OPTIMIZING INTRAPROCEDURAL STATIC ANALYSIS USING CALL-GRAPH INFORMATION AND MEMORY DEPENDENCIES

JAKOB SCHWARZWELLER

BACHELOR THESIS

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

Advisor: Florian Sattler, M. Sc.

Supervisor: Prof. Dr.-Ing. Sven Apel

March 28, 2018

ABSTRACT

Over time, the average complexity of software has increased drastically. This makes maintenance of software projects increasingly difficult. One recurring issue when changing code in a program is seeking what the impact of this change is. Because of growing complexity, giving a useful answer to this kind of question becomes more and more difficult, but is necessary to assist software developers.

This problem can be approached using static analysis of the flow of control and data in a program. However, this type of analysis can be very complex and thus take a long time. At the same time, it often yields very imprecise results [4], but even this can help in tracing down an interaction bug. Because this is favored over overlooking a possible interaction, we aim for providing measures to both improve the performance of such analyses and making their results more precise.

Specifically, we try to achieve that by incorporating call-graph information to optimize the order of execution of such analyses and use memory dependency information to improve its preciseness. We deploy this in practice by optimizing the Taint Flow Analysis featured in the VaRA LLVM framework as a proof of concept.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DAG directed acyclic graph

IR Intermediate Representation

GZIP GNU zip

SCC strongly connected component

SSA static single-assignment form

VaRA Variability-aware Region Analyzer

1 INTRODUCTION

A common problem when developing software is seeking what the implications of a change are. This question is increasingly difficult to answer, due to the heavy complexity in modern software systems – even a small change to one part of a program can have an effect on a entirely different part of it.

Static analysis can give guidance on such issues, by trying to determine interactions in a program, thus figuring out parts of a program influenced by a change, in order to support the programmer. One framework that can be used for that purpose is the Variability-aware Region Analyzer (VaRA) framework for LLVM. It provides an interface to run arbitrary analyses over regions of code defined by a user of the framework. One use case could be to define regions that are relevant to a feature (i. e., a characteristic or end-user-visible behavior of a system) and then use an analysis to try and find out what other parts of the code base are influenced by it. The VaRA framework does already offer analyses and regions for that purpose: a region type (`FeatureRegion`) to represent a feature and a Taint Flow Analysis.

However, static analysis for larger code bases is expensive and imprecise. For example, Heckman and Williams found in a literature review of static analyzers meant to identify potential source code anomalies that somewhere between 35 % and 91 % of anomalies reported by these analyzers were deemed unimportant by developers [4]. Hence we need to increase the analysis's performance, so we can analyze real-world applications. Furthermore, we need to make it precise to reduce the amount of miss-predictions.

In this thesis, we propose two ways of improving this type of analysis, one that increases performance and one that increases accuracy.

1.1 GOALS

Our goal is to improve the performance of VaRA's Taint Flow Analysis by reducing the amount of re-evaluations necessary using call-graph information. Furthermore, we aim for increasing the accuracy of that analysis by considering memory dependency information.

1.2 CONTRIBUTION

We approach the performance improvement in Chapter 3 by ordering the evaluation of functions according to the call graph. Because the data-flow relations of the callee are not influenced by the analysis

result of the caller this reduces the need for re-evaluation of functions due to information about influences having changed.

The accuracy improvement in Chapter 4 is achieved by taking into account memory dependencies between instructions. In some cases, the data-flow predecessor of a instruction can unambiguously be determined. We enhance the data-flow analysis, so that in cases where we can determine a direct memory dependency, the analysis only considers the direct predecessor.

1.3 OVERVIEW

In Chapter 2, we provide an overview of frameworks we later build upon and introduce key concepts. We introduce the two frameworks LLVM and VaRA and LLVM's call-graph representation, further we explain the concepts of Memory-Dependency Analysis and static data-flow analysis; particularly how they are used to build a Taint Flow Analysis. The contribution is divided into two parts: In Chapter 3, we discuss a performance improvement by ordering the examination by the structure given by the call graph. In Chapter 4, we improve the accuracy of the Taint Flow Analysis by considering memory dependencies between instructions. In Chapter 5, we evaluate the effectiveness of these modifications. Finally, in Chapter 6, we summarize our efforts and their impact.

2 BACKGROUND

In this chapter, we introduce the LLVM framework and explain relevant aspects of the framework's Intermediate Representation (IR), as well as its call-graph representation and its Memory-Dependency Analysis. Then we introduce the VaRA framework, an extension of LLVM. After that, we discuss the concepts of data-flow analysis and how to use them to build a Taint Flow Analysis.

2.1 LLVM

LLVM is modern compiler framework that gives its users useful toolchain technologies (e. g., data structures and analyses) to build compilers and related tools. It supports a number of different languages (through so-called *frontends*) and target architectures (via *backends*) by using a decoupled approach: Frontends compile code into an Intermediate Representation (IR). Language-independent optimizations or analyses are then performed on that IR code. At the end of a compilation process, a backend compiles the IR into native code for the target architecture.

