Bachelor's Thesis

IMPROVING VARA'S FEATURE REGION DETECTION BY USING AN INTERPROCEDURAL CONTEXT-SENSITIVE TAINT ANALYSIS

LAURITZ HENRIK TIMM

November 23, 2021

Advisor:

Florian Sattler, M.Sc.	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Sebastian Hack	Compiler Design Lab

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ____________________________
(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Most modern software allows users to adapt the behaviour of a program during compilation or run-time. In our work, we focus on run-time configurability and refer to code that depends on configuration options as features. Using a certain set of features may greatly affect the performance of a program compared to a different selection / configuration.

Researchers are often interested in specific code regions with a certain criterion. VaRA is a tool, which was developed to analyse those regions. The current implementation of VaRA already provides us with a rudimentary detection, allowing us to identify which parts of the program are feature regions. However, the analysis used by the current detection algorithm is too imprecise, as it may not find all regions related to features, or (even worse) may attribute unrelated code to a feature.

The region detection in VaRA is based on a data-flow analysis, which allows us to annotate code with feature taints. Therefore, our results are highly dependent on the accuracy of the used analysis. We replace the current one with a more elaborate implementation from the static analysis framework PhASAR which utilizes the IDE algorithm. The new data-flow analysis improves the precision of our algorithm so that we can now detect feature regions more accurately.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ESG	exploded super-graph
FOSD	feature-oriented software development
IDE	interprocedural distributive environment
IFDS	interprocedural finite distributive subset
IR	intermediate representation
SPL	software product line
CFG	control-flow graph
SSA	static single assignment
CI	continuous integration

1 INTRODUCTION

Modern software is often large in scale and highly configurable. This allows users to create sets of configurations, which are each tailored to a specific task. A good example of this is LLVM and its Clang compiler, which provides a vast range of configurable options[1] for compiling C++ source files. We will look into details of this framework in Section 2.5 (*LLVM*), but as it is way too complex to provide a comprehensible example on configurability, we consider a more approachable task instead:

> Assume that we have implemented some software with messaging capability, for example a chat client (like Telegram[2]). To what extent will the speed of sending a message be affected, if we turn on encryption?

Our main idea is, that we want to find code, which is related to a specific feature. In this example, we want to detect all parts of our implementation, which are only executed, if we enable encryption. To achieve this, we need to trace the data flow from a parsed command-line argument, which enables the given feature, to some conditional statements. This way we may find consecutive lines of code, which are only executed, if we enable encryption. With those code regions, we can now precisely measure performance through instrumentation. For this thesis, we will only focus on the precision of the described analysis.

Our main tool for region detection is VaRA[3], while we will use together with PhASAR to solve data-flow problems, in particular to improve the propagation of feature taints. Both frameworks are based on LLVM and use the same intermediate representation (IR), allowing us to easily integrate PhASAR into our analysis pipeline.

1.1 GOAL OF THIS THESIS

We want to show, that if we replace the old data-flow analysis of VaRA with one from PhASAR, we improve the precision of our algorithm to detect feature-related code. To guide the comparison of the analyses, we formulate three research questions:

RQ1 Can we improve the detection accuracy of the rudimentary data-flow analysis used in VaRA with an external tool?

RQ2 Can our new analysis correctly model common feature and implementation patterns used in C/C++?

RQ3 To what extent does our new analysis support different static analysis properties, such as flow-, context-, or field-sensitivity?

[1] https://clang.llvm.org/docs/ClangCommandLineReference.html (visited on 21/11/2021)
[2] https://telegram.org/ (visited on 21/11/2021)
[3] https://github.com/se-sic/VaRA-Tool-Suite (visited on 21/11/2021)

2 BACKGROUND

In this chapter, we introduce the theoretical foundations of our analysis. First, we establish specific terms and definitions. We start with describing feature models and their components, continue on how we analyse data flow using taint propagation with PhASAR, and look into details of the LLVM framework. In the last section, we will see how we can use VaRA to bring everything together to detect feature regions.

2.1 FEATURE-ORIENTED SOFTWARE DEVELOPMENT

As software gets more complex, the number of configuration options for large projects increases. To handle requirements and configurability on those systems, Apel et al. [3] introduce an encapsulating concept:

> Feature-oriented software development is a paradigm for the construction, customization, and synthesis of large-scale software systems.

The concept of feature-oriented software development (FOSD) provides a way to tailor software to the needs of the user and the possible application. Therefore, a software product line (SPL) consists of different software systems generated from a common set of features.

2.1.1 *Features and Configurations*

A software system may offer a set of features, which makes it configurable. Those configuration options may be selectable either during compilation or at run-time.

The general concept of a feature is driven by the distinction of problem and solution space [6]. The set of all valid system specifications in a domain (for example all valid feature combinations) is referred to as the problem space, and the set of all concrete systems in the domain is called the solution space. A goal of domain engineering is to automate the mapping as seen in Figure 2.1 between specifications and concrete implementations.

The term feature was first introduced as an abstract term to describe aspects, quality, or characteristics of software systems, which are visible to the user [8]. Instead of this very broad definition, we again refer to Apel et al. [3], who define a feature in a more technical way as a structural modification of a given program in order to satisfy a user requirement, or to implement a design decision, while offering a configuration option.

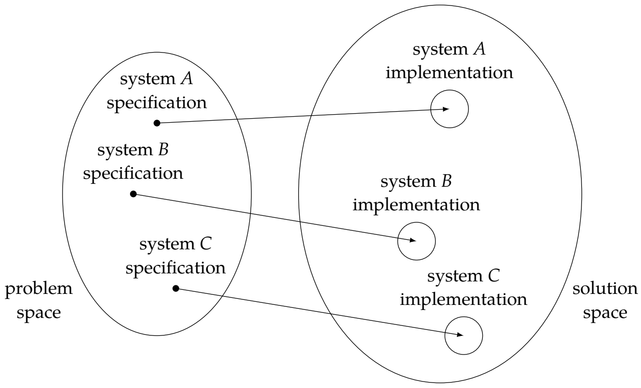


Figure 2.1: Problem and solution space, reproduced from Czarnecki, Østerbye, and Völter [6].

2.1.2 *Feature Models*

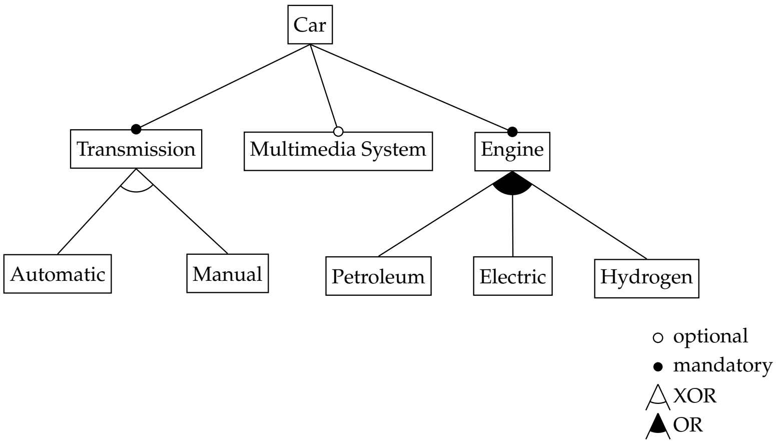


Figure 2.2: Example of a feature diagram, showing exemplary configuration options of a car.

Kang et al. [8] introduce feature models to describe an entire SPL:

> A feature model represents the standard features of a family of systems in the domain and relationships between them.

A feature diagram is a human-readable way to represent all features of a model as a graph. We see in Figure 2.2, that this visualization is easily comprehensible, while containing most relevant information about feature relations. For this example we require either an automatic or manual transmission for the car, represented as an alternative group. The engine can be petroleum, electric or hydrogen based, or any combination of these, expressed with a disjunction. A multimedia system is just optional.

This graph must not be empty, as we always require some kind of root feature representing the part of the system, which is independent of any feature. We distinguish relationships as optional or mandatory. If the relation between a feature and a subfeature is mandatory, the child is required to be present in any configuration containing the parent [7]. On the other hand, if the relationship is optional, the subfeature is not required to be present. In addition, we introduce two groupings, or-groups where any subset of subfeatures may be selected, but the selection must not be empty, and alternative-groups, of which only one child is allowed to be selected at any time [8].

In addition to these parental relationships, Kang et al. [8] introduce composition rules, which we can draw through cross-tree constraints. For the sake of simplicity, we do not go into detail on this topic, but they can be used with the VaRA feature library[1].

