Bachelor's Thesis

FEATURE TAINT ANALYSIS: HOW PRECISE CAN VARA TRACK THE INFLUENCE OF FEATURE VARIABLES IN REAL-WORLD PROGRAMS?

JANIK KELLER

March 7, 2023

Advisor:
Florian Sattler   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

The configuration options of features in modern software systems are often implemented with the help of runtime parameters and representative variables in the source code. Accesses to the data of these variables are referred to as feature taints. The feature taints of a program expose valuable information about the code regions that implement the functionalities of features.

The analysis framework VaRA provides a static detection of the feature-related code that can be of great use for developers when it comes to debugging and optimisation. An integral part of this detection is the feature taint analysis (FTA), which has not been thoroughly evaluated to this day. Our evaluation on 4 real-world software projects found that the feature taint analysis (FTA) detects correct feature taints with a high certainty, but still leaves room for improvement. The current version of the analysis probably provides VaRA's feature-region detection with the information that are necessary in order to determine the correct feature-regions. However, inaccuracies are still possible and need to be addressed in future revisions of the feature taint analysis (FTA).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

FAR	feature analysis report
FOSD	feature-oriented software development
FTA	feature taint analysis
IDE	integrated development environment
IR	intermediate representation
PTA	PhASARFeatureTaintAnalysis
SPL	software product line
VaRA-TS	VaRA-Tool-Suite

1 INTRODUCTION

Most modern software systems are configurable in order to fit the individual needs of different users with varying use cases and application scenarios. To offer this flexibility, software has a set of configuration options among which the users can choose to enable certain functionalities. A feature provides these configuration options and extends or modifies the program to implement the respective functionalities. By selecting or deselecting configuration options, users are able to create tailor-made configurations of the software. For example, a user of the workplace messaging service SLACK[1] may want to send photos without compression. The software allows him or her to do so by deselecting the respective option in the menu.

In case a feature does not work according to the wishes of the users, the developers need to adapt the software code. For the developers, it is then interesting to know what code regions implement which feature. This supports them in optimization and debugging of the feature-related code. In case there is some problem with the image compression of SLACK, we can find the relevant regions of the source code by analysing the data-flow. We identify some variable in the code that represents the configuration option of the feature and search for statements where the data of the variable is accessed. This process is called feature taint analysis (FTA) and is described in detail in Section 2.4.2. In a next step, we can identify the conditional control flow statements that were identified by the feature taint analysis (FTA). With these, we can draw conclusions about the feature-dependent control flow decisions and detect the feature-regions.

The analysis framework VARA[2] implements such a feature-region detection. The feature taint analysis (FTA) is given the source code to analyse and a mapping from features to the code locations of the representative variables. The tool performs a static analysis to generate modified LLVM intermediate representation (IR)[3] code containing the feature information. With this, VARA can detect the regions of code that implement the features' functionalities.

A correct feature-region detection needs a properly working feature taint analysis (FTA). If wrong statements are tainted, wrong feature-regions will be detected. VARA's FTA has not yet been thoroughly evaluated on real-world software, only on small test cases. But as real-world programs are way more complex, running the analysis on these gives us better information about how well the feature taint analysis (FTA) performs. Furthermore, we can determine shortcomings of the employed analysis and identify bugs to fix in future versions of VARA.

1 `https://slack.com/` (visited on 03/07/2023)
2 `https://github.com/se-sic/VaRA` (visited on 03/07/2023)
3 `https://llvm.org/docs/LangRef.html` (visited on 03/07/2023)

1.1 GOAL OF THIS THESIS

In this thesis, we evaluate VaRA's feature taint analysis (FTA). We investigate the source code of real-world programs in an integrated development environment (IDE), and determine manually the relevant statements that should be tainted by tracing the feature variables. Then, we run the analysis on these programs and compare the detected feature taints with the previously determined ground truth. Doing this with a number of real-world projects gives us quantitative data that we use to reason about the overall quality of VaRA's feature taint analysis (FTA). Furthermore, we use the data to mine problematic corner cases, which we then convert into small code examples that can be reasoned about.