This architecture has an important advantage: Any optimizations or analyses that run on IR can be used on any software project regardless of programming language or target architecture, as long as they are supported by LLVM. For adding support for a new programming languages, solely a compiler from that language into IR must be written, and all other components (including the backends for compiling for a variety of platforms) can be re-used. The same is true for adding support for a new target platform [6].

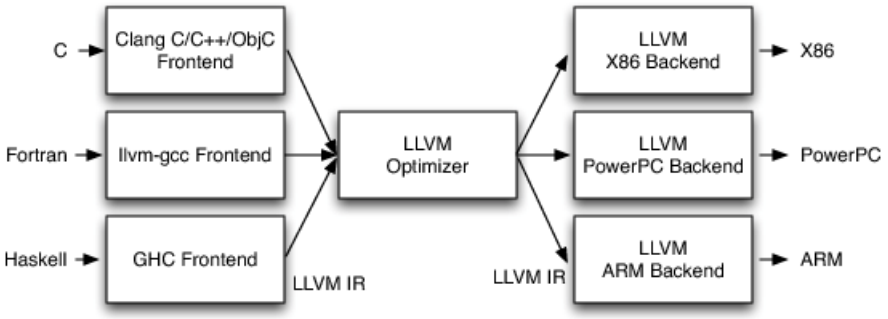

Figure 2.1: A schematic portrayal of LLVM's three-stage design [5, Figure 11.3]

2.1.1 *LLVM-IR*

IR is the intermediate representation that LLVM uses for code. It is designed to be both flexible – so it can represent code by lots of different programming languages and for lots of different targets – and simple, to ease human-readability. The code for an application may be separated into a set of *modules* (similar to C++ translation units). In Listing 2.1 a small example program in C can be seen, Listing 2.2 shows the correspondent IR.

```c
int main () {



    int a = 1337;

    a += 23;

    return 0;
}
```

Listing 2.1: C example

```
1  define i32 @main() #0 {
2  entry:
3      %retval = alloca i32, align 4
4      %a = alloca i32, align 4
5      store i32 0, i32* %retval, align 4
6      store i32 1337, i32* %a, align 4
7      %0 = load i32, i32* %a, align 4
8      %add = add nsw i32 %0, 23
9      store i32 %add, i32* %a, align 4
10     ret i32 0
11 }
```

Listing 2.2: Listing 2.1 in IR

The C program declares a variable `a`, initializes it with `1337`, adds `23` and return `0`. In the IR listing, the declaration of the variable `a` using the **`alloca`** instruction can be seen in Line 4. Then, in Line 6, the **`store`** instruction is used to initialize the variable.

Lines 7–9 in the IR listing show the adding process. First, the **`load`** instruction is used to load the current value of `a` into the register `%0`. In the following line, the **`add`** instruction is used to add `23` to that value and stores the result in the register **`%add`**. Lastly, the result is written to the `a` variable using the **`store`** instruction.

There are multiple properties of IR that can be seen here. For once, the basic structure of loading the value of a variable into a register, do computations and storing the result in variables again. Also, that results of computations are stored in a new register rather than the input register. The reason for this is that IR is in static single-assignment form (SSA). This means that every register is only written to once and is only read after that.

A notable instruction to look at in more detail is the **`store`** instruction, which represents saving a value in memory. Its simplified general syntax can be seen in Listing 2.3. It takes two operands and their corresponding types, `%val` and `%pointer`. `%val` is the value to be saved in memory, while `%pointer` is the address in memory to save

the value at. A **store** instruction may have more options passed than shown in Listing 2.3; those are not relevant for our work, however.

```
store <val type> %val, <pter type>* %pointer
```

Listing 2.3: Simplified general syntax of the **store** instruction

Another important instruction is the **load** instruction, which represents loading a value from memory. Its general syntax can be seen in Listing 2.4. It takes one operand plus its type, `%pointer`, and returns the value at this address. A **load** instruction may have more options passed than shown in Listing 2.4; those are not relevant for our work, however.

```
%res = load <res type>, <pter type>* %pointer
```

Listing 2.4: Simplified general syntax of the **load** instruction

2.1.2 *Memory-Dependency Analysis*

A value loaded from memory by a **load** must have been written there by an **store** instruction. Such an relationship between two instructions is called a *memory dependency*; an analysis that aims to find such relationships is called *Memory-Dependency Analysis*. An example for such a memory dependency can be seen in Listing 2.5: The value written by a **store** is read by a **load** immediately afterwards. This a common case for memory dependencies in IR: a **load** on a memory location that is not separated from the previous **store** by any control-flow change (e. g., branching). Such a memory dependency can easily be determined by iterating through the instruction before or after the considered **load** or **store**.

LLVM features a Memory-Dependency Analysis, which can in some cases – including the one explained above – unambiguously determine the preceding **store** for a given **load** or vice versa. It returns its results for a function as a `MemoryDependenceResults` object, which in turn has methods to get the results for a particular instruction as a

```
...
store i32 42, i32 * %a
%0 = load i32, i32 * %a, align 4
...
```

Listing 2.5: Example for a memory dependency

`MemDepResult` object. This result can be, among others, that it has a definitive memory dependency, in which the case the corresponding instruction can be obtained.