We will see, that our analysis does not require much information about features except from their name and occurrences as feature variables in source code.

2.2 CONTROL-FLOW GRAPH

A control-flow graph (CFG) (or simply "flow graph" according to Aho, Sethi, and Ullman [1]) is a way of representing execution paths of a program.

```cpp
int main() {
  int x = 0;
  int y = 42;

  while (x < 10) {
    x = x + 1;
    y = y + x;
  }

  return y;
}
```

Listing 2.1: Simple C++ program

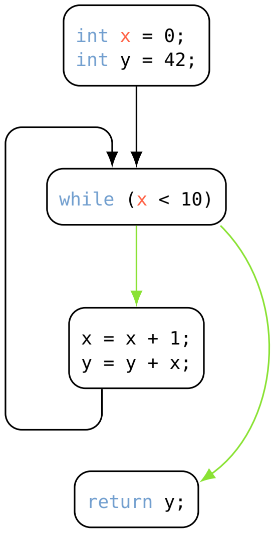


Figure 2.3: Example of a CFG with a simple loop.

[1] https://github.com/se-sic/vara-feature (visited on 21/11/2021)

Figure 2.3 is the CFG of the C++ program in Listing 2.1. We decided to use a program with a while-loop, as this shows how the control flow of a loop is represented in the graph. If we use an if-statement instead, the program becomes sequential.

In addition, we highlighted the initialization and first use of the variable x in the condition of the while-loop. This shows, that whether we continue looping is dependent on the value of x. Green edges indicate possible branches, as we consider if x is less than 10.

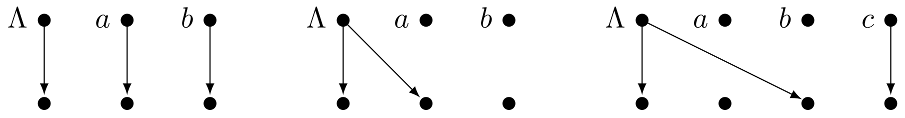
2.3 DATA-FLOW ANALYSIS

A data-flow analysis aims at solving the question, where variables are used in a program and how their values are propagated. Therefore, we need a way to detect how the values of variables changes throughout a program. This can be achieved by tracing flow facts for a selected set of variables [12].

The interprocedural finite distributive subset (IFDS) framework [14] defines a general way to solve interprocedural, flow-sensitive and context-sensitive analyses of subset problems [4]. With this algorithm, we try to reduce any program-analysis problem to a graph-reachability problem.

Let $f : 2^D \rightarrow 2^D$ be an arbitrary distributive function on some domain of variables D. We define a set $S \subseteq \{\Lambda\} \cup D$ and represent f as a graph.

Representations of exemplary functions are shown in Figure 2.4. The identity $\Lambda \mapsto \Lambda$ is always present, as Λ represents an empty set of live variables. An edge $\Lambda \mapsto d$ means $d \in f(S)$ for all S, and an edge $d_1 \mapsto d_2$ means $d_2 \in f(S)$ if $d_1 \in S$, therefore $d_2 \notin f(\emptyset)$.



(a) $\lambda S.S$ (b) $\lambda S.\{a\}$ (c) $\lambda S.(S - \{a\}) \cup \{b\}$

Figure 2.4: Function representation in IFDS, reproduced from Reps, Horwitz, and Sagiv [14]. (a) is the identity function. (b) generates a and kills b. (c) kills a, generates b, and leaves c untouched.

An interprocedural distributive environment (IDE) is very similar to an IFDS, but we may additionally annotate edges with micro-functions for each variable. Instead of only checking whether a node is reachable, the IDE algorithm can propagate information along a path.

To solve an arbitrary data-flow problem with a preselected set of variables, we taint these variable and assign a unique identifier to each declarative instruction. If a tainted variable v is used in the program, we taint this instruction as well. We can propagate the taint throughout the program for usages of v, meaning, we find data flow from the initialization of v.

We will see with PhASAR that we can encode any interprocedural CFG into an exploded super-graph (ESG) (see the example in Figure 2.5) and use the IDE algorithm to propagate taints. A node s in the ESG is reachable from a start node s_0, if it is in the set of live variables at this point [4].

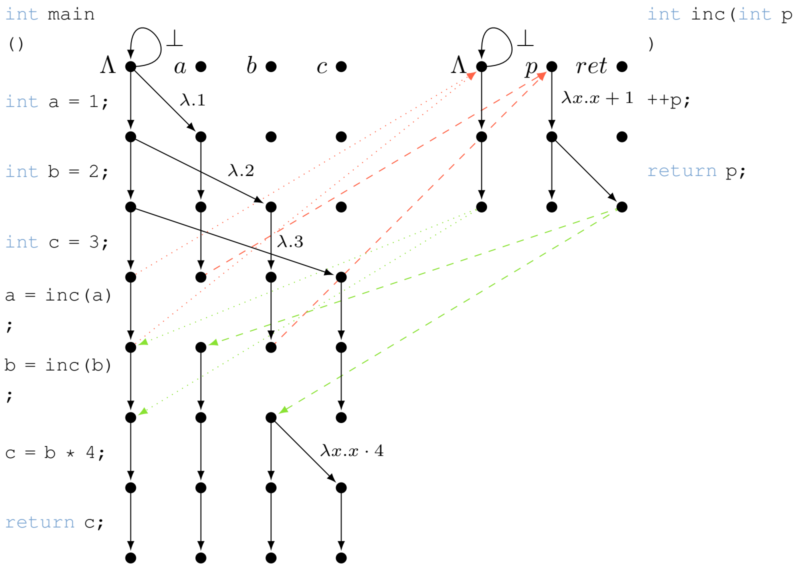


Figure 2.5: Example of an ESG with a main method and a simple sub-procedure, reproduced from Schubert, Hermann, and Bodden [16]. Black Edges indicate intraprocedural data-flow through transition functions, colored edges show interprocedural dataflow.

2.4 PHASAR

PHASAR[2] is a static analysis framework developed at Paderborn University, which focuses on solving data-flow problems with LLVM. First, we construct an ESG from the interprocedural CFG with the IDE algorithm. Any analysis on this graph is fully context-sensitive by construction. An example of such a graph is Figure 2.5, where we see two procedures connected by edges, which visualize the interprocedural data flow. With this representation, we are able to encode an arbitrary data-flow problem as a graph-reachability problem with edge annotations [16], and solve it as we build the ESG.

As we will see with our evaluation, the analysis is currently not field-sensitive, but this property is already in development and may be added in the future.

Additionally PHASAR supports an alias analysis, which detects pointers / addresses of the same object in memory and track them accordingly.

2.5 LLVM

LLVM is an umbrella term used to describe a framework, which provides tools and projects related to compiler infrastructure and code optimization / generation [11]. In Section 2.5.3 (*LLVM Intermediate Representation*)), we will look into details of the LLVM-IR, which enables the framework to work independent of any specific programming language.

[2] https://phasar.org (visited on 20/11/2021)

2.5.1 *Compiler*

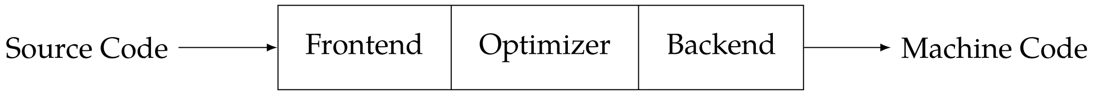


Figure 2.6: Components of a three-phase compiler [10].

In abstract terms, a compiler works as a translator between source code written in a high level programming language and a low level machine language [1]. As we drew out in Figure 2.6, LLVM uses a three-way compiler to process arbitrary source code, for example C++, into machine code, for example AMD64 [10]. LLVM is not limited on these languages, but supports a variety of languages as source or target language. The compilation is divided into processing a source with the frontend, optimization and the code generation with the backend. For the internal steps, LLVM utilises its own IR, which we will introduce in Section 2.5.3 (*LLVM Intermediate Representation*).

In general, we distinguish single-pass and multi-pass compilation [1]. While a single-pass compiler runs over the source code only once and directly outputs machine code, multi-pass compilation produces intermediate results, which can be transformed before generating output. We use the multi-pass design of LLVM to run our analysis in the optimization step on the intermediate results generated with the frontend.

The C++ compiler of LLVM is CLang, and the optimizer, which works independent of the source language, is called Opt.

2.5.2 *Pass Framework*

We want to use the optimization step of LLVM to run our analysis on its IR.

LLVM provides two pass interfaces for either modules of function. A module pass is run over a whole translation unit and may access all members of the module, function passes process each function individually. We can use a pass to extract and insert information into the IR.