1.2 OVERVIEW

This thesis is divided into six parts. In Chapter 2, we provide the theoretical foundation that we need for our work. We introduce the core concepts of feature-oriented software development (FOSD), the compiler infrastructure LLVM, different forms of data-flow and data-access, and the variability-aware region analyser VaRA. In Chapter 3, we explain how we can conveniently run VaRA's feature taint analysis (FTA) in order to to generate a structured output that contains all the information that we need for our evaluation. Chapter 4 introduces the operationalisation of the evaluation. Here, we talk about the data that we collect and how we use it in order to reason about the precision of VaRA's feature taint analysis (FTA). The results of our research are presented and discussed in Chapter 5. In Chapter 6, we contextualise our work in the research field. Last, we summarise our results in Chapter 7 and provide an outlook on future work.

2 BACKGROUND

This chapter introduces the core concepts on which we build upon. We explain central concepts of software product lines (SPL), such as features, configurations, and feature models. We also take a look at the compiler infrastructure LLVM and its intermediate representation (IR), with focus on the instructions that handle the control flow of a program, called terminator instructions. Then, we take a brief look at different types of data-flow and data-access. Next, we introduce the feature taint analysis (FTA) and the feature-region detection of VaRA. At the end of the chapter, we explain how to use the Tool-Suite of VaRA to automatically run experiments over multiple real-world software projects, as well as evaluate the precision of the feature taint analysis (FTA).

2.1 FEATURE-ORIENTED SOFTWARE DEVELOPMENT

Different users have different aims in different contexts when they use a software system. For that reason, they should be able to tailor the software individually, by choosing the functionalities they need. Software should be designed appropriately to achieve this goal. For that purpose, the term of feature-oriented software development (FOSD) was introduced. Apel and Kästner [1] defined FOSD as

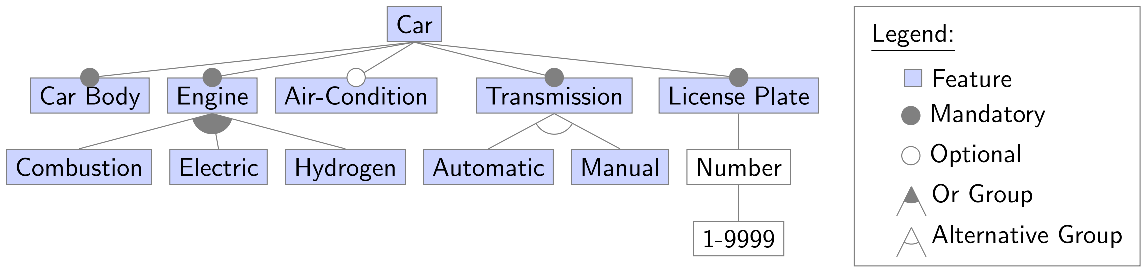
> A paradigm for the construction, customization, and synthesis of large-scale software systems.

FOSD serves as an umbrella term of general development principles that cover the whole software lifecycle instead of presenting a single method or technique. The crucial element that connects all of the approaches in FOSD are *features*. They provide functionalities and configuration options among which users can choose in order to tailor a system to their needs. The set of possible variants of a software product is called a software product line (SPL) [8].

In the following sections, we explain the role of features in FOSD and present a way to describe the dependencies among the features of a SPL using feature models.

2.1.1 *Features and Configurations*

The different variants of a SPL differ in the features they realise. To get a tailored software product, the user has to select the features of interest. This set of features that specifies a software variant is called a configuration. Usually, not all combinations of features describe possible system variants. Take for example a SPL of a car where the customer can choose between an automatic and a manual transmission. A car with both transmission types is not possible, and therefore a configuration with both features invalid. We call a configuration valid if it specifies a possible system variant.



$$\textit{Electric} \implies \textit{Air-Condition}$$

Figure 2.1: Exemplary feature diagram of a car

There are a lot of different ways to define a feature [7]. Basically, these definitions take two perspectives on the concept of a feature. While some see them as domain-specific abstractions that represent requirements of the users [6, 15], others take a more technical perspective on features by comprising the fact that the requirements must be implemented in a way [2, 4].