2.1.3 *LLVM pass infrastructure*

In LLVM, analyses and optimizations are structured into so-called passes. Generally LLVM passes are separated into two categories: Transformation passes for optimization (e. g., elimination of dead code) and analysis passes for analyses, the results of which may be used by other passes.

There are different types of passes that differ in the nature of the analysis they perform. Most of the types can be distinguished by the type of element that the pass's main function is processing, such as `ModulePass`, `FunctionPass` or `LoopPass`. These passes do not guarantee any specific order of execution. Some passes, however, differ in way or order of processing, such as the `CallGraphSCCPass`, that iterates the call graph of a program bottom-up (i. e., callees before callers).

A pass can also require the results of other passes, even of different types. This requires a comprehensive way of ensuring order of execution for passes, so the results of a required pass are there when needed. This is done by a `PassManager`. An implementation of a `PassManager` comes with the LLVM framework. When adding a pass to LLVM, it must be registered at this `PassManager`.

In order to establish its dependencies, a pass may implement a `getAnalysisUsage()` method. The `PassManager` will then call this method with an `AnalysisUsage` object. The pass can then communicate its dependencies by calling the `addRequired` method on this object. Furthermore, it can declare whether it will change data that will invalidate other analysis's results by calling appropriate methods on this `AnalysisUsage` object.

2.1.4 *LLVM call-graph representation*

A *call graph* is a representation of relationships between the functions of a program. Its nodes are the functions; edges are drawn from the functions that call other functions to those that they call. As follows, a call graph is directed and contains cycles, if the program has recursive calls.

A further concept is that of a *SCC call graph*, which is a graph of strongly connected components (SCCs) from the call graph, called `CallGraphSCC`s in LLVM. A SCC is a sub-graph in which all nodes are reachable by each other node. A node of a SCC call graph can therefore either consist of only one function (in which case the transformation from call-graph node to `CallGraphSCC` node is trivial) or it

```
int main() {
___REGION_START __RT_Commit "main"
  int a = 1;
  int b = a + 42;
___REGION_END __RT_Commit "main"
}
```

Listing 2.6: Example for region annotation

can consist of multiple nodes from the call graph. If it consists of multiple nodes, it is a sub-graph of the call graph that contains a cycle. By grouping cyclic sub-graphs of the call graph into SCCs, the SCC call graph becomes acyclic and thus a directed acyclic graph (DAG) [3].

2.2 VARIABILITY-AWARE REGION ANALYZER

The Variability-aware Region Analyzer (VaRA) is an extension of the LLVM framework that adds the concept regions of interest. This regions are source code regions that have semantic meaning for the programmer. VaRA has the abstract interface `IRegion` that represents a generic region. Researchers can write analyses that work on this interface and are therefore independent of the actual type of region that the analysis runs on. Developers, in turn, can implement the `IRegion` interface and run any analysis that has been written for VaRA `IRegions` over their implementation.

An example usage of this framework is to create `FeatureRegion` as an implementation of `IRegion`. A `FeatureRegion` is a piece of code that is related to a specific feature of the application; a feature being defined as "a characteristic or end-user-visible behavior of a software system" [2]. Then an analysis that tries to detect interactions between those regions can be used to try to find interactions between different features of a software. A feature region is a region in an application's code that gets executed only when a specific feature (or combination of features) is activated [9].

Another usage that is already a part of VaRA is the implementation of a `MarkerRegion` `IRegion` type. This type is capable of extracting region information from suitable meta-data in IR modules. This in turn may be placed there using a modified version of a LLVM frontend. There is a modified version of the C/C++ frontend Clang that can process region information given in C/C++ code using newly added keywords `___REGION_START` and `___REGION_END` (as can be seen in Listing 2.6). An analysis in VaRA that aims for finding interactions between such regions is called Marker Flow Analysis [7].

The goal of VaRA is to allow developers and researchers to analyse and reason about their regions of interest.

2.3 TAINT DATA-FLOW ANALYSIS

The Taint Flow Analysis is a data-flow analysis. For data-flow analyses, a representation of every possible program state at a certain point during the execution is called a *data-flow value*. A program is then viewed as a sequence of transformations from one data-flow value to another. The data-flow values before an instruction s are denoted with $IN[s]$ and those after it with $OUT[s]$. There are two constraints on the transformations between those: those depending on the semantics of the instruction and those depending on the flow of control.

The semantics of an instruction s dictate the relation between of $IN[s]$ and $OUT[s]$, depending on those, data-flow values get eliminated or introduced. This is done by a so-called *transfer function*. The control flow defines the relations between $OUT[s]$ for an instruction s and the IN of any subsequent instructions. A sequence of instructions that will only begin execution at its start and only branch at its end is called a *basic block*. Inside a basic block, that simply is $IN[s] = OUT[s-1]$ (as long as both s and $s-1$ are inside the basic block).