Only after running arbitrary passes, the result is finalized into an executable. While CLang can produce LLVM-IR when invoked with the arguments `-S -emit-llvm`, we can call Opt in succession to run our pass over the intermediate result from the first compilation step.

2.5.3 *LLVM Intermediate Representation*

Our framework and tests are written in C++, but performing the actual analysis on any higher language can be hard because of the complex structure, syntax and type system. Instead, we use an assembly like representation, which limits the number of instructions and syntax as well as provides a more linear program structure.

LLVM-IR (also often referred to as LLVM assembly language) is a human-readable low-level language, which is the common intermediate representation throughout all phases of the LLVM toolchain [11].

2.5.3.1 *Static Single Assignment*

To help with flow insensitive analyses, LLVM uses static single assignment (SSA) with its intermediate representation. In this form, each variable has only a single program point, at which it occurs at the left-hand side of an assignment. To construct SSA from a given program, we need to make variable definitions unique and insert ϕ-functions to resolve conflicts at control-flow joins. The transformed program has the same semantics as the original code. As shown in Figure 2.7, we can efficiently compute node dominance with the CFG of a program and deduce joins where we reach a node by two different definitions of a (non-SSA) variable [5]. An fast way to construct unique identifiers is by adding indices to the original variable name.

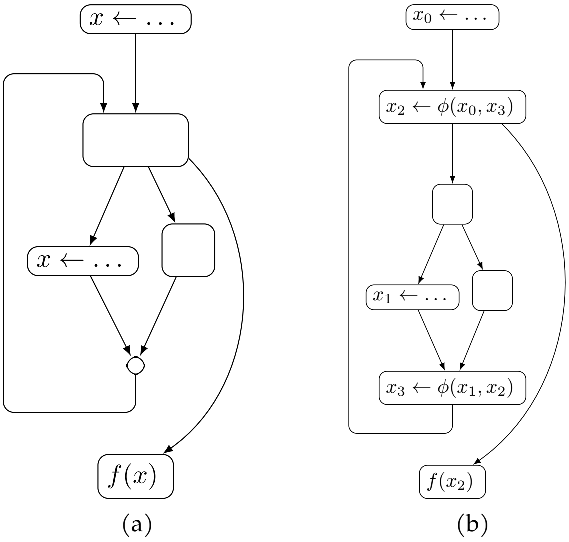


Figure 2.7: Example for global value numbering for a simplified CFG. (a) represents a program with a simple loop, conditional branching and a variable x. (b) is the same program in SSA form with ϕ-functions to select proper definitions for x at joins.

2.5.3.2 *Basic Blocks*

Most assembly languages only allow rudimentary control-flow manipulation with jumps to predefined memory locations.

LLVM-IR groups all statements into blocks[3] labeled with an unique identifier. Each block can be executed by either jumping to its label or continuing from a previous block (as the program is stored sequentially in memory).

2.5.3.3 *Functions*

Like most common languages LLVM-IR provides callable functions at a global scope. They consist of an unique identifier, which may be a mangled name[4] and a list of parameters. Functions can be called similar to higher languages and change the control flow to a corresponding implementation.

[3] https://llvm.org/docs/ProgrammersManual.html#the-basicblock-class (visited on 22/11/2021)
[4] https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling (visited on 22/11/2021)

2.5.3.4 *Global Variables*

In addition to functions, LLVM is able to handle globally defined variables[5]. Global variables use memory, which is allocated at compile time. Usage of these variables is similar to any identifier in LLVM-IR and will be shown in the next section.

2.5.3.5 *Common Instructions*

In the following we introduce some common syntactical constructs of LLVM-IR. Note that this is only a small excerpt of the language, more detailed specifications of all instructions can be found in the *LLVM Language Reference Manual* [13].

ALLOCA [6]

This instruction allocates memory for a variable on the stack frame with a given type, which determines the allocations size.

```
<result> = alloca <type>
```

Example:

```
%1 = alloca i32
```

Allocate space for a 32-bit integer on the stack, which can be referred to as `%1`.

LOAD [7]

We read from memory and assign this value to new variable. This can be used, for example, to load a globally defined variable, or the value stored at `%1`, from our previous example.

```
<result> = load <ty>, <ty>* <pointer>
```

Example:

```
%2 = load i8, i8* @Foo
```

Load the 8-bit integer stored in the global variable `@Foo` into `%2`.

STORE [8]

Write value to memory of given variable.

```
store <ty> <value>, <ty>* <pointer>
```

Example:

```
store i8 %3, i8* %4
```

Store the 8-bit integer of `%3` in `%4`.

[5] `https://llvm.org/docs/LangRef.html#global-variables` (visited on 22/11/2021)
[6] `https://llvm.org/docs/LangRef.html#alloca-instruction` (visited on 21/11/2021)
[7] `https://llvm.org/docs/LangRef.html#load-instruction` (visited on 22/11/2021)
[8] `https://llvm.org/docs/LangRef.html#store-instruction` (visited on 21/11/2021)

BR [9]

To provide basic control-flow manipulation, this instruction transfers control flow to another basic block based on a condition. The semantic is very similar to a `if-else`-statement in C++.

```
br i1 <cond>, label <iftrue>, label <iffalse>
```

Example:

```
br i1 %5, label %6, label %7
```

Continue with either block `%6` or `%7` dependend on the boolean value of `%5`,

ICMP [10]

Also very important are integer comparisons, which are evaluated in regard to the specifier `<cond>`.

```
<result> = icmp <cond> <ty> <op1>, <op2>
```

Example:

```
%9 = icmp slt i32 %8, 10
```

Determine if the value of `%8` is signed less than (`slt`) 10 and, respectively, set `%9` to either true or false.

ADD [11]

LLVM-IR allows us to perform arithmetic calculation, in this case the addition of two integers. With the keyword `nsw` ("No Signed Wrap"), the result value is undefined if signed overflow occurs.

```
<result> = add nsw <ty> <op1>, <op2>
```

Example:

```
%11 = add nsw i32 %10, 1
```

Increment the value of `%10` by the constant `1` and store the result in `%11`.

RET [12]

If we reach the end of a method, we always return the control from the current position back to the caller. Similar to C++, we may additionally return a value instead of `void`.

```
ret <type> <value>
ret void
```

Example:

```
ret i32 %12
```

Return the integer stored in `%12` as return value to the caller.

[9] https://llvm.org/docs/LangRef.html#br-instruction (visited on 22/11/2021)
[10] https://llvm.org/docs/LangRef.html#icmp-instruction (visited on 22/11/2021)
[11] https://llvm.org/docs/LangRef.html#add-instruction (visited on 22/11/2021)
[12] https://llvm.org/docs/LangRef.html#ret-instruction (visited on 22/11/2021)

2.5.3.6 *Metadata*

LLVM-IR provides us with an extensible metadata format, which allows to attach additional information to instructions. Those metadata nodes are very useful to store intermediate results or pass information between different compile / optimization steps. Listing 2.2 shows a global variable `Foo`. The notation of metadata always starts with an exclamation mark. The instruction has for one a named metadata node `!FVar`, and a second references `!0`, which is a metadata declaration at the end of the program. The later approach allows reuse of metadata nodes and saves space. In this example, this resolves to the name of the feature `"Foo"`.

```
@Foo = dso_local global i8 1, align 1, !FVar !0

!0 = !{!”Foo”}
```

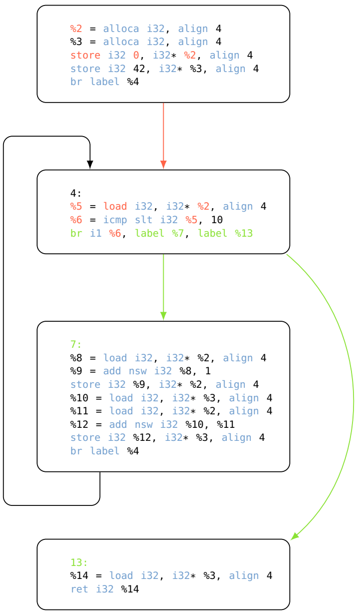
Listing 2.2: Metadata.

In the LLVM library, metadata nodes of an instruction are accessible as `MDNode` with an `MDString` holding the name of the node, allowing us to easily parse the contained data. As we have already seen through our example, metadata nodes allows us to attach information about features, but we will later use them to also store intermediate analysis results.

