (−76.5%)	0.98 (−69.7%)

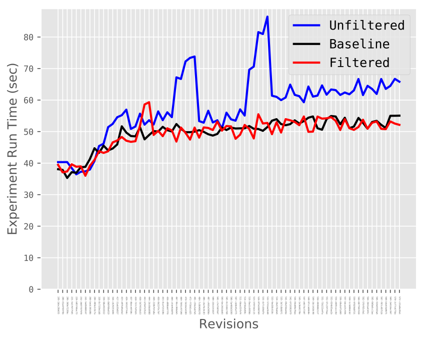
and data-flow interaction plots show the total number of detected interactions on the top. The bottom plot shows the number of interactions where the source or the target commit is the newest commit (HEAD) of the project. As expected, the baseline experiment detected no interactions, since the



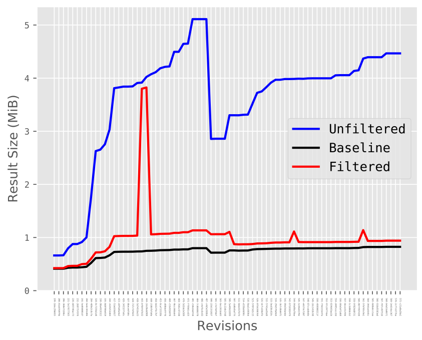
(a) Control-Flow Interactions



(b) Data-Flow Interactions

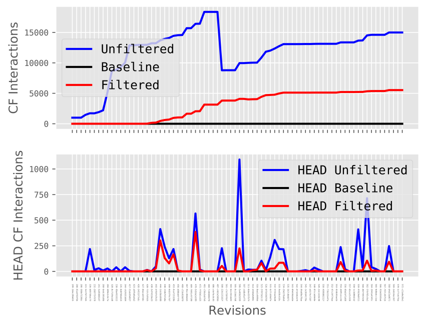


(c) Analysis Time

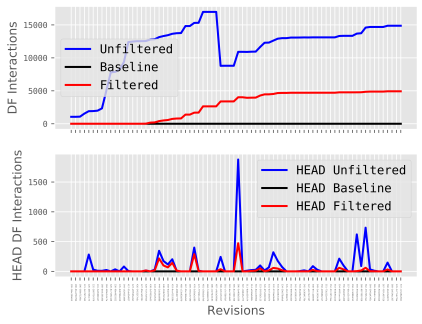


(d) Result Size

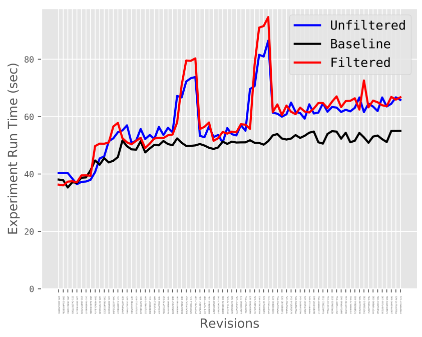
Fig. 4.3. Gravity IF2.1 Results



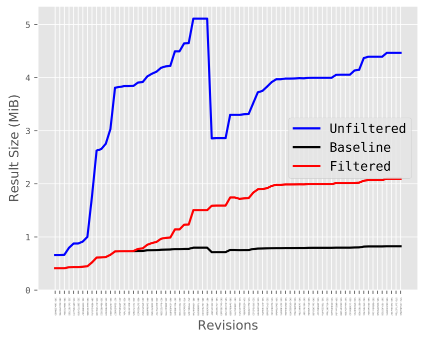
(a) Control-Flow Interactions



(b) Data-Flow Interactions



(c) Analysis Time



(d) Result Size

Fig. 4.4. Gravity IF2.2 Results

used interaction filter removed all BlameRegions before the analysis. When comparing the unfiltered analysis with the filtered one, we see a substantial decrease in detected interaction. The filter decreased the detected control-flow interactions by 87.1% on average (median: 90.1%). The average reduction of data-flow interaction is 95.3% (median: 99.3%).