Let us now also introduce $IN[B]$ and $OUT[B]$ for any basic block B. The relations between $IN[B]$ and $OUT[B]$ for the same basic block B can simply be derived from the instructions inside that basic block.

For the construction of $IN[B]$ we define a *meet* operator on the data-flow values, $\wedge$, with the following properties:

1. Idempotency: $x \wedge x = x$
2. Commutativity: $x \wedge y = y \wedge x$
3. Associativity: $x \wedge (y \wedge z) = (x \wedge y) \wedge z$

For a basic block B, $IN[B]$ can then be defined as follows [1]:

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

For *monotone data-flow frameworks* this meet operator must fulfill

For all data-flow values x, y and transfer functions f :

$$x \leq y \Rightarrow f(x) \leq f(y)$$

where

$$x \leq y :\Leftrightarrow x \wedge y = x.$$

A use case for data-flow analysis is Taint Flow Analysis. For this analysis, the data-flow values are *taints*: instructions get tainted with a specific taint, then those taints get propagated through the program,

tainting every instruction they touch. A taint relates to a semantic fact (e. g., a variable getting influenced by a configuration option). Propagating the taint along the analysis shows which other instruction get tainted (i. e., are influenced by the configuration option).

For example, in VaRA, every instruction in a `FeatureRegion` is tainted with a taint belonging to that region. Then, after the analysis has propagated the taints through the program, we can reason about which other statements get influenced by the `FeatureRegion`.

VaRA's Taint Flow Analysis is based on a monotone framework. Its data-flow values are subsets of the set of all taints occurring in the analysis, meet is the union operator, $\cup$.

The Taint Flow Analysis in VaRA adopts this approach by considering a *def-use graph* (definition-use graph) with the instructions as nodes. Since IR is in SSA-form, the definition of every value can be described with the instruction that defined it. The edges of the def-use graph are the memory relations (both usages of registers and usages of memory locations) between those instructions. This optimizes the data-flow analysis, because every $IN[s]$ set only contains data that is directly relevant to the instruction s.

In code, the edges in the graph are indirectly defined by iterators, via LLVM's so-called `GraphTraits`. When the analysis needs the incoming edges of an instruction (its so-called *sources*) it creates a source iterator for that instruction. This iterator is then used for constructing the IN set for that instruction. Likewise, when the analysis needs the outgoing edges of an instruction (*sinks*), it creates a sink iterator for that instruction.

3 CALL-GRAPH-ORDERED PROCESSING

In this chapter, we introduce our proposal for improving the performance of the analysis by considering call-graph information, detail the changes made to VaRA to incorporate this proposal into the framework and show its impact using an example.

3.1 REASONS

Ordering the processing by the call-graph structure reduces the need for reprocessing of functions. Generally, functions that interact with each other may influence each others analysis results, more precisely, the functions that get called by a function influence the results of the caller function. If the analysis results of a function change or are only becoming available in the midst of an analysis run, all callers of this function need to be re-evaluated.

Because this only affects callers of a function and not callees, changing the processing order to try to examine callees before callers will reduce the amount of re-evaluations necessary. Also, since as of now the order of examination of functions is up to the LLVM `PassManager`, which does not guarantee any particular order of execution, changing the order will not affect the final analysis results, but can affect the run time of the analysis.

3.2 IMPLEMENTATION

In order to achieve this call-graph order, we impose a partial order on the nodes of the SCC call graph. This partial order is defined as follows:

> For all x, y nodes of the SCC call graph:
> $x \leq y :\Leftrightarrow y$ can be reached from x

The order of evaluation of SCCs then follows this partial order. Analysis on this SCCs themselves is as follows: Either the SCC consists of only one function; then this function is evaluated.

If the SCC consists of more than one function, it is a sub-graph. This sub-graph is acyclic, since if it was it would be divided further into SCCs. A cyclic call graph, however, means that there are recursive calls between at least some of the functions in the graph, which means that those functions can influence each other's analysis results recursively. In this case, call-graph information cannot be used to determine in which order functions should be evaluated. For this reason, functions

in a multiple-element SCCs are evaluated in arbitrary order, which is the current guarantee for the analysis order. Therefore, this does not introduce imprecisions, but falls back to the unoptimized version.

3.2.1 *Changes to VaRA*

In order to achieve this execution order, the first step is to to copy the evaluation from a `FunctionPass`, where the `PassManager` decides on the order of evaluation of the functions, to a `ModulePass`. This also involves elevating previously local variables to members of that pass class. To reduce code duplication, we moved the analysis into a separate analysis class, which then gets re-used by both the original analysis and our new pass.

The reason that we want to keep the former analysis is, that the current `PassManager` only allows using passes as dependencies that are on the same level or lower. For example, this means that a `FunctionPass` cannot require a `ModulePass`. Therefore, we want to preserve the old `FunctionPass` analysis, so other `FunctionPasses` can still request it.