2.5.3.7 *Control Flow in LLVM*

In Section 2.2, we have already seen a CFG for the simple C++-program from Listing 3.1. The same way we may construct a flow graph from the corresponding LLVM-IR basic blocks. We see in Figure 2.8 that the resulting graph is isomorphic to the one seen in Figure 2.3.

Again, we highlighted the variable `%2`. Similar to the data flow in Figure 2.3, branching to either `%7` or `%13` depends on the current value of `%2`.



```
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 0, i32* %2, align 4
store i32 42, i32* %3, align 4
br label %4
```

```
4:
%5 = load i32, i32* %2, align 4
%6 = icmp slt i32 %5, 10
br i1 %6, label %7, label %13
```

```
7:
%8 = load i32, i32* %2, align 4
%9 = add nsw i32 %8, 1
store i32 %9, i32* %2, align 4
%10 = load i32, i32* %3, align 4
%11 = load i32, i32* %2, align 4
%12 = add nsw i32 %10, %11
store i32 %12, i32* %3, align 4
br label %4
```

```
13:
%14 = load i32, i32* %3, align 4
ret i32 %14
```

Figure 2.8: Example of a CFG with a simple loop.

2.6 VARA

VaRA is a variablity aware region analyzer and part of the VaRA Tool Suite[13] developed at Saarland University. This tool provides a framework based on LLVM to detect code regions of several kinds, and to analyse interactions between them [15].

2.6.1 *Regions*

Our approach is based on detecting code regions, meaning consecutive statements, which share a common analysis property.

For our analysis, we need two different region concepts. Feature regions are basically if-regions with a condition, which has data-flow from a prior defined feature variable. We will see in Chapter 4 how we can combine feature variables and if-regions into a feature region.

IF-REGIONS If-Regions are an important intermediate step for our analysis. VaRA already provides this analysis[15] and uses metadata nodes to identify instructions as either conditions or branch regions.

If we look at the excerpt in Listing 2.3, we see the instructions annotated with `!IfRegion` and an unique identifier, which contains information about whether the region is a condition or branch. Those regions are identified by a tuple of either `C` for condition, `T` for then-branch or `E` for else-branch, and the number of the if-region.

We refer to regions of both branching options as branch-regions.

```
entry:

  ⋮

  %tobool = ..., !IfRegion !3
  br i1 %tobool, label %if.then, label %if.else, !IfRegion !3

if.then:
  ..., !IfRegion !4

if.else:
  ..., !IfRegion !5

⋮

!3 = !{!”[C,1]”}
!4 = !{!”[T,1]”}
!5 = !{!”[E,1]”}
```

Listing 2.3: If-region.

[13] https://github.com/se-sic/VaRA-Tool-Suite (visited on 22/11/2021)

FEATURE-REGIONS Feature regions are part of a program, which is the implementation of a certain configuration option.

In Listing 2.4, we see a common way to enable a feature, in this case `Foo`, with a feature variable. We assign some value calculated from, for example, the command line arguments to the variable `Foo`. Dependent on the value of `Foo` only one of the branches is executed. Both branches together form the feature region of `Foo`.

```
bool Foo = ...
if (Foo) {
  ...
} else {
  ...
}
```

Listing 2.4: Feature region.

3 DETECTING FEATURE REGIONS

So far we only covered the fundamental principles of VaRA, PhASAR and our analysis, but now we now want to go into detail how to detect feature regions in code.

3.1 PIPELINE OVERVIEW

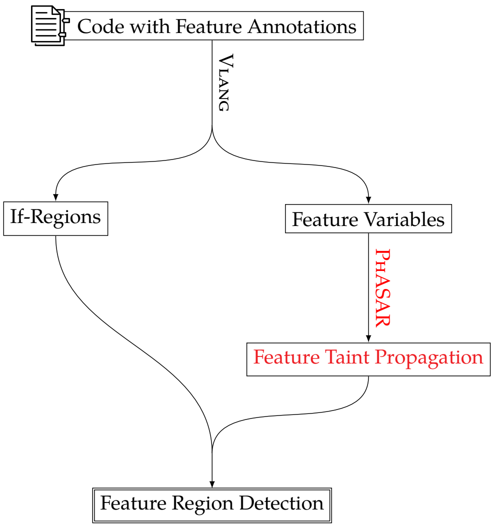


Figure 3.1: Pipeline to detect feature regions.

Figure 3.1 shows how our feature detection is divided into multiple steps.

First, VLang, a modified version of Clang that utilizes VaRA, detects feature variables and if-regions. We annotate the resulting IR with this information as metadata.

With this LLVM-IR and metadata, we now apply our improved data-flow analysis, which in the background uses PhASAR to propagate taints of feature variables throughout the program. Finally, we can bring everything together and map features to conditional regions, so we identify the corresponding branch regions as feature regions, whose execution therefore depends on the value of given feature variable.

VaRA already provides VLang, so we only had to implement the feature taint propagation with PhASAR (highlighted in red), and a second pass to construct feature regions.

Section 3.3 (*Taint Analysis in Detail*) will show in detail how we annotate feature variables and how the metadata is represented in LLVM-IR.

3.2 IMPLEMENTATION

In this section we will give a short summary of how we implemented our analysis. This part is split into initializing and using the PHASAR analysis with feature taints / variables, and detecting / extracting feature regions.

3.2.1 *Analysis Pass*

We implement the class `PhasarTaintAnalysis` from the PHASAR library in `PhasarFTA` as a module pass.

First, we need to initialize feature variables in VARA's singleton `FeatureManager`. On one hand we call `initializeGlobalFeatures` on the module to initialize globally defined feature variables, on the other hand `initializeFunctionFeatures` for each function.

Next, we need to provide an `EdgeFactGenerator`, which computes for either a `GlobalVariable` or an `Instruction` the corresponding taints.

We need to add some special handling to look up `!FVar` metadata.

If we handle a global variable, we can simply look for metadata with `!FVar` For `load` and `alloca` instruction, we process metadata of the operand instead.

With the extracted metadata, we can now look up any corresponding feature by name and construct a corresponding feature taint. We collect all taints for possible features and return this set.

The pass `PhasarFeatureTestAnalysis` runs the analysis on a module and prints taints for each instruction of the LLVM-IR. `PhasarFTAWrapperPass` runs only the analysis and can be used in other analyses.

3.2.2 *Extraction Pass*

We implement a module pass to extract feature regions from our analysis. As described in our general overview of the detection pipeline in Figure 3.1, we first need to run both the `PhasarFTAWrapperPass` and VARA's `IfRegionDetection`.

We take the analysis data from the data-flow analysis and iterate over the function list of the module, and get analysis data from the `IfRegionDetection` for ech function.

For each if-region, we combine features of detected taints with condition and branch regions to create feature regions. We store those results in the VARA's `IRegionStore` allowing us to either print all regions in a human-readable way or use the data in other analyses.

3.3 TAINT ANALYSIS IN DETAIL

Up to now, we only covered, on a high level, how our analysis pipeline works. Next, we depict in detail how the specific analysis steps work and demonstrate each step on the example in Listing 3.1. This very simple C++ program has only a single globally defined feature variable.

```
bool Foo __attribute__((feature_variable(”Foo”))) = true;

int main() {
  if (Foo) {
    return 42;
  } else {
    return 0;
  }
}
```

Listing 3.1: Example C++ program with a feature variable.

We notice, that dependent on the value of `Foo`, we execute one of the `return` statements. Therefore, we expect to detect those statements as a feature regions of `Foo` with our analysis.

The following steps work on the LLVM intermediate representation of given program, therefore we compile it with VLANG to the following code while already inserting information about If-Regions.

```
@Foo = dso_local global i8 1, align 1, !FVar !0

define dso_local i32 @main() {
  entry:
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
    %0 = load i8, i8* @Foo, align 1, !IfRegion !3
    %tobool = trunc i8 %0 to i1, !IfRegion !3
    br i1 %tobool, label %if.then, label %if.else, !IfRegion !3

  if.then:
    store i32 42, i32* %retval, align 4, !IfRegion !4
    br label %return, !IfRegion !4

  if.else:
    store i32 0, i32* %retval, align 4, !IfRegion !5
    br label %return, !IfRegion !5

  return:
    %1 = load i32, i32* %retval, align 4
    ret i32 %1
}

!0 = !{!”Foo”}

!3 = !{!”[C,1]”}
!4 = !{!”[T,1]”}
!5 = !{!”[E,1]”}
```

Listing 3.2: Example with if-regions.

We notice, that our feature variable is now present as a global allocation, whereas the name of the feature itself becomes a metadata node `!FVar` (highlighted in red) which value is present in the symbol table. At the same time, VLANG adds `!IfRegion` metadata to certain statements, annotated with either condition (`C`), then-branch (`T`) or else-branch (`E`), and a region identifier. Those annotations can be parsed and stored easily as if-regions.