The interaction filter reduced the analysis time by 13.1% (mean) / 13.0% (median). The baseline filter reduced the analysis time by 12.9% on average (median: 12.8%). The interaction filter of this scenario reduced the analysis time slightly more than the baseline experiment (by 0.2% on average). Since this corresponds to only a difference of 0.16 seconds in the total experiment run-time, we assume this is due to uncertainties in our measurement setup.

The size of the result file was reduced by 69.7% on average (median: 75.5%) with a baseline reduction of 76.5% (median: 80.1%).

IF2.2

Figure 4.4 and Table 4.2 on the next page show the results of the gravity analysis when using the IF2.2 filter that removes all interactions where the time between the source and target commit is less than 12 months. The interaction filter reduced the detected control-flow interactions by 76.4% on

Tab. 4.2. Gravity IF2.2 Results (mean values across all tested revisions)

	Unfiltered	Baseline	Gravity IF2.2
Control-Flow Interactions	11 800	0 (−100.0%)	3 036 (−76.4%)
Data-Flow Interactions	11 622	0 (−100.0%)	2 768 (−78.6%)
Analysis Time (s)	58.0	49.6 (−12.9%)	59.1 (+1.6%)
Result Size (MiB)	3.59	0.73 (−76.5%)	1.44 (−58.5%)

average (median: 63.2%). The data-flow interactions were reduced by 78.5% on average (median: 66.8%).

The analysis time with the interaction filter is slightly higher than without it (1.6% slower on average; median: 2.0%). This is because the used interaction filter does not remove any BlameRegions before the analysis. It can only remove interactions afterwards, since both the source and the target of an interaction are needed to make a filter decision. The total run-time is made up of the complete analysis process and, in addition, the filtering step after the analysis. This leads to a slightly larger run-time when using this kind of filter. The baseline experiment reduced the run-time by 12.9% on average (median: 12.8%).

Similar to the number of interactions, the result size is also lower. On average, it is 58.5% smaller (median: 51.2%). The baseline size reduction is 76.5% (median: 80.1%).

4.6. Experiment 3 — GNU Gzip

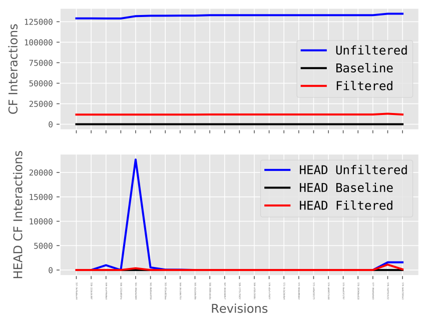
We also use the gzip case study to answer **RQ2.1** and **RQ2.2**. The experiment is very similar to the gravity case study. We run VaRA's commit interaction analysis on gzip with and without interaction filters and compare the results.

4.6.1. Used Interaction Filters

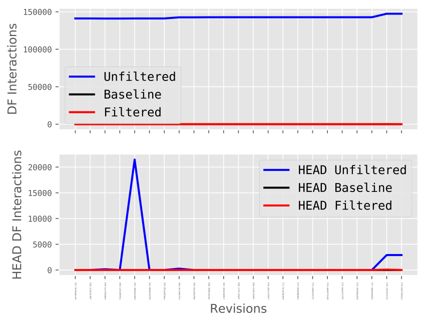
In two different experiment runs, we use the following interaction filters:

- **IF3.1:** The first interaction filter configures the analysis to ignore all BlameRegions where the commit author is not one of the four main authors of gzip and gnulib (Paul Eggert, Jim Meyering, Bruno Haible, and Jean-loup Gailly). This is achieved by a `NotOperator` filter with an `OrOperator` as child, which in turn has four `AuthorFilters` as children. Similarly to IF2.1, we add a second subtree to the filter with a `TargetOperator` so that the result contains no interactions where the four developers are involved.