Now we have to iterate over the functions in the module according to the partial order. LLVM provides the analysis `CallGraphWrapperPass`, which calculates the call graph for a module. We can access the call graph that the analysis generates using the `getAnalysis()` method, after having registered a dependency on that analysis by calling `AnalysisUsage.addRequired()` in the `getAnalysisUsage()` method. We can then iterate over the SCC call graph by calling LLVM's `scc_begin()` function on the call graph. Finally, instantiating `CallGraphSCC` with the call graph and the `scc_iterator` as arguments yields the current SCC call-graph node. We can then iterate over the functions in that node and run the analysis for each of them.

3.2.2 *Example*

In Listing 3.1, we see a C program featuring two functions, `main` and `foo`. The function `main` calls `foo`, `foo` thus possibly influencing the analysis result for `main`. But `foo` is declared only after `main` (its usage in `main` is only possible by using C's forward declaration). Let us assume for simplicity, that this order of declaration is also in place in an IR translation of this program and that LLVM's `PassManager` will execute `FunctionPasses` in this order, which is the current behavior.

If the evaluation of this program is done using the order specified by the `PassManager`, this means: When `main` is analyzed, the analysis results for `foo`, which those of `main` depend on, will not be available, yet. Because of this, `main` has to be to analyzed again after `foo` has been analyzed. This results in a total of three function analyses executed.

```c
int foo(int a);

int main() {
    int a = 3;
    int b = foo(a);
    return 0;
}

int foo(int a) {
    return a*2;
}
```

Listing 3.1: C example with one function being used before it is declared

In the call graph, however, there is an edge from `main` to `foo`, because `main` calls `foo`. But there are no edges leading away from `foo`. This means, that in the partial order defined above, `foo` comes before `main`.

So, when the order of evaluation is determined using that order, as our newly added pass does, `foo` will be evaluated before `main`. The function `foo` does not call other functions, so its analysis does not depend on any. This means that `foo`'s analysis can be completed in one run.

Now, when the `main` function is analyzed next, all information about `foo` is already available. Because of that, `main`'s analysis can be completed in one go, as well. As a result, the analysis is completed after two function analyses have been run.

4 MEMORY-DEPENDENCY ANALYSIS

In this chapter, we introduce our proposal for improving the accuracy of the analysis by considering memory-dependency information, detail the changes made to VaRA to incorporate this proposal into the framework and show its impact using an example.

4.1 REASONS

In IR program code, a repeating pattern is loading the one or more values from memory using the **load** instruction, doing computations with those values and writing the result back into memory with the **store** instruction. When the Taint Flow Analysis encounters such a pattern, it determines the taints attached to the values loaded by checking **store** instructions that write to the corresponding memory location for the taints of the values those instructions have written.

After having determined the taints of the loaded values this way, it propagates those taints through the computations. This is usually straightforward (e. g., the taints of the result of an arithmetic instruction with two operands are the union of the taints of the operands). When this propagation reaches a **store** instruction, its taints are remembered for when a **load** instruction loads a value from this address.

A potential problem with this modus operandi is obvious: Not all **store** instructions that ever write to a memory location might be the last **store** instruction to write to this location before any specific **load** instruction. In other words: if a **store** instruction can never be the last point at which a certain memory location was written to before a specific **load** instruction, than the value that this **load** loads will never have been written by this **store** instruction. Hence the taints, that the value that **store** instruction wrote to memory had, are irrelevant for that **load** instruction and should not be propagated.

Also, if we could reduce the amount of **store** instructions having to be considered when determining the taints of a value loaded by a **load**, then the taints of the result of a computation written back to memory could potentially be reduced. So, if we could determine which **store** the value a **load** loads comes from, this could reduce the amount of taints that values in the program have and thus increase the accuracy of the analysis.

This is why we try to determine the preceding **store** to the **load**s in a program using Memory-Dependency Analysis and, if it can be

unambiguously determined, use only this **store** as a source for the taints for this **load**.

4.2 IMPLEMENTATION

As of now, propagation of taints from **store**s to a memory location to **load**s from that memory location is happening via the **alloca** instructions for that memory location. This means that any taints that end up at a **store** instruction are propagated to the **alloca** instruction and are then distributed to the **load** instructions from there, introducing imprecision in the analysis. This means that there is no direct relationship from **load** to **store** instructions.

So in order to be able to respect memory dependencies between **load** and **store** instructions, we need **load** instructions to be aware of the **store** instructions they are originally getting their taints from. For this purpose, we modify the def-use graph of the Taint Flow Analysis so, that the sinks of the **store** instructions are no longer the corresponding **alloca**s, but instead the **load** instructions. So when iterating the sources of a **load**, we now directly iterate over the corresponding **store** instructions.

Having established these relationships, we can now filter out any unwanted predecessors of the **load** instructions. For all possible sources for an instruction as returned by the source iterator, we call a filter method that takes the current instruction and a sources for that instruction. This filter then checks if it can make a decision based on memory dependencies, which it can only do if the current instruction is a **load** instruction and there is a definitive memory dependency. If it can, it will then notify the analysis to ignore all instructions that are not this definitive dependency. Similarly, we install a filter for the sinks of instructions.