We now have LLVM-IR with a lot of annotations. In the next step, we can run our analysis (including PHASAR) pass over this code and use the present information. This step is shown in the following example.

In Listing 3.3, we initialize PHASAR to taint the data flow of the feature variable `Foo`. We highlighted the usage of `Foo` and propagated taints in red. The conditional part of the if-region is colored in blue and both brach-region also red.

```
@Foo = dso_local global i8 1, align 1, !FVar !0

main
  entry
    %retval = alloca i32, align 4, !psr.id !4 FTaints: {}
    store i32 0, i32* %retval, align 4, !psr.id !5 FTaints: {}
    %0 = load i8, i8* @Foo, align 1, !IfRegion !6, !psr.id !7 FTaints: {Foo}
    %tobool = trunc i8 %0 to i1, !IfRegion !6, !psr.id !8 FTaints: {Foo}
    br i1 %tobool, label %if.then, label %if.else, !IfRegion !6, !psr.id !9 FTaints: {Foo}
  if.then
    store i32 42, i32* %retval, align 4, !IfRegion !10, !psr.id !11 FTaints: {}
    br label %return, !IfRegion !10, !psr.id !12 FTaints: {}
  if.else
    store i32 0, i32* %retval, align 4, !IfRegion !13, !psr.id !14 FTaints: {}
    br label %return, !IfRegion !13, !psr.id !15 FTaints: {}
  return
    %1 = load i32, i32* %retval, align 4, !psr.id !16 FTaints: {}
    ret i32 %1, !psr.id !17 FTaints: {}
```

Listing 3.3: Example with taint propagation.

With all required information now present, we are able to construct a feature region for `Foo`. The region consists of both branch regions (colored in red), as the condition of the if-region was tainted with `Foo`. Following up, we can now use our results in further analyses, for example, we can add instrumentation points for both branch regions and measure feature performance precisely.

4 MEASURING ANALYSIS ACCURACY

We need to construct a set of test cases, to determine the properties of our analysis based on this test suite. This allows us to answer our research question from Section 1.1 (*Goal of this Thesis*) and measure the precision of our analysis. To achieve a good coverage of common C++ features, we based our test suite on the work of Labrenz [9], who already implemented a similar analysis.

4.1 ASSERTING TAINTS

How do we practically apply those analyses described in previous overview given a C++ file?

Initially, we need to compile source code to LLVM-IR, which already injects meta information about if-regions and feature variables. To enable this, VLANG provides the flag `fvara-IFA`. Afterwards, we can apply the PHASAR based data-flow analysis by calling OPT on the compiled file with `vara-PFA`, which runs our pass over the LLVM-IR and prints results to `stdout`.

The detection results can now be extracted, but for the sake of a detailed look into the actual taint propagation, we only look at the resulting LLVM-IR and propagated taints.

```
$ clang -fvara-IFA -S -emit-llvm <filename>.cpp -o <filename>.ll
$ opt -vara-PFA -S <filename>.ll}
```

Listing 4.1: Invocations.

All tests use a set of feature variables and are injected with the expected output (a ground truth, which we build by hand) so that LLVM's file checker (FILECHECK) could be used to determine whether the output of our analysis is correct. Most assertions use regular expressions so that they may work independent of concrete values (for example region IDs or labels). These expressions result from the LLVM-IR a perfect analysis would produce.

```
// RUN: %clang -fvara-IFA -S -emit-llvm %s -o %t
// RUN: %opt -vara-PFA -S %t -o /dev/null | FileCheck %s

bool Foo __attribute__((feature_variable("Foo"))) = true;

int main() {
  if (Foo) {
    // CHECK: [[L0:%.*]] = load i8, i8* @Foo, align 1, !IfRegion [[RID:!.*]], !psr.id !{{.*}} FTaints: {Foo}
    // CHECK: [[ToBool:%.*]] = trunc i8 [[L0]] to i1, !IfRegion [[RID]], !psr.id !{{.*}} FTaints: {Foo}
    // CHECK: br i1 [[ToBool]], label %{{.*}}, label %{{.*}}, !IfRegion [[RID]], !psr.id !{{.*}} FTaints: {Foo}
    return 42;
  } else {
    return 0;
  }
}
```

Listing 4.2: C++ program with assertions for a single FILECHECK pass.

We modularize different test cases by passes of LLVM FileCheck where each run asserts statements of a single function body, which is detected by the mangled function name standardized for C++ compilers. This allows us to neglect the order in which functions may appear in LLVM-IR. Listing 4.3 shows the content of a source file, which tests whether the faint of Foo is propagated through the function callee_foo.

```
// RUN: %clang -fvara-IFA -S -emit-llvm %s -o %t
// RUN: %opt -vara-PFA -S %t -o /dev/null > %t2
// RUN: cat %t2 | FileCheck %s --check-prefix=CHECK-0
// RUN: cat %t2 | FileCheck %s --check-prefix=CHECK-1

volatile int Foo __attribute__((feature_variable(”Foo”))) = 1;

void callee_foo(int Arg) {
  // CHECK-1: {{^}}_Z10callee_fooi{{$}}
  // CHECK-1: [[Arg:%.*]] = alloca i32, align 4, !psr.id !{{.*}} FTaints: {}
  volatile int A = 21;
  // CHECK-1: [[A:%.*]] = alloca i32, align 4, !psr.id !{{.*}} FTaints: {}
  A = Arg;
  // CHECK-1: [[L0:%.*]] = load i32, i32* [[Arg]], align 4, !psr.id !{{.*}} FTaints: {Foo}
  // CHECK-1: store volatile i32 [[L0]], i32* [[A]], align 4, !psr.id !{{.*}} FTaints: {Foo}
}

void caller() {
  // CHECK-0: {{^}}_Z6callerv{{$}}
  int A = 42;
  // CHECK-0: [[A:%.*]] = alloca i32, align 4, !psr.id !{{.*}} FTaints: {}
  // CHECK-0: store i32 42, i32* [[A]], align 4, !psr.id !{{.*}} FTaints: {}
  A = Foo;
  // CHECK-0: [[L0:%.*]] = load volatile i32, i32* @Foo, align 4, !psr.id !{{.*}} FTaints: {Foo}
  // CHECK-0: store i32 [[L0]], i32* [[A]], align 4, !psr.id !{{.*}} FTaints: {Foo}
  callee_foo(A);
  // CHECK-0: [[L1:%.*]] = load i32, i32* [[A]], align 4, !psr.id !{{.*}} FTaints: {Foo}
  // CHECK-0: call void @_Z10callee_fooi(i32 [[L1]]), !psr.id !{{.*}} FTaints: {}
}
```

Listing 4.3: C++ program with assertions for multiple FileCheck passes.

FileCheck is limited on checking, whether checks appear in the required order and may unintentionally match to arbitrary subsequent code. If statements do not appear in the expected way, we have to be careful with assertions. As we check failing test anyway from hand, this is a good way to filter out errors in our test suite.

4.2 TEST CASES

To test, whether our analysis is flow-, context-, or field-sensitivity, as we are wanted to show for RQ3, we construct specific test programs, which implement corresponding patterns.

We divided our test suite into five categories, where each category groups tests with similar properties of the analysis together. Most of the successful tests are already integrated into VaRA's development branch and part of the continuous integration (CI).

On the next pages, we describe the different test categories in detail and motivate why we need them in our evaluation.

CONTROL

Looking into control flow modification in C++ [17] we came up with different tests for commonly used syntactical structures [9].

IF GLOBAL

We use a globally defined feature variable in condition for if-else-statements.

IF LOCAL

In addition to global declarations, we also test whether we can use locally declared feature variables to enable / disable execution of feature-related code.

WHILE

In the next step we use a while-loop and expect correct taint propagation, if we use a tainted variable in either condition or body.

DO WHILE

For this test, we replace the prior loop with a do-while-loop.

FOR

Similar to the other tests of loops, this test consist of a simple for-loop.

SWITCH

We use feature variable as switch condition.

SWITCH CASES

This also test a switch statement, but uses different features in cases.

SWITCH FALLTHROUGH

We test, if a taint is propagated even if the control flow falls through cases to a statement below.

EXCEPTION

This text expects that even the use of feature variable in a catch block is tainted.

JUMP

A jump with a goto-statement to an arbitrary location should not mess up taint propagation.