Czarnecki and Eisenecker [9] distinguish between problem space and solution space. The problem space considers the users requirements on a software system and therefore consists of the set of all valid configurations. The solution space consists of the actual software systems that realise the configurations. With this distinction, the definitions above can be classified. The domain-specific perspective on features corresponds to the perspective of the problem space, while the implementation-oriented perspective fits the perspective of the solution space.

For our work, we need a more implementation-oriented definition of a feature that describes it from the perspective of the solution space as VaRA detects the code regions that implement a feature. We use the following definition by Apel et al. [2]:

> A feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option.

This technical approach describes features as code extensions of a software system that realise the requirements of and can be selected by the users. Specific parts of the source code are dedicated to the features and implement them.

2.1.2 *Feature Models*

We use feature models to describe the features of a system and the dependencies among them [15]. They are a precise way to present the valid configurations of a SPL. The graphical depiction of a feature model as seen in Figure 2.1 is called a feature diagram. We use an extended version of the feature diagram notation introduced by Czarnecki and Eisenecker [9].

Figure 2.1 shows the feature model of a car SPL. The nodes of the feature diagram represent the features, with the blue ones being binary features and the white ones being numeric features. A feature can only be selected if its parent feature is selected. The root of

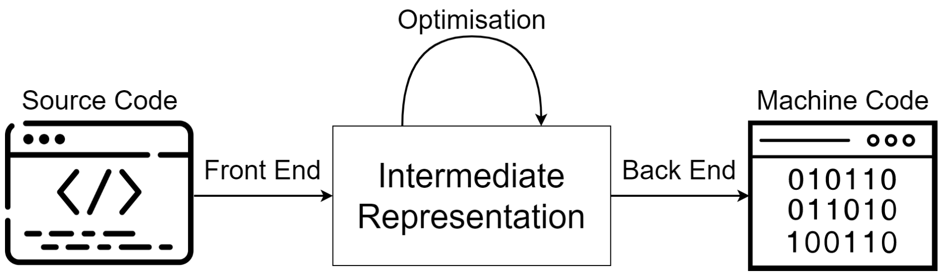


Figure 2.2: Basic compilation pipeline

the diagram must always be selected. A filled circle marks a feature as mandatory, what means that it has to be part of the configuration if its parent is. An empty circle marks a feature as optional, what means that it can but must not be selected. An empty arc at the edges to the children indicates an alternative group, i.e., exactly one of the children must be enabled. The filled arc represents an or group, that is, at least one child must be selected. Numeric features are always part of the configuration, no matter if the parent is enabled or not. The child of a numeric feature contains the possible values the feature can have. Dependencies among the features that cannot be expressed in the hierarchy if the diagram, so-called cross-tree constraints, are listed as propositional formulas right under the feature diagram.

In our example above, every car needs to have a car body, an engine, a transmission, and a license plate, as these are mandatory features. A car may or may not have an air condition. The engine works either per combustion, electric, or with hydrogen but also every hybrid combination is possible, for instance an engine that can switch between combustion and electric. The transmission must be automatic or manual but not both and the license plate has a number with a value in the range from at least 1 and at most 9999. A car with an electric engine always has an air-condition as the cross-tree constraint indicates.

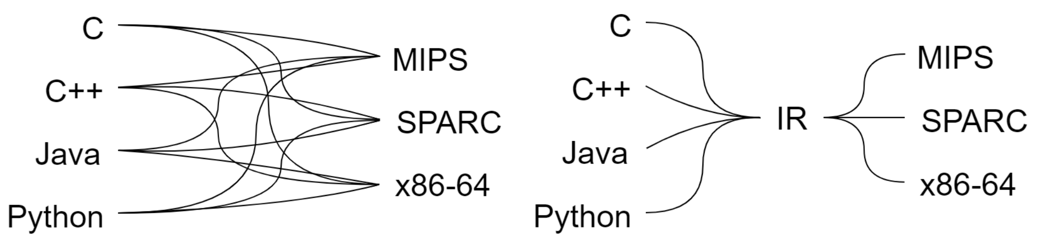
2.2 LLVM

The LLVM framework[1] comprises a number of technologies related to compilers and toolchains, e.g., the C language family compiler Clang[2]. The starting point of LLVM was as a language-independent compiler framework with the goal of code optimisation [17]. The optimiser nowadays forms the core of the project. It operates on an IR which we discuss in the next section.