4.2.1 *Changes to VaRA*

In order to add this filtering to VaRA, we first have to establish the aforementioned direct relationship between **store** and **load** instructions. This can be achieved be changing the `StoreSinkIter` and adding a `LoadSourceIter` to the `GraphTraits` of `FunctionDefUseGraph`. `FunctionDefUseGraph` is the graph implementation that VaRA's Taint Flow Analysis works on.

The `StoreSinkIter` is instantiated with a **store** instruction (`StoreInst`) and then iterates over all the sinks (i. e., the places where the taints of a **store** instruction may be propagated to) of this store instruction. Similarly, the `LoadSourceIter` iterates over all sources for a **load** instruction (i. e., all the places where the taints of a **load** instruction may come from).

```
bool MemoryDependenceSourceFilter::operator()(
    NodeRef Current, NodeRef Node) {
  if (!isa<AllocaInst>(Node)) {
    if (LoadInst *I = dyn_cast<LoadInst>(Current)) {
      const MemDepResult &R = MDR.getDependency(I);

      if (R.isDef()) {
        return R.getInst() != Node;
      }
    }
  }

  return false;
}
```

Listing 4.1: Implementation of source filtering

Next, we have to build the necessary infrastructure for integrating the Memory-Dependency Analysis. The `FlowAnalysis` class already has an interface to register source and sink filters. However, since the `TaintFlowAnalysis` class does not inherit from that class, but just uses it internally, we have to build a similar interface for registering and storing such filters there. Then, when the `FlowAnalysis` is instantiated in the `TaintFlowAnalysis`'s `run` method, we register all the stored filters on that instance right before the analysis starts.

We also have to introduce classes that actually do the filtering, `MemoryDependenceSinkFilter` and `MemoryDependenceSourceFilter`. These get passed the `MemoryDependenceResults` for the analyzed function to the constructor and act as a filter function by overloading the `()` operator.

As a last infrastructure step, we have to register the memory dependency filters whenever the analysis is used. Usually, this happens in so called wrapper passes. These wrap the analysis in `FunctionPasses`. Whenever another analysis needs the Taint Flow Analysis, it requests the wrapper pass's results. This way, scheduling can be left to the `PassManager`. An example of such a wrapper pass is the `CommitTaintFWrapperPass` for the Marker Flow Analysis.

With the groundwork laid, the actual filtering is quite simple, as can be seen in Listing 4.1, which shows the filtering in `MemoryDependenceSourceFilter`. We first check, whether the source is not a **`alloca`** instruction, since these are always valid sources for taints. Then we check whether the current instruction is a **`load`** instruction. Only if it is, we try to filter (the filter has to return `true` for a source to be dropped).

We then get the Memory-Dependency Analysis for that `load` instruction. If this analysis has a definitive result (i. e., the `store` instruction that wrote to that memory location most recently can unambiguously be determined) then we filter out every instruction that is not this definitive dependency. If the Memory-Dependency Analysis result is ambiguous, we don't filter out any sources.

The `MemoryDependenceSinkFilter` works in the same way, except for testing for a `store` instruction and the roles of `Current` and `Node` being reversed.

Furthermore, we implement a statistic feature for evaluation. The LLVM framework has support for statistic variables, which makes integration easy. We define four statistic variables using LLVM's `STATISTIC` macro: a counter for the filtered nodes and those that were not filtered, for both sources and sinks. We then increment these counters at the appropriate places in the filtering process in the `FlowAnalysis` class. Output of those values is then controlled by LLVM.

4.2.2 *Example*

In Listing 4.2, we see a small example program in IR to demonstrate the behavior of our changes. A memory location is allocated then a value is stored and loaded from it two consecutive times. Each time the value is written, though, a different taint is applied to the `store` instruction.

```
define i32 @main() #0 {
entry:
  %a = alloca i32, align 4

  store i32 42, i32* %a, !FVar !0
  %0 = load i32, i32* %a, align 4

  store i32 0, i32* %a, !FVar !1
  %1 = load i32, i32* %a, align 4

  ret i32 0;
}

!0 = !{!"Foo"}
!1 = !{!"Bar"}
```

Listing 4.2: Memory dependence example

Previously, this would have led to both `load` instructions being tainted with both taints. Now, when first `load` asks for its sources,

initially both **store**s and the **alloca** are returned. Then the filtering is triggered; the only filters that are registered are our new memory dependency source and sink filters. For illustration, let us look at what the source filter does in detail. It first has to decide whether the **alloca** as a valid source for the first **load**. Since the filter does always consider **alloca** instructions to be valid sources, it is not filtered out.

Then, the filter is asked whether the first **store** is a valid source. Since the current instruction is a **load**, and the memory dependency has found a definitive dependency, it checks whether the first **store** is this dependency. Since it is, the filter return `false`. Finally, the filter is asked whether the second **store** is a valid dependency for the first **load**. Again, the filter knows that there is a definitive dependency. But since the second **store** is not this dependency, it is filtered out.