UNREACHABLE

This test consists of an infinite loop with a conditional jump. In Listing 4.4, we expect that code after this if-statement is still tainted correctly, although it may not be reachable dependent on the value of `Foo`. It is important to initialize `Foo` in a non-static way, so that the compiler does not detect (or even remove) this code as dead.

```
Foo:
  if(Foo) {
    goto Foo;
  }
```

Listing 4.4: Infinite loop.

OPERATORS

These programs test, whether feature taints are present after operator usage.

ARITHMETIC

We test most of arithmetic operations in C++ with for all fundamental types. Listing 4.5 shows an small program, with which we can see whether a taint is correctly set and propagated. We expect that the integer a is initially untainted. After we add the feature variable Foo to a, we expect a single taint of this feature. Only with the following multiplication of the feature variable Bar, we expect two taints, respectively.

```
int A = 0;
A += Foo;
A *= Bar;
```

Listing 4.5: Arithmetic.

TERNARY

Instead of an if-else-statement, we use a ternary statement with a conditional operator, and a feature variable as operand.

SHIFT

In addition to previously tested operators, we explicitly test, whether taints are present after we bitshift a variable either left or right.

BITMASK

Listing 4.6 show, how we can use a bitmask on a feature variable to extract a specific bit. The result of this operation should still be tainted.

```
int Foo = 0b0100;
(Foo & 0b0100) >> 2;
```

Listing 4.6: Use of bitmask.

CAST

This test only casts a boolean feature variable to an integer, and we expect, that the result is still tainted.

INDEX

Similar to Listing 4.7, we set only a specific index of an array with a tainted feature variable. This test is sorted in this category, as the array is accessed via pointer arithmetic. We expect that not the whole array is tainted, but the taint only is propagated, if we use the specific index. With this test, we can assert whether our analysis is field sensitive, in which case it would be able to handle the tainted field independently.

```
bool A[256];
*(A + 42) = Foo;
```

Listing 4.7: Testing field-sensitivity.

DATA STRUCTURES AND POINTERS

We test, if our new analysis is field-sensitive.

CONST

In this test, we declare a global and a local feature variable constant, and check whether the usage of those is tainted correctly.

ARRAY

We store value of feature variable in an array. This test is similar to INDEX, but we use square brackets to access the array at a specific index.

UNION

We use a union of `bool` and `int`, initialize the active member with a tainted variable, and check whether the taint is propagation correctly. Afterwards we set the other member of the union to an untainted value and expect, that no taint is present if we use this member now.

UNION STRUCT

Instead of a basic type, we now initialize a union with a struct, which itself has a tainted field. We test, if accessing this field is tainted, but also if using any other field is not.

STRUCT

To test taint propagation with a struct, we use an object with two fields. Only first field is initialized with a tainted variable. As we already mentioned, we would expect, that the taint is only propagated, if we use the first member.

STRUCT BASE

This test checks, whether taint propagation works with on an inherited member of a struct.

POINTER

Instead of copying the value of a feature variable, we only use a pointer to the address of this global variable. As the analysis should taint the identifier of the pointer, any dereference should be tainted.

POINTER GOLF

For a more elaborate approach, we test different level of pointers to a feature variable and check whether tainting works for dereferencing as well.

POINTER STRUCT

We test, whether working with the pointer to a struct holds the same results as working without indirection.

FUNCTION POINTER

Depending on a configuration option, we may set a function pointer to a method, which executes code of that feature. We see in Listing 4.8 how we can use an if-else-statement to switch between a default implementation and code, which uses `Foo` and returns a tainted variable. We expect, that the return-statement is tainted, as the taint should have been propagated to the return value.

```
bool getFoo() { return Foo; }
bool getFalse() { return false; }

int main() {
  bool (*Func)(int);

  if (...) {
    Func = getFoo;
  } else {
    Func = getFalse;
  }

  return Func();
}
```

Listing 4.8: Model feature with function pointer.

FUNCTION VOID

Similar to the previous test, but we use a void-pointer instead and only cast to a function pointer when we need to call the method called. We expect, that the taint propagation works regardless of whether we use a function pointer or void-pointer.

FUNCTION VIRTUAL

For this test we build an object structure with inheritance similar to Listing 4.9, where only one of the objects uses a feature variable, but both implement the virtual method `getFoo`. We call `getFoo` on either `Enabled` or `Disabled`, and expect taint propagation only for the first struct, as it actually returns a tainted variable.

```
struct Base {
  virtual bool getFoo();
};

struct Enabled : Base {
  bool getFoo() override {
    return Foo;
  }
};

struct Disabled : Base {
  bool getFoo() override { return false; }
};
```

Listing 4.9: Virtual function.

FUNCTIONS

To assert interaction of multiple functions, this test suite provides tests, which work on multiple methods. We look at how taints are propagated through function calls, arguments and return values. As we here test interprocedural properties, we can also test whether our analysis is context-sensitive.

call
For this test, we call methods with a single tainted parameter and expect, that the taint is correctly propagated.

call parameter
We test, whether the analysis still works, if we pass multiple tainted arguments to a method.

return
For this test, we simply call a method and return a tainted variable. We expect, that the function call gets tainted, as the taint should get propagated through the returned value.

call chain
In addition to the previous tests, now the function themselves calls other methods forwarding potentially tainted variables. If at some point a return statement is tainted, we expect that all enclosing calls get tainted as well.

struct parameter
This test taints the field of a struct and passes it as argument to a method. The function should then return the tainted field, and therefore we expect to see the initial function call tainted as well.

field parameter
We initialize a field with a tainted variable, but then only passes this variable as copy to a method. The taint should be propagated to any use of this parameter in the method body.

struct return
Instead of using a whole struct as argument, we now call a method, which returns a struct with a tainted field. We expect from a field-sensitive analysis, that only if we access the field this usage is tainted. If our analysis has no such property, we would instead see a taint of the whole struct and consequently the function call.

struct nested
We initialize the field of a struct, which itself is nested in a second struct, with a feature variable. Like in Listing 4.10, we pass an instance of the enclosing struct as argument to a method. When the function uses that parameter, this should be tainted.

```
A.B.Foo = Foo;
foo(&A.B);
```

Listing 4.10: Nested struct as argument.

variadic
If we can pass a tainted variable as argument to a variadic function. Accessing `va_arg` should be tainted.

variadic forwarding
In addition to the previous test, the variadic function now forwards all its parameter to a second method. The taint should be propagated correctly through `va_list`.

TEMPLATES

In addition to the before mentioned more or less straightforward programs (in a sense, that the LLVM-IR is kind of structural similar to the source code), this tests use templates and therefore more elaborate syntax and control flow, than previous tests.

TEMPLATE

This test utilizes a simple template, similar to Listing 4.11, and checks, whether the taint of `Foo` is propagated if we use the parameterized typed member `Value` of this object.

```
template <typename T>
struct Template {
  Template() : Value(Foo){};

  T Value;
};
```

Listing 4.11: Basic template.

TEMPLATE CONST

For this test we use the feature variable a type parameter for the template. Note, that the program Listing 4.12 only compiles, if `Foo` is constant, therefore this test is dependent on the success of CONST .

```
template <int Value>
struct Template {
  int Tainted = Value;
};

⋮

Template<Foo>().Tainted;
```

Listing 4.12: Template with type prameter.

TEMPLATE POINTER

We see in Listing 4.13, how we can remove the dependency on CONST for the previous test by using a pointer to the feature variable instead.

```
template <bool *Value>s
struct Template {
  bool *Tainted = Value;
};

⋮

Template<&Foo>();
```

Listing 4.13: Template with pointer type.

TEMPLATE METHOD
The approach of the previous test can be applied not only to a generic struct, but to a generic method as well.

TEMPLATE BASE
In addition to the previous two tests, we add inheritance from `Template` in Listing 4.12, and check whether the taint of `Foo` is propagated correctly.

4.3 COMPARISON

Our goal is to evaluate whether the PHASAR based dataflow analysis is an improvement compared with the original analysis. For this, we needed a baseline to show the capability of our analysis, therefore we run all tests and evaluated whether features are detected and taints correctly propagated throughout given programs. The same way, we then run the original analysis on the same tests and table whether we now detect prior missed taints and vice versa. For some tests the results are the same between both analyses.

With this automated approach, we are limited to only see whether FILECHECK succeeded or failed. In case of a successfully test, we already know that the analysis produces results similar to a *perfect* analysis, and we do not need to investigate further. On the other hand, if a test does not succeed, we can not automatically deduce, whether the fail results from missing taints or falsely positive tainted instruction. For those tests, we need to look into detail of the analyses outputs and compare them by hand to the expected values.