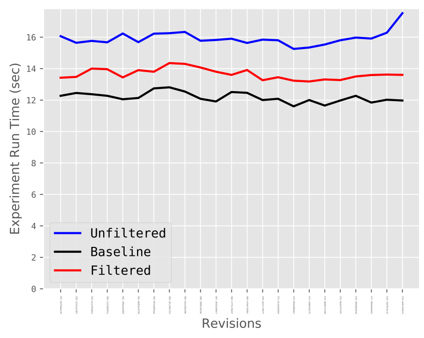
- **IF3.2:** The second interaction filter removes BlameRegions taints based on their commit time. With this filter, the analysis only considers BlameRegions whose commits were committed in the years 2015 to 2017, so the result contains only interactions were the source and target



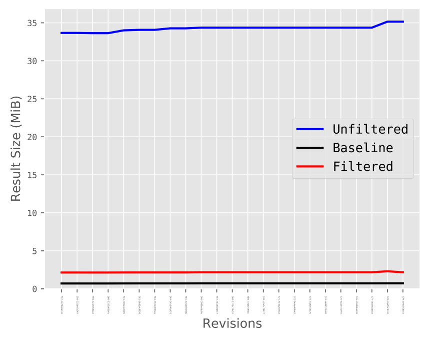
(a) Control-Flow Interactions



(b) Data-Flow Interactions



(c) Analysis Time



(d) Result Size

Fig. 4.5. Gzip IF3.1 Results

commits are from this time range. For this we combine a `CommitDate-MinFilter` with a `CommitDateMaxFilter` by using an `AndOperator`. To also remove interactions where the target commit is outside of the time range, we also add a second subtree with a `TargetOperator`.

Similar to the gravity experiment, we again use one interaction filter that filters based on the author and one that filters based on time.

4.6.2. Results

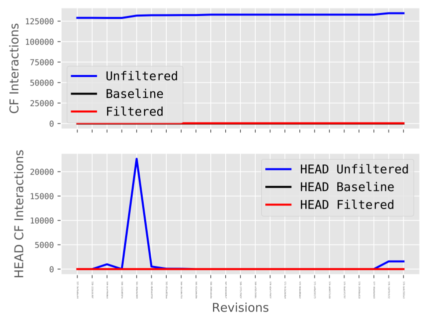
For each of the two interaction filters, we perform VaRA's commit interaction analysis with and without using the filter. Like with the gzip experiment, a baseline experiment is also run that removes all BlameRegions before the analysis. We compare result size, analysis time, number of data-flow interactions and number of control-flow interactions for each case.

IF3.1

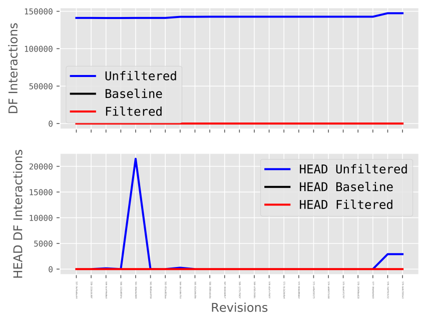
Figure 4.5 and Table 4.3 on the next page show the results of the gzip analysis when using the AuthorFilters IF3.1 filter that removes all BlameRegions whose commits were not created by the four main project authors. By using the AuthorFilters, we reduced the number of control-flow interactions by 91.0%

Tab. 4.3. Gzip IF3.1 Results (mean values across all tested revisions)