2.2.1 *LLVM Intermediate Representation*

A compiler translates human-readable source code of a programming language into machine-readable executable code. Often, compilers transform the source code into an abstract machine language, the so-called intermediate representation (IR), before generating the machine code. The IR serves as a language-independent code representation on which

1 https://llvm.org/ (visited on 03/07/2023)
2 https://clang.llvm.org/ (visited on 03/07/2023)



(a) Without IR (b) With IR

Figure 2.3: Compilers for source languages and target machines. (a) Without an IR, you need one compiler for every combination of language and machine. (b) With an IR, you only need one front end per language and one back end per machine.

the compiler can operate to analyse and optimise the generated machine code. So, in an intermediate step, the IR gets optimised, potentially several times, before the machine code is produced. The front end of the compiler generates the IR from the source code and the back end transforms the IR to the machine code. This basic compilation pipeline is depicted in Figure 2.2.

The IR consists of a set of instructions similar to the instructions of a machine language but without being bound to a specific machine. So, it is independent of the source language as well as of the machine language. Figure 2.3 visualises this property. Without an IR, it would need one compiler for every combination of source language and machine language. With an IR, it only takes one front end per source language and one back end per machine language. This brings modularity, as a compiler can than easily be created by a combination of both. [3]

The LLVM compiler framework includes the optimiser Opt[3] as well as a back end for a number of machines[4]. It also provides several so-called passes[5]. These are optimisations and analyses that can be run with Opt. Passes can be divided into different categories. For our work, we only distinguish between analysis passes and transform passes. Analysis passes investigate the program and compute information, such as alias information. Transform passes modify the IR, for instance by deleting dead code or unrolling loops.

The LLVM compiler framework operates on LLVM IR. We use the equivalent human-readable assembly language representation of LLVM IR when showing corresponding instructions or code fragments. The reference manual for the assembly language can be found online[6]. For the purpose of our work, there is no need to explain the IR in detail. The instruction set of the assembly language is subdivided into several classes of instructions. We only focus on terminator instructions, more precisely BR and SWITCH instructions because these are important for our evaluation as we explain in Section 4.1.

3 `https://llvm.org/docs/CommandGuide/opt.html` (visited on 03/07/2023)
4 `https://llvm.org/docs/CodeGenerator.html` (visited on 03/07/2023)
5 `https://llvm.org/docs/Passes.html` (visited on 03/07/2023)
6 `https://llvm.org/docs/LangRef.html` (visited on 03/07/2023)

2.2.1.1 *Terminator Instructions*

Within LLVM, terminator instructions manage the control flow of a program. The terminator instructions BR and SWITCH evaluate a condition and direct the control flow accordingly. In the following, we take a look at these. Paraphrased from the LLVM reference manual:

BR INSTRUCTION[7]

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

DESCRIPTION: The BR instruction directs the control flow after checking the boolean value `<cond>`. If it is `true`, the program execution branches to the label `<iftrue>`, if it is `false`, it branches to the label `<iffalse>`. It is semantically similar to IF statements of high-level programming languages like C or C++. The BR keyword also accepts only one label without a condition to perform an unconditional branch.

EXAMPLE: `br i1 %cond, label %IfThen, label %IfElse`
If the value of `cond` is `true`, go to label `IfThen`. If it is `false`, go to label `IfElse`.

SWITCH INSTRUCTION[8]

SYNTAX: `switch <intty> <value>, label <default> [ <intty> <val>, label <dest> ... ]`

DESCRIPTION: The SWITCH instruction compares the integer value `<value>` with the integer values `<val>` of the given array of pairs of integers and labels. If an equal value is found, the program execution branches to the corresponding label `<dest>`. In case `<value>` does not match any value of the array, the execution continues at the label `<default>`. The SWITCH instruction is semantically similar to the SWITCH statement of high-level programming languages like C or C++.