As output from our analysis can be huge for large tests, because we are basically printing the LLVM IR with taint annotations, we needed a fast way to identify, list and compare tainted statements between two analyses. To ease this process, we wrote a tool to display output in a side-by-side manner with colored highlighting of tainted instructions.

An example with an arbitrary test can be seen in Figure 4.1. We color coded instruction if taints are only detected by the new or old analysis and if both analyses find the same data flow. If both analyses find the same taints, statements are colored blue, otherwise the difference is highlighted green / red for additions / missing taints respectively.

VaRA

```
_Z6FooGetb

   %2 = alloca i8, align 1 FTaints: {}
   %3 = zext i1 %0 to i8 FTaints: {}
   store i8 %3, i8* %2, align 1 FTaints: {}
   %4 = load i8, i8* %2, align 1 FTaints: {}
   %5 = trunc i8 %4 to i1 FTaints: {}
   br i1 %5, label %6, label %9 FTaints: {}

   %7 = load i8, i8* @Foo, align 1 FTaints: {Foo}
   %8 = trunc i8 %7 to i1 FTaints: {Foo}
   br label %12 FTaints: {}

   %10 = load i8, i8* %2, align 1 FTaints: {}
   %11 = trunc i8 %10 to i1 FTaints: {}
   br label %12 FTaints: {}

   %13 = phi i1 [ %8, %6 ], [ %11, %9 ] FTaints: {Foo}
   ret i1 %13 FTaints: {Foo}
⋮
```

VaRA & PhASAR

```
_Z6FooGetb

   %2 = alloca i8, align 1, !psr.id !8 FTaints: {}
   %3 = zext i1 %0 to i8, !psr.id !9 FTaints: {Bar, Baz}
   store i8 %3, i8* %2, align 1, !psr.id !10 FTaints: {Bar, Baz}
   %4 = load i8, i8* %2, align 1, !psr.id !11 FTaints: {Bar, Baz}
   %5 = trunc i8 %4 to i1, !psr.id !12 FTaints: {Bar, Baz}
   br i1 %5, label %6, label %9, !psr.id !13 FTaints: {Bar, Baz}

   %7 = load i8, i8* @Foo, align 1, !psr.id !14 FTaints: {Foo}
   %8 = trunc i8 %7 to i1, !psr.id !15 FTaints: {Foo}
   br label %12, !psr.id !16 FTaints: {}

   %10 = load i8, i8* %2, align 1, !psr.id !17 FTaints: {Bar, Baz}
   %11 = trunc i8 %10 to i1, !psr.id !18 FTaints: {Bar, Baz}
   br label %12, !psr.id !19 FTaints: {}

   %13 = phi i1 [ %8, %6 ], [ %11, %9 ], !psr.id !20 FTaints: {Foo, Bar, Baz}
   ret i1 %13, !psr.id !21 FTaints: {Foo, Bar, Baz}
⋮
```

Figure 4.1: Example of side-by-side view of analysis output with the old analysis on the left and the new with PhASAR on the right.

5 EVALUATION

In this chapter, we look at the results from the tests described in Section 4.2 (*Test Cases*) on both, the original VaRA taint propagation, and our replacement implementation using PhASAR for the data flow analysis. We will discuss each test suite separately and look into whether the tested properties are fulfilled by our analysis.

5.1 RESULTS

While the following tables are grouped by tested properties, an overview over all tests and their respective results running the original VaRA analysis and our new analysis with PhASAR is presented in Table A.1.

CONTROL

These tests show the improved capability for data-flow analysis with PhASAR, as we pass all. While the original analysis does perform decent, especially complicated control flow modifications like switch-statements may not work. Labrenz [9] stated in his thesis, that the original taint propagation of VaRA falsely taints exceptions, which we were not able to reproduce with our analysis.

Test	VaRA	VaRA & PhASAR
DoWhile	yes	yes
Exception	yes	yes
For	yes	yes
Goto	no	yes
IfGlobal	yes	yes
IfLocal	yes	yes
Switch	yes	yes
SwitchCases	no	yes
SwitchFallthrough	yes	yes
Unreachable	no	yes
While	yes	yes
Passed (of 11)	8	11

OPERATORS

Looking at any kind of operators, we are able to improve on the original analysis by now support all arithmetic operations. Using structs, especially testing on field-sensitivity shows, that PhASAR does not support this property yet.

Test	VaRA	VaRA & PhASAR
Arithmetic	no	yes
Bitmask	yes	yes
Cast	yes	yes
Index	no	no
Shift	yes	yes
Ternary	yes	yes
Passed (of 6)	4	5

DATA STRUCTURES AND POINTERS

We now support union types.

If we initialize our feature variable as global constant, if is directly optimizes by the compiler, therefore the feature variabel is missing in the first produced IR. Using a local constant shift the problem so that the compiler now keeps the allocation, but replaces any usage directly with the stored value.

Several test results are marked in brackets, as we are not able to produce comparable results with our new analysis. The load operation which works for most other tests, but for these test loading a global feature variable does not hold a taint. Sadly we will see this issue in the next scenarios more often and discuss it in Section 5.2 (*Issues*).

Test	VaRA	VaRA & PhASAR
Array	no	(no)
Const	no	no
FunctionPointer	no	yes
FunctionVirtual	no	(no)
FunctionVoid	no	(no)
Pointer	yes	yes
PointerGolf	no	yes
PointerStruct	no	(no)
Struct	no	no
StructBase	no	yes
Union	no	yes
UnionStruct	no	(no)
Passed (of 12)	1	5

FUNCTIONS

As PhASAR constructs an ESG, we now can analyse interprocedural data flow in a sophisticated manner. For the tests with structs, we again see, that our analysis is not field-sensitive. Five test are unable to load a feature variable.

Test	VaRA	VaRA & PhASAR
Call	no	(no)
CallChain	no	yes
CallParameter	no	(no)
FieldParameter	no	yes
Return	no	yes
StructParameter	yes	no
StructNested	no	(no)
StructReturn	no	no
Variadic	no	(no)
VariadicForwarding	no	(no)
Passed (of 10)	1	2

TEMPLATES

Generic templates are support by neither of the analyses, and our new one additionally fails once without even loading at loading a feature variable. We assume, that we do not support any kind of generic code.

Test	VaRA	VaRA & PhASAR
Template	no	(no)
TemplateBase	no	no
TemplateConst	no	no
TemplateMethod	no	no
TemplatePointer	no	no
Passed (of 5)	0	0

5.2 ISSUES

Most tests work out of the box. For some test we saw strange behaviour, as our analysis sometimes fails to taint the load of a feature variable.

This behavior does not occur for the original analysis and only in selected tests. We assume that the bug is actually part of PhASAR, therefore we only inherit it for our analysis.

We addressed this issue with the development team of PhASAR, but at the point of this publication, we have not found a solution or workaround yet.

An example of this bug can be seen in Listing 5.1. This program does not produce any usable results with our analysis. Interestingly, if we comment-out line 6, we taint the load of the feature variable correctly.

```
int main() {
  bool A[256];
  A[42] = Foo;
  // CHECK: [[L0:%.*]] = load i8, i8* @Foo, align 1, !psr.id !{{.*}} FTaints: {Foo}
  bool *B = A;
  bool Tainted = B[42];
  return 0;
}
```

Listing 5.1: Failing test program.

5.3 DISCUSSION

To sum up, we now pass 24 of 44 tests, while the old analysis only passed 14 taints of our suite. Given the already mentioned issues with PHASAR, we can still conclude, that our analysis performs better (measured on correctly propagated taints) than the original one.

We try to answer our research questions declared in Section 1.1 (*Goal of this Thesis*):

RQ1 Can we improve the detection accuracy of the rudimentary data-flow analysis used in VaRA[1] with an external tool?

> Yes. Given, that we showed with our vastly different test scenarios, that we are able to propagate way a lot more taints correctly than the original analysis, we can assume, that we improved our overall detection accuracy of the data-flow analysis.

RQ2 Can our new analysis correctly model common features and implementation patterns used in C/C++?

> As seen in Table A.1 we are able to model most common control flow modifiers and operators in C++. Functions are supported to a certain degree, while their may currently still be issues with some use cases, as discussed in Section 5.2 (*Issues*). Our analysis is not field-sensitive, therefore we only have rudimentary support of struct and classes, and none for templates.

RQ3 To what extent does our new analysis support different static analysis properties, such as flow-, context-, or field-sensitivity?