Similar steps happen for the filtering for the second **store**. In Listing 4.3 the full result after our changes can be seen: both **load** instructions are only tainted with the taint of the previous **store** instruction.

```
%a = alloca i32, align 4 T: {}

store i32 42, i32* %a, !FVar !0 T: {Foo }
%0 = load i32, i32* %a, align 4 T: {Foo }

store i32 0, i32* %a, !FVar !1 T: {Bar }
%1 = load i32, i32* %a, align 4 T: {Bar }

ret i32 0 T: {}
```

Listing 4.3: Memory dependence example result

5 EVALUATION

In this chapter, we evaluate our optimizations, using statistical output of the analysis. For this purpose we use a real-world program, the compression tool GNU zip (GZIP), as a case study.

5.1 CALL-GRAPH-ORDERED PROCESSING

Here, we evaluate our new call-graph-ordered evaluation. We first take a look at a small example already introduced during the implementation of the optimization. Then, we use a case study to further examine its effectiveness.

5.1.1 *Basic functionality*

```
int foo(int a);

int main() {
___REGION_START __RT_Commit "main"
    int a = 3;
    int b = foo(a);
    return 0;
___REGION_END __RT_Commit "main"
}

int foo(int a) {
___REGION_START __RT_Commit "foo"
    return a*2;
___REGION_END__RT_Commit "foo"
}
```

Listing 5.1: C example with one function being used before it is declared

Let us recall the example used to illustrate our changes in Section 3.2.2: A small C program with one function used before it is declared. It can be seen again in Listing 5.1, but instrumented with region statements for the Marker Flow Analysis. These region markers are used to denote the start and end of region that is relevant for the respective analysis. When compiling the program with a modified version of LLVM's C frontend Clang, the region markers get translated to taints in IR.

Remember, that before the introduction of call-graph-ordered processing of functions, `main` will be evaluated first, followed by `foo`. When we run that analysis on this example and let it print statistics, we get the results seen in Table 5.1. This shows us, that during the evaluation of the `main` function, when results for `foo` where not yet available, the result taint of `foo` was accessed 3 times, until its placeholder was eventually resolved when `foo` was analyzed.

STATISTIC	VALUE
Number of placeholder return taints needed	3
Number of placeholders resolved	1

Table 5.1: Statistic output for the Marker Flow Analysis of the simple example without call-graph-ordering

With our new call-graph-sorted analysis order, however, `foo` is evaluated before `main` leading to the results seen in Table 5.2: Again, the result taints of `foo` are used 3 times for the analyses of `main`, but this time, they are already available at all those occasions.

STATISTIC	VALUE
Number of early resolved return taints	3

Table 5.2: Statistic output for the Marker Flow Analysis of the simple example with call-graph-ordering

5.1.2 *Case study*

Now, we evaluate our optimization with a larger case study. We choose GNU zip (GZIP), as this was already the case study of choice in the thesis of Niederhuber [7] when introducing the Marker Flow Analysis. We now use their tool-chain to annotate GZIP's code with commit information. Running the Marker Flow Analysis both with the previous evaluation order and with call-graph-ordering, yields the result displayed in Table 5.3.

STATISTIC	BEFORE	WITH ORDER	CHANGE
Early resolved ret. taints	184	754	570
Placeholder return taints	3,104	2,411	−693
Placeholders resolved	238	52	−186

Table 5.3: Comparison of statistics for the Marker Flow Analysis of GZIP with and without call-graph-ordered evaluation

As can be seen, the amount of placeholder taints needed drops from 3,104 to 2,411 or by 22.33 %. Simultaneously, the count of early resolves rises from 184 to 754 (or by 309.78 %). This suggest a decrease of function evaluation runs, as evaluation results are already available when they are needed more often. Consequently, the amount of times a placeholder has been resolved (i. e., when a later function analysis run triggers a re-evaluation of another function) declines from 238 to 52 (by 78.15 %).

The reason that not all placeholder return taints are resolved is due to dependencies outside of the current program (e. g., calls to libraries). Those called functions are never analyzed, hence their placeholders never get resolved. The reason that the amount of placeholders that are not resolved changes is due to a quirk in statistic collection: The placeholder counter is increased every time the result taint of function is requested but not available, the resolve counter is only increased once when the function analysis result is available.

5.2 MEMORY-DEPENDENCY ANALYSIS

For the evaluation of the integration of Memory-Dependency Analysis filtering, we again use GZIP as a case study. We look at the filtering statistic output added by us. This statistic's output for running the analyses both with and without memory dependence filtering enabled is shown in Table 5.4. No other filters were activated during the evaluation, so all filtering is done by our filters.

STATISTIC	BEFORE	WITH FILTERING	CHANGE
Sources not filtered	57,948,198	2,779,754	−55,168,444
Filtered sources	–	953,776	953,776
Total sources encountered	57,948,198	3,733,530	−54,214,668
Sinks not filtered	515,005	74,725	−440,280
Filtered sinks	–	3,795	3,795
Total sinks encountered	515,005	78,520	−436,485

Table 5.4: Comparison of statistics for the Taint Flow Analysis of GZIP with and without filtering using memory dependencies

As can be seen, with the filters activated, 3,733,530 sources were encountered, of which 953,776 (or about 25.55 %) were filtered out. At the same time, 78,520 sinks were encountered, of which 3,795 (or about 4.83 %) were filtered out, however, only filtered sources are relevant for the analysis accuracy. Yet, the share of filtered sources is not

directly translatable to a accuracy improvement: This count is also increased for intermediate results and an equal distribution of sources that can be filtered is neither guaranteed, nor likely.