EXAMPLE: `switch i32 %val, label %default [ i32 0, label %IfZero  i32 1, label %IfOne ]`
If the value of `val` is `0` or `1`, go to label `IfZero` or `IfOne`, respectively. Otherwise, go to label `default`.

2.3 DATA-FLOW & DATA-ACCESS

We distinguish between implicit and explicit flows when it comes to analysing the data-flow of a program. An explicit flow arises if the data of a variable is directly assigned to another variable. For instance, in the assignment Y := X, the variable X explicitly flows to the variable Y. On the other hand, we talk about an implicit flow if a variable is assigned due to a control flow decision dependent on another variable. In the code fragment IF (X) THEN Y := TRUE; ELSE Y := FALSE, the variable X implicitly flows to Y. Here, Y is implicitly assigned to the value of X by a control flow decision dependent on X. [16]

For this work, we also introduce the concept of direct and indirect data-access. We speak of a direct access to a variable X when we access the data of X by directly referencing X itself. Now assume that X explicitly flows to another variable Y. Then, we speak of an indirect

7 https://llvm.org/docs/LangRef.html#br-instruction (visited on 03/07/2023)
8 https://llvm.org/docs/LangRef.html#switch-instruction (visited on 03/07/2023)

```
 1 int main(int argc, char *argv[]) {
 2
 3   int Foo;
 4
 5   if (Foo) {
 6     Foo = 1;
 7   }
 8
 9   int local = Foo;
10
11   if (local) {
12     return 42;
13   } else {
14     return 0;
15  }
16 }
```

Listing 2.1: Direct and indirect access. Direct accesses to `Foo` are highlighted red, indirect accesses to `Foo` are highlighted orange.

access to X whenever the data of Y is accessed. In Listing 2.1, we can see 3 accesses to the variable `Foo`. In Line 5 and 9, the data of `Foo` is accessed by referencing `Foo` itself. Therefore, these are direct accesses. In Line 9, `Foo` explicitly flows to `local`. Hence, the direct access of `local` in Line 11 is also an indirect access of `Foo`.

2.4 VARIABILITY-AWARE REGION ANALYSER

The variability-aware region analyser VaRA is a framework that provides access to static and dynamic analyses of C/C++ source code. The overall goal of the analyses is to detect code regions that users are interested in. VaRA uses a customised Clang compiler to generate modified LLVM IR code and implements passes for its customised LLVM optimiser Opt to operate on it. One of the provided analyses is the static detection of the feature code regions, introduced by Sattler [22] and later extended with PhASAR[9] support by Timm [24]. The static analysis framework PhASAR offers implementations for solving data-flow problems on the IR of LLVM. With the inclusion of PhASAR, the feature taint analysis (FTA) was introduced in VaRA. The FTA, also known as feature taint propagation, is an integral part of the feature-region detection pipeline and subject of our evaluation.

In the following sections, we take a look at VaRA's FTA as well as its feature-region detection. We also explore the Tool-Suite that provides means for using VaRA and that we utilise for our research.

9 `https://phasar.org/` (visited on 03/07/2023)

```
 1  #include <string.h>
 2
 3  int main(int argc, char *argv[]) {
 4
 5    int Foo = 0;  // Feature Variable
 6
 7    if (argc >= 2 && !strcmp(argv[1], "--foo")) {
 8      Foo = 1;
 9    }
10
11    if (Foo) {
12      return 42;
13    } else {
14      return 0;
15    }
16  }
```

Listing 2.2: C example with feature variable `Foo`. The feature taints are highlighted red and the feature-regions are highlighted blue.

2.4.1 *Feature-Region Detection*

The aim of VaRA's feature-region detection is to identify the feature-regions of a given C or C++ project. In order to understand this process, we first need to clarify what a feature-region actually is.