> As introduced in Section 2.3 (*Data-flow Analysis*) an IDE analysis is always context-sensitive. With the interprocedural ESG, our analysis is also flow-sensitive. On the other hand, as we saw with our test cases, the analysis is not field-sensitive. Field-sensitive patterns (like structs) have only rudimentary support, which is the expected behaviour without field-sensitivity.

[1] https://github.com/se-sic/VaRA-Tool-Suite (visited on 22/11/2021)

5.4 THREATS TO VALIDITY

While using *The C++ Programming Language*[17] as reference on C++ language features, this analysis is obviously biased on our selection of common languages features. We tried to minimize this effect by systematically constructing test cases from the language specification of C++.

Our more complex tests depend on the correctness of basic taint propagation. For example, testing behaviour inside a function requires, that the taint for function calls is already correctly handled in the first place. As this may have become a major issue, in practice we recognized, that on one hand basic syntactical structures are in most cases already perfectly handled and on the other hand constructed our tests trying to minimize the overlap of tested patterns.

Given, that all assertions on which instructions should be tainted are dependent on the correctly writen file checks, we may have wrong assertions at some point. As this can only prevented to some degree by carefully writing those tests, we tried to minimize those bugs by evaluating each tests several times and constructing scenarios in a way, that it should be pretty clear, whether an instruction should be tainted or not. We additionally compared our results (as far as they are also covered) with Labrenz [9], our analyses have different behaviour and looked into detailed reasoning, why this may be the case.

6 RELATED WORK

A very similar approach to improve the accuracy of a detection algorithm in VaRA was made by Labrenz [9]. He compared the data-flow analyses of VaRA and PhASAR in a similar fashion, but mainly focused on commit regions instead of feature regions. We improve on his test suite by adding sophisticated assertion using FileCheck and adapted all tests to propagate taints of feature variables. To improve the coverage of modern language features, we added several tests for modern C++ like generic templates. We tried to improve on the results, aiming at a more elaborated measurement of our analysis properties. Our test suite is intended to be reusable within the VaRA framework.

A more recent paper was written by Allen, Gauthier, and Jordan [2], utilising static taint analysis to detect vulnerabilities in web applications. They aim at adapting the IFDS algorithm on very large industrial Java projects and proof that this type of analysis is still researched, and a very successful approach to solve data-flow problems.

Our main framework for region analyses is VaRA, which was introduced by Sattler [15]. We improved on the original data-flow analysis by replacing it with PhASAR to provide a more precise detection of feature regions.

The initial publication on PhASAR by Schubert, Hermann, and Bodden [16] described the algorithmic approach and allows us to inherit the properties of this analysis for our work. For example, as PhASAR's data flow analysis is already context-sensitive, we can assume the same holds for our analysis.

7 CONCLUSION AND OUTLOOK

In this chapter we want to summarize previous findings and give a recommendation on whether to use the new over the old analysis. Last, we will look into the future, how others can benefit from our work and what we can improve in our own framework to make taint propagation even more precise and increase the amount of obtained information about code regions.

7.1 SUMMARY

This thesis looked into theory and implementation of data-flow analyses and how we can use LLVM to implement a pipeline with VaRA and PhASAR to detect feature regions. We have shown an example of how we annotate LLVMIR with metadata and how we extract relevant information after running our compiler passes.

To measure whether we can expect an improvement of the new analysis with PhASAR over the old analysis, we evaluated both new data-flow analysis on a large test suite and compared the results. This showed that for several test cases PhASAR performs better and more robust in regard to more complex language features of C++.

While we had some issues with processing taints, we can still conclude with the remaining results, that we can improve the feature-region detection by using an external analysis tool, in this case we decided on PhASAR.

We have seen no disadvantage from using the new analysis over the old one, and therefore strongly encourage the use of PhASAR within VaRA for further work with data flow problems.

7.2 FURTHER IMPROVEMENTS / INTEGRATIONS

As discussed in Section 5.2, we encountered some isolated issues with taint initialization and need to investigate on possible bugs further, as we were not able to resolve them so far. We currently assume, the problem originates from PhASAR itself, but we still wait for feedback from their development team. To keep track of this progress, we provided all failing test cases as individual branches / repository issues in our codebase.

For the feature-region detection in VaRA, we are currently developing a dominator / post-dominator taint propagation, which allows propagating a taint to not only the first feature related code block, but all dominated instructions as well, therefore detecting more affected instructions. This does not only increase the size of the acquired information on feature regions, but also works in addition to the current detection only improving the current results.

It would be interesting to evaluate our analysis on larger scale industrial projects. So far, all our tests have been purely synthetically created to test specific properties of the data-flow analysis, so we do not know yet, how we perform on real-world software. As we are currently developing a feature library for VaRA, we may construct feature variables directly from source locations given in a feature model.

Unrelated to the implementation of the data-flow analysis itself, we would like to expand on how feature variables may be initialized and used to switch-on features in VaRA. An example of such a pattern, which we currently do not support, is to use a function pointer, which either points to a method executing code of a feature, enabling it dynamically, or some default implementation. This differs from our current approach, where we can only detect conditional and branch statements to create feature regions.

It may be worth to look into other data-flow analyses frameworks, which may even perform better than PhASAR. At the time of this publication, field-sensitivity for PhASAR is already in development. This would hopefully resolve some of our failing test cases regarding data structures.

A APPENDIX

Table A.1: Overview of all test cases.

	Test	VaRA	VaRA & PhASAR
CONTROL	DoWhile	yes	yes
	Exception	yes	yes
	For	yes	yes
	Goto	no	yes
	IfGlobal	yes	yes
	IfLocal	yes	yes
	Switch	yes	yes
	SwitchCases	no	yes
	SwitchFallthrough	yes	yes
	Unreachable	no	yes
	While	yes	yes
OPERATORS	Arithmetic	no	yes
	Bitmask	yes	yes
	Cast	yes	yes
	Index	no	no
	Shift	yes	yes
	Ternary	yes	yes
DATA STRUCTURES AND POINTERS	Array	no	(no)
	Const	no	no
	FunctionPointer	no	yes
	FunctionVirtual	no	(no)
	FunctionVoid	no	(no)
	Pointer	yes	yes
	PointerGolf	no	yes
	PointerStruct	no	(no)
	Struct	no	no
	StructBase	no	yes
	Union	no	yes
	UnionStruct	no	(no)
FUNCTIONS	Call	no	(no)
	CallChain	no	yes
	CallParameter	no	(no)
	FieldParameter	no	yes
	Return	no	yes
	StructParameter	yes	no
	StructNested	no	(no)
	StructReturn	no	no
	Variadic	no	(no)
	VariadicForwarding	no	(no)
TEMPLATES	Template	no	(no)
	TemplateBase	no	no
	TemplateConst	no	no
	TemplateMethod	no	no
	TemplatePointer	no	no
	Passed (of 44)	14	24

BIBLIOGRAPHY

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[2] Nicholas Allen, François Gauthier, and Alexander Jordan. *IFDS Taint Analysis with Access Paths*. 2021.

[3] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[4] Eric Bodden. "Inter-procedural data-flow analysis with IFDS/IDE and Soot." In: *International Workshop on State of the Art in Java Program analysis*. Beijing, China: ACM.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph." In: *ACM Transactions on Programming Languages and Systems* (1991).

[6] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. "Generative Programming." In: *Object-Oriented Technology*. Lecture Notes in Computer Science. Springer, 2002.

[7] Christian Kaltenecker, Alexander Grebhahn, Norbert Siedmung, Jianmei Guo, and Sven Apel. "Distance-Based Sampling of Software Configuration Spaces." In: *Software Engineering, Fachtagung des GI-Fachbereichs Softwaretechnik*. Innsbruck, Austria, 2020.

[8] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. Software Engineering Institute, 1990.

[9] Simon Labrenz. "An Empirical Comparison of Static Taint Analyses from VaRA and PhASAR." Bachelor's Thesis. Germany: University of Passau, 2019.

[10] Chris Lattner. "LLVM." In: *The Architecture of Open Source Applications*. Vol. 1. 2011.

[11] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation." In: *International Symposium on Code Generation and Optimization*. Palo Alto, California, USA, 2004.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[13] LLVM Project. *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html (visited on 22/11/2021).

[14] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability." In: *Symposium on Principles of Programming Languages*. San Francisco, California, USA: ACM Press, 1995.

[15] Florian Sattler. "A Variability-Aware Feature-Region Analyzer in LLVM." Master's Thesis. Germany: University of Passau, 2017.

[16] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Interprocedural Static Analysis Framework for C/C++." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science. Springer, 2019.

[17] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley, 2013.