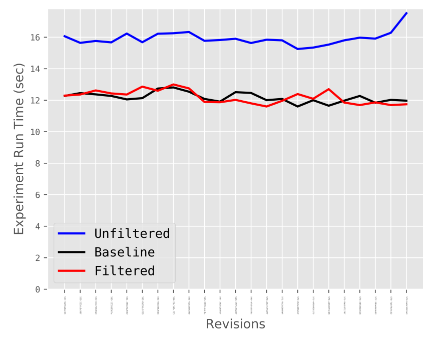
	Unfiltered	Baseline	Gzip IF3.1
Control-Flow Interactions	132 233	0 (−100.0%)	11 838 (−91.0%)
Data-Flow Interactions	142 590	0 (−100.0%)	92 (−99.9%)
Analysis Time (s)	15.9	12.2 (−23.5%)	13.7 (−14.2%)
Result Size (MiB)	34.3	0.71 (−97.9%)	2.17 (−93.7%)



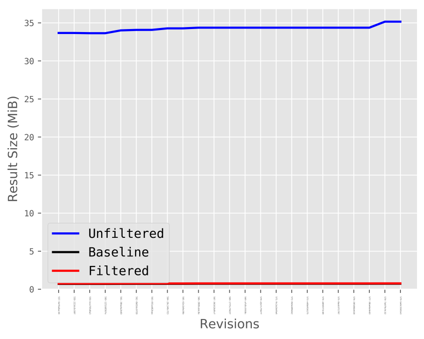
(a) Control-Flow Interactions



(b) Data-Flow Interactions



(c) Analysis Time



(d) Result Size

Fig. 4.6. Gzip IF3.2 Results

on average (median: 91.1%). The number of detected data-flow interactions is reduced by 99.9% on average (median: 99.9%).

The interaction filter lowered the required analysis time by 14.2% on average (median: 14.3%). The maximum time reduction any filter could achieve (baseline) is 23.5% (median: 23.4%).

The result size was reduced by 93.7% on average (median: 93.7%). The baseline experiment reduced the size by 97.9% on average (median: 97.9%).

IF3.2

Figure 4.6 and 4.4 on the next page show the gzip results for the IF3.2 filter that restricts the analysis to commits from the years 2015 to 2017. The unfiltered result contains on average 132 233 control-flow interactions and

Tab. 4.4. Gzip IF3.2 Results (mean values across all tested revisions)

	Unfiltered	Baseline	Gzip IF3.2
Control-Flow Interactions	132 233	0 (−100.0%)	339 (−99.7%)
Data-Flow Interactions	142 590	0 (−100.0%)	41 (−99.9%)
Analysis Time (s)	15.9	12.2 (−23.5%)	12.2 (−23.4%)
Result Size (MiB)	34.3	0.71 (−97.9%)	0.76 (−97.8%)

142 590 data-flow interactions. The filtered result contains only an average number of 339 control-flow interactions and 41 data-flow interactions. This corresponds to an average reduction of 99.7% for control-flow interactions (median: 99.7%) and 99.9% for data-flow interactions (median: 99.9%).

The time reduction of the baseline experiment is 23.5% on average (median: 23.4). The used interaction filter results in a 23.4% lower analysis time on average (median: 23.8%). Similar to the IF2.1 results, the median time reduction by the interaction filter is slightly larger (0.4%) than the baseline experiment. However, since the total run-time of the baseline experiment is only 12.2 seconds on average, we think this difference can also be explained by uncertainties in our measurement setup.

The result size is also almost as small as the baseline. The interaction filter reduced the result size by 97.8% on average (median: 97.8%). The baseline size reduction is 97.9% on average (median: 97.9%).

4.7. Addressing RQ2.1

In order to answer RQ2.1, we look at the gravity and gzip experiments and how the used interaction filters reduced the number of detected control-flow and data-flow interactions and the size of the result file.

For the gravity experiment, the two used interaction filters reduced the number of data-flow interactions in the result by 95.3% and 78.6% on average. The number of control-flow interactions was reduced by 87.1% and 76.4% on average, and the size of the result file was 69.7% and 58.5% lower.

The achieved reduction is even higher for the gzip experiment. In the first scenario (IF3.1), the interaction filter reduced the detected data-flow interactions by 99.9% and the control-flow interactions by 91.0% on average. The result size is 93.7% lower. In the second scenario (IF3.2), the filter reduced the data-flow interactions by 99.9%, the control-flow interactions by 99.7% and the result size by 97.8% on average.