2.4.1.1 *Feature-Regions*

VaRA's feature-region detection expects features to be implemented with the help of runtime parameters. By selecting or deselecting a feature, e.g., via command-line options, the user influences a certain variable at runtime. This means that there are variables in the source code, the so-called feature variables, that represent the configuration options of the SPL. Through control flow decisions dependent on the feature variable, e.g., in the form of IF conditions, feature-related code fragments are executed. These code fragments implement the functionality of the feature and are called feature-regions.

Listing 2.2 shows a simple C program with the optional feature *Foo* which can be selected by the user per command-line option. If the user sets the respective flag, the variable `Foo` is touched in Line 8. This allows us to identify `Foo` as the feature variable of *Foo*. Since `Foo` is subject of the condition of the IF statement in Line 11, the THEN block and the ELSE block of the statement are its feature-regions. They are highlighted in blue.

2.4.1.2 *Pipeline Overview*

Now that we know what the feature-region detection is supposed to find, we take a look at how VaRA achieves this. There is more than one feature-region detection implemented in VaRA but their differences are not important for this work. The aspects of it that are interesting for us are the same throughout all the implementations. So, we explain one exemplary and generalised detection pipeline to convey these aspects.

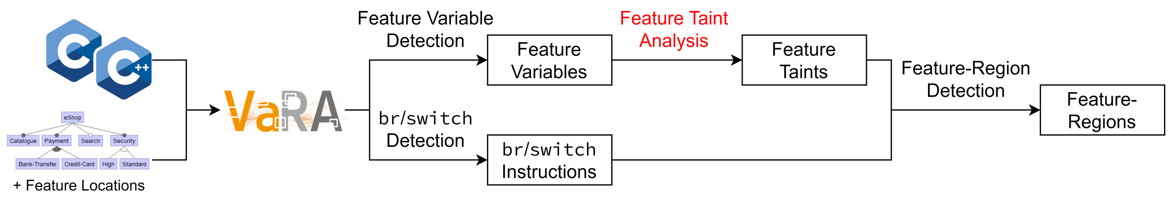


Figure 2.4: Pipeline of VaRA's feature-region detection

In Figure 2.4, we see the processing pipeline of VaRA's feature-region detection. As input, we give VaRA the C/C++ source code to analyse and information about the features of interest containing the locations of their respective feature variables in the source code, i.e., the mapping from feature to feature variable in the code. This information can be transmitted in the form of a feature model, for instance. VaRA then translates the source code to LLVM IR and produces modified IR code with annotated feature-regions as the result of the analysis.

An intermediate step of the pipeline is the detection of the BR and SWITCH instructions. Feature-regions always occur in combination with conditional control flow decisions and these are the instructions of LLVM IR that produce this type of control flow. Hence, VaRA annotates every BR and SWITCH in the generated IR.

In parallel with the BR/SWITCH detection, VaRA marks the feature variables in the generated IR code based on the locations given as input. Then, the FTA is run to trace the data-flow of the feature variables and find the feature taints of every feature. A more detailed explanation of the FTA is given in the next section.

In the last step, VaRA builds the intersection of feature taints and BR/SWITCH instructions, i.e., it identifies every tainted BR and SWITCH in the LLVM IR. Those instructions produce conditional control flow dependent on the feature variable. Therefore, the branching options that correspond to the instructions form the feature-regions. [24]

2.4.2 *Feature Taint Analysis*

The feature taint analysis (FTA) plays an important role in the pipeline of VaRA's feature-region detection. The principle of taint analysis underlies the FTA. A taint analysis traces the flow of data that comes from outside the system, e.g., via the command-line parameters `argc` and `argv`, and identifies the situations, in which this data may be leaked again to the outside, e.g., via print-methods. Functions that read data from outside the system are referred to as sources and functions that can leak data to the outside are referred to as sinks. The data from outside is considered as tainted and the goal of a taint analysis is to report whenever tainted data is accessed by a sink. [23]

In contrast to taint analysis, we are not interested to trace data from outside and find potential data leaks with the FTA. Instead, VaRA's FTA tracks the data of feature variables and identifies the instructions that access it. The FTA considers feature variables as the sources and instructions as the sinks. Whenever an instruction accesses the tainted data of a feature variable, the instruction is tainted by the respective feature.