Still, this drop suggests a moderate increase of accuracy: as we showed in our small test example in Section 4.2.2, our filters prevent unnecessary taint propagation, which reduces over-approximation and, therefore, increases accuracy. We verified soundness of our optimizations with a small set of example test cases, but would recommend follow-up work that tries to evaluate this on larger case studies.

Also catching attention is the huge decline of sources encountered with the filters activated, from 57,948,198 to 3,733,530, a drop of about 93.56 %. This is likely due to the following: Every time a source is filtered, the amount of instructions that need to be queued for re-evaluation because the analysis results may have changed, get reduced. In addition, for every instruction that does not need to be queued, their dependencies do not have to be queued, as well, leading to a ripple effect.

This, of course, also suggests a performance boost: If less sources have to be queued, this means less resources have to be used to process all the sources. In fact, naïve run-time evaluations on our two subject systems show a decrease from 145.71 s total (user plus system) CPU time to 13.56 s (a 90.69 % drop) on one system and from 5.96 s to 0.48 s (91.95 %) on the other system for running the Taint Flow Analysis on GZIP. For evaluation we used two subject systems: The first system has a Intel Core i3-3110M @ 2.40 GHz with 2 cores, 4 hyperthreads and 4 GB RAM and runs Ubuntu 14.04. The second system has a Ryzen Threadripper 1950X @ 3.4 GHz with 16 cores, 32 hyperthreads and 32 GB RAM and runs gentoo. Furthermore, we repeated our measurements 5 times and used the average as result to reduce the impact of system overhead.

6 CONCLUSION

We conclude this thesis by summarizing the optimizations we proposed and their impact. Furthermore, we discuss ideas to improve them in future work.

6.1 SUMMARY

In this thesis, we introduced two optimizations for a Taint Flow Analysis and implemented them in the VaRA framework:

The first optimization was using call-graph information for improving the order of execution of the analysis. The aim of this was to increase analysis performance by reducing the amount of re-evaluations needed. The second optimization was the integration of a Memory-Dependency Analysis into the Taint Flow Analysis, in order to reduce the amounts of taint sources considered and thus improve the analysis's accuracy.

Evaluation showed that taking call-graph information into consideration when determining the execution order proved to cut the amount of re-evaluations needed considerably, by 78.15 %, with an increase of evaluation results being already available when needed by 309.78 % in our case study. The evaluation results for the integration of Memory-Dependency Analysis suggest a moderate increase in accuracy, with about 25.55 % of taint sources being filtered out. Remarkably, it also led to a substantial performance improvement.

6.2 FUTURE WORK

UNWIND RECURSION IN SCCS Our optimization so far uses conversion of the recursive elements in a call graph into SCCs, in order to be able to impose a partial order on the resulting SCC call graph. A further step would be to (possibly using heuristic methods) try to find an optimal order of evaluation for the recursive elements of a call graph.

BETTER MEMORY-DEPENDENCY ANALYSIS The Memory-Dependency Analysis currently used by our implementation is the one provided by the LLVM framework. As it is mostly used during the compilation process, where demands on execution speed are higher than during an extensive analysis. Because of this, it is trimmed for speed and often fails to find a memory dependency, even if there is a definitive one. Using a Memory-Dependency Analysis that tries harder to

find memory dependencies could further improve the accuracy of the Taint Flow Analysis. Currently, LLVM is preparing a Memory SSA analysis [8]. This will provide an SSA form for memory, allowing users of the framework to reason about relationships between definitions and usages of memory.

BIBLIOGRAPHY

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Second Edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2007. ISBN: 0321486811.

[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013. ISBN: 978-3-642-37520-0.

[3] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. "Practical and Accurate Low-Level Pointer Analysis." In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 291–302. ISBN: 0-7695-2298-X. URL: http://dx.doi.org/10.1109/CGO.2005.27.

[4] Sarah Heckman and Laurie Williams. "A systematic literature review of actionable alert identification techniques for automated static code analysis." In: *Information and Software Technology* 53.4 (2011), pp. 363–387.

[5] Chris Lattner. *The Architecture of Open Source Applications: LLVM*. 2015. URL: http://aosabook.org/en/llvm.html (visited on 03/19/2018).

[6] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, p. 75. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

[7] Florian Niederhuber. "Change-Region Detection in LLVM." MA thesis. Universität Passau, Feb. 2018.

[8] Diego Novillo. "Memory SSA - A Unified Approach for Sparsely Representing Memory Operations." In: *Proc of the GCC Developers' Summit*. 2007.

[9] Florian Sattler. "A variability-aware feature-region analyzer in LLVM." MA thesis. Universität Passau, Mar. 2017.

DECLARATION

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

Passau, Germany, March 28, 2018

Jakob Schwarzweller