In summary, we find that in all four scenarios, the interaction filters lead to a considerable reduction in the number of detected interactions. This leads us to accept RQ2.1.

4.8. Addressing RQ2.2

To answer RQ2.2, look at the four scenarios of the gzip and gravity experiments and by how much the interaction filters were able to reduce the analysis time.

For the gravity experiment, the second interaction filter increased the required analysis time by 1.6% on average because it only filtered out interactions after the analysis and did not allow the analysis to ignore certain regions.

The remaining scenarios, however, all lead to an increased analysis speed because the used filters allowed the analysis to ignore certain BlameRegions. The first interaction filter of the gravity experiment reduced the analysis time by 13.1%. The two filters of the gzip experiment lead to an performance increase of 14.2% and 23.4% on average.

In summary, all three interaction filters that lower the number of taints that have to be created for the analysis were able to increase the speed of the analysis. Because of this, we accept RQ2.2.

5 Conclusion

In this chapter, we conclude the thesis by providing a summary of its contributions and describing some related work. We also present possibilities to further improve VaRA and solve some difficulties that we have encountered.

5.1. Summary

In this thesis, we have shown two approaches that improve VaRA's analyses. We introduced BlameRegions as a new region type for VaRA's commit interaction analysis. Compared to the previously used CommitRegion type, BlameRegions more accurately map a commit to the code regions that were changed by it, which increases the accuracy of the interaction analysis. We implemented a Git metadata interface for VaRA that can be used to extract additional metadata about the commits of a software project from its repository. This interface is also used by LLVM's frontend clang to find the BlameRegions in the analyzed source code. We additionally extended the libgit2 library—the Git library used by VaRA's Git metadata interface—with the ability to identify source code changes that have not yet been committed to the project's Git repository. In addition, we added a filter mechanism to VaRA's analysis that can be used to ignore BlameRegions in the analysis and to filter out interactions from the result based on different filter criteria (e.g., the commit author's name or the commit date). On the one hand this can increase the analysis speed, but on the other hand it can also reduce the number of detected interactions and the result size which can make it easier to interpret the result and answer a user's question.

To evaluate our approaches we analyzed three software projects, a small hand-crafted example and the two real-world software projects gzip and gravity. We found that interaction filters were able to reduce the number of reported interactions by up to 99.9% and the size of the result file by up to 93.7%. Interaction filters can also increase the speed of the analysis. In our evaluation, they were able to reduce the required analysis time by up to 23.4%.

5.2. Related Work

To the best of our knowledge, no previous works have used metadata of revision control systems to filter control-flow and data-flow analyses results or to increase the speed of a single, complete analysis run. However, Arzt and Bodden [AB14] presented an approach to use the change information of a revision control system to reuse data from a previous analysis run and thus save time. While this approach does not increase the speed of the first, complete analysis run, it can increase the speed of subsequent analyses of the same software project. They propose to first perform a complete data-flow analysis of a software project. If a commit is added at a later time, the commit can be analyzed to determine which parts of the program have been changed so that only a small part of the analysis result has to be recomputed and a large part can be reused.

5.3. Future Work

Interaction Filters

In Section 3.3.1 on page 27, we described that in some cases the interaction filters can lead to unexpected results. With the current implementation, when using an UnaryInteractionFilter like an AuthorFilter to ignore interactions in which a certain person is involved, VaRA removes all taints from the appropriate BlameRegions before the analysis. This removes all interactions from the analysis with the filtered BlameRegions as source of the interactions but the result still contains interactions with these regions as targets. That behavior can easily be solved by modifying the interaction filter to explicitly remove these interactions, but we would like to modify the filters' behavior in the future to make it easier for the user to get the expected result.

Larger case studies

We would like to perform VaRA's analyses on more case studies in the future, in particular on larger software projects to determine how well VaRA's analyses scale and whether interaction filters improve the scalability.