It is worth to mention that VaRA's FTA, as well as VaRA's feature-region detection, are interprocedural analyses. They take the data-flow across function calls into account and

```
 1 main
 2   entry
 3       %retval = alloca i32, align 4, !psr.id !5 FTaints: {}
 4       ...
 5       %Foo = alloca i32, align 4, !FVar !8, !psr.id !9 FTaints: {Foo}
 6       ...
 7       br i1 %cmp, label %land.lhs.true, label %if.end, !psr.id !16 FTaints: {}
 8   land.lhs.true
 9       ...
10       br i1 %tobool, label %if.end, label %if.then, !psr.id !22 FTaints: {}
11   if.then
12       store i32 1, i32* %Foo, align 4, !psr.id !23 FTaints: {}
13       br label %if.end, !psr.id !24 FTaints: {}
14   if.end
15       %3 = load i32, i32* %Foo, align 4, !psr.id !25 FTaints: {Foo}
16       %tobool1 = icmp ne i32 %3, 0, !psr.id !26 FTaints: {Foo}
17       br i1 %tobool1, label %if.then2, label %if.else, !psr.id !27 FTaints: {Foo}
18   if.then2
19       store i32 42, i32* %retval, align 4, !psr.id !28 FTaints: {}
20       br label %return, !psr.id !29 FTaints: {}
21   if.else
22       store i32 0, i32* %retval, align 4, !psr.id !30 FTaints: {}
23       br label %return, !psr.id !31 FTaints: {}
24   return
25       %4 = load i32, i32* %retval, align 4, !psr.id !32 FTaints: {}
26       ret i32 %4, !psr.id !33 FTaints: {}
```

Listing 2.3: Intermediate representation (IR) excerpt of Listing 2.2 with annotated feature taints

analyse the whole program, rather than single functions. For the FTA, this basically means that tainted data which is passed as a parameter of a function call is also considered as tainted inside the function. So, whenever an instruction of the respective function accesses the tainted data of the parameter, the FTA should identify this instruction as tainted.

In Listing 2.2, the feature taints are highlighted red. When we trace the feature variable `Foo`, we see that the data of the variable is only accessed insight the IF condition in Line 11. Accordingly, the only feature taints of the program are related to this IF condition. Note, that the other occurrences of the feature variable in the Lines 5 and 8 are no feature taints, since the variable is only assigned to values and the data of the variable is not used.

An excerpt of the IR module as we get it when we run the FTA on Listing 2.2 is shown in Listing 2.3. Here, the feature taints are annotated behind the respective instruction. All tainted instructions are highlighted red. There are four tainted instructions in the IR, all of them are taints of the feature *Foo* with the feature variable `Foo`. The instructions in Line 15-17 realise the IF condition and as expected, they are tainted. The other feature taint in Line 5 belongs to the memory allocation of the feature variable and does not directly map to any statement of the high-level code. Memory allocations of feature variables are initially tainted by the FTA. So, the output of the FTA perfectly matches the feature taints that we previously identified in the high-level code.