Whitespace Commits

When using VaRA's commit interaction analysis, we sometimes found commits with an exceptionally large number of interactions with other commits. Some of these commits, however, did not actually change the behavior of the analyzed source code. Instead, they merely reformatted large parts of the codebase which lead to these commits being reported by Git blame as being the last commits to modify these source code lines. In the future, we would like to add the ability to VaRA to ignore such commits in its analyses.

Appendices

A
BenchBuild Project Definition

```python
@with_git(
    "https://github.com/marcobambini/gravity.git",
    refspec="HEAD")
class Gravity(Project):
    NAME = 'gravity'
    GROUP = 'c_projects'
    DOMAIN = 'UNIX utils'
    VERSION = 'HEAD'

    BIN_NAMES = ['gravity']
    SRC_FILE = NAME + "-{0}".format(VERSION)

    def run_tests(self, runner: run):
        pass

    def compile(self):
        self.download()

        clang = cc(self)
        with local.cwd(self.SRC_FILE):
            with local.env(CC=str(clang)):
                cmake("-G", "Unix Makefiles", ".")
            run(make["-j", int(CFG["jobs"])])
```

Lst. A.1. Example of BenchBuild project definition

B
Calculator Case Study

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

Lst. B.1. Source Code of the Calculator Case Study

Tab. B.2. Description of the commits of the calculator example (from oldest to newest commit)

Nr.	Hash	Message	Diff

1	7a32a36	Added Main with no functionality.	`@@ -0,0 +1,5 @@`
			`+#include <stdio.h>`
			`+int main() {`
			`+  printf("Welcome to the Calculator!\n");`
			`+  return 0;`
			`+}`

2	5a09507	Added Add function.	`@@ -1,4 +1,9 @@`
			` #include <stdio.h>`
			`+`
			`+int add(int a, int b) {`
			`+  return a + b;`
			`+}`
			`+`
			` int main() {`
			`   printf("Welcome to the Calculator!\n");`
			`   return 0;`

3	7662b84	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");`
			`   return 0;`

4	60e791d	Added Console Read function.	`@@ -8,6 +8,12 @@ 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;`

5	6f3ec41	Added Print Op function.	`@@ -14,6 +14,11 @@ int readVariable() {`
			`   return res;`
			` }`
			
			`+void listOperations() {`
			`+  printf("0 - Addition\n");`
			`+  printf("1 - Subtraction\n");`
			`+}`
			`+`
			` int main() {`
			`   printf("Welcome to the Calculator!\n");`
			`   return 0;`

6	dd56d2a	Implemented main operation chooser.	`@@ -21,5 +21,13 @@ void listOperations() {`
			
			` int main() {`
			`   printf("Welcome to the Calculator!\n");`
			`+`
			`+  listOperations();`
			`+  printf("Choose operation: ");`
			`+  int v1, v2, op = readVariable();`
			`+  switch(op) {`
			`+    default:`
			`+      printf("Unknown operation: %d\n",`
			` ↪  op);`
			`+  }`
			`   return 0;`
			` }`

7	bb04138	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\n", add(v1, v2));`
			`+      break;`
			`     default:`
			`       printf("Unknown operation: %d\n",`
			`        ↪  op);`
			`   }`

8	83d9756	Implemented Sub handling.	
			```
			@@ -33,6 +33,13 @@ int main() {
			        v2 = readVariable();
			        printf("%d\n", add(v1, v2));
			        break;
			+     case 1:
			+       printf("Var 1: ");
			+       v1 = readVariable();
			+       printf("Var 2: ");
			+       v2 = readVariable();
			+       printf("%d\n", sub(v1, v2));
			+       break;
			      default:
			        printf("Unknown operation: %d\n",
			         ↪  op);
			    }
			```

9	oada3de	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: ");
			```

6

Bibliography

[Aho+06] Alfred V. Aho et al. “Compilers: Principles, Techniques, and Tools”. Second. Addison Wesley, 2006-09-10. ISBN: 978-0321486813 (*cited on pp. 4, 5, 6, 10*).

[AB14] Steven Arzt and Eric Bodden. “Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes”. In: *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014.* 2014, pp. 288–298 (*cited on pp. 49, 50*).

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. “A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis”. In: *ACM Trans. Program. Lang. Syst.* 19.6 (1997), pp. 992–1030 (*cited on p. 3*).

[Kin+08] Dave King et al. “Implicit Flows: Can’t Live with ’Em, Can’t Live without ’Em”. In: *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings.* 2008, pp. 56–70 (*cited on p. 7*).

[Lar18] Michael Larabel. “The Linux Kernel Gained 2.5 Million Lines Of Code, 71k Commits In 2017”. 2018-01-01. URL: `https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Commits-2017` (visited on 2019-08-28) (*cited on p. 1*).

[Lat12] Chris Lattner. “LLVM”. In: *The Architecture Of Open Source Applications.* Ed. by Amy Brown and Greg Wilson. Lulu.com, 2012-03-15. Chap. 11. ISBN: 978-1257638017 (*cited on pp. 9, 10, 11*).

[LLV19a] LLVM Project. “LLVM Language Reference Manual”. 2019-07-13. URL: `https://llvm.org/docs/LangRef.html` (visited on 2019-07-23) (*cited on pp. 10, 11, 12*).

[LLV19b] LLVM Project. “Writing an LLVM Pass”. 2019-07-23. URL: `https://llvm.org/docs/WritingAnLLVMPass.html` (visited on 2019-07-24) (*cited on p. 13*).

[Nie18] Florian Niederhuber. “Change-Region Detection in LLVM”. MA thesis. University of Passau, 2018-02 (*cited on pp. 14, 16, 36*).

[Sat17] Florian Sattler. “A variability-aware feature-region analyzer in LLVM”. MA thesis. University of Passau, 2017-03-20 (*cited on pp. 13, 14*).

[SHB19] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Tomáš Vojnar and Lijun Zhang. Cham: Springer International Publishing, 2019, pp. 393–410. ISBN: 978-3-030-17465-1 (*cited on p. 7*).

[Sim+16] Andreas Simbürger et al. "BenchBuild: A Large-Scale Empirical Research Toolkit". Tech. rep. MIP-1602. Faculty of Informatics and Mathematics, University of Passau, 2016-06 (*cited on p. 19*).

[Ske14] Ian Skerrett. "Eclipse Community Survey 2014 Results". Archived at `https://web.archive.org/web/20140625152145/http://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/`. Ianskerrett.wordpress.com. 2014-06-23. URL: `https://ianskerrett.wordpress.com/2014/06/23/eclipse-community-survey-2014-results/` (visited on 2019-07-12) (*cited on p. 7*).

[Sta18] Stack Overflow. "Stack Overflow Developer Survey 2018". Archived at `https://web.archive.org/web/20190711051752/https://insights.stackoverflow.com/survey/2018/`. 2018. URL: `https://insights.stackoverflow.com/survey/2018` (visited on 2019-07-12) (*cited on p. 7*).

[VaR19] VaRA Project. "VaRA-Tool-Suite". 2019. URL: `https://github.com/se-passau/VaRA-Tool-Suite/` (visited on 2019-07-28) (*cited on p. 19*).

Eidesstattliche Erklärung:

Hiermit bestätige ich, Julian Breiteneicher, dass ich die vorliegende Arbeit selbstständig und ohne unzulässige Hilfe verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die wörtlich und sinngemäß übernommenen Passagen aus anderen Werken kenntlich gemacht habe. Die Arbeit ist weder von mir noch von einer anderen Person an der Universität Passau oder an einer anderen Hochschule zur Erlangung eines akademischen Grades bereits eingereicht worden.

Passau, 02. September 2019

Julian Breiteneicher