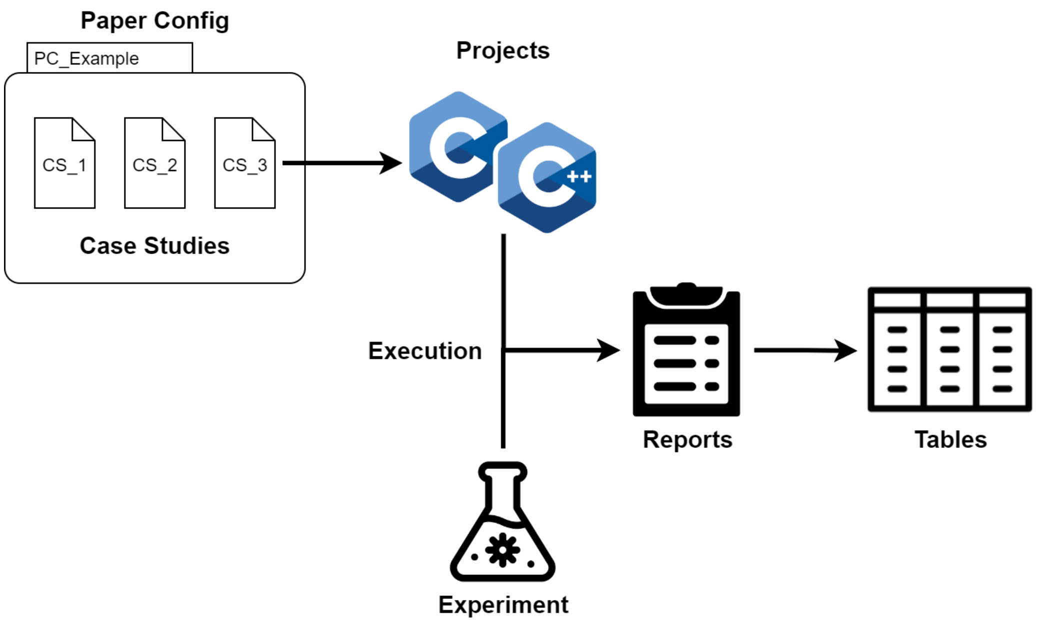


Figure 2.5: VaRA-Tool-Suite pipeline

2.4.3 *Tool-Suite*

VaRA-Tool-Suite (VaRA-TS)[10] is a framework written in Python that provides research tools to ease the use of VaRA in order to execute experiments on real-world software projects. Figure 2.5 gives an overview of the tools that we use and shows how they work together in VaRA-TS. In the following, we briefly describe them.

PROJECTS[11] Projects are used to access software projects with C/C++ source code. They contain information about how to build and execute the respective software project, and serve as a key abstraction to write project-independent experiments. A number of projects is already shipped with VaRA-TS.

CASE STUDIES[12] A case study stores a set of revisions of a project. So, if we want to run an experiment several times over the same project revisions, we can use case studies. This allows us to make the experiment results of a project conveniently reproducible. VaRA-TS ships with the command-line tool `vara-cs`[13] that gives handy support in the management of case studies.

PAPER CONFIGS[14] Paper configs are collections of case studies. With a paper config, we can group together all the projects and revisions used in the thesis and make them easily accessible for reproducing the evaluation results. A paper config is realised

10 `https://vara.readthedocs.io/en/vara-dev/index.html` (visited on 03/07/2023)
11 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/projects.html` (visited on 03/07/2023)
12 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/paper.html#how-to-use-case-studies` (visited on 03/07/2023)
13 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/tools/vara-cs.html` (visited on 03/07/2023)
14 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/paper.html#how-to-use-paper-configs` (visited on 03/07/2023)

as a directory which contains the case studies of interest. Similar as for case studies, VaRA-TS provides the command-line tool `vara-pc`[15] for support in the use of paper configs.

EXPERIMENTS[16] An experiment declares a set of steps that should be executed. We use them to set how VaRA should compile and optimise the respective projects. They separate experiment specific steps from project specific ones in order to make experiment execution project-independent and reproducible. We can use the command-line tool `vara-run`[17] to run experiments with VaRA-TS. It operates on the projects as specified in the case studies of the selected paper config.

REPORTS[18] The results of experiments are documented in the form of reports. Reports are stored in report files and we can interact with the data in a report file by creating a report class in VaRA-TS. A report class parses the content of the file and makes it accessible for further analyses.

TABLES[19] With tables, we can process reports and other input files to generate and visualise tabular data. We can create tables by implementing a table class. VaRA-TS provides the command-line tool `vara-table`[20] to automatically generate a table for the successfully produced reports, restricting the data for the table to the projects and revisions specified in the paper config. When implementing a table, we specify what arguments the tool expects.

15 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/tools/vara-pc.html` (visited on 03/07/2023)
16 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/experiments.html` (visited on 03/07/2023)
17 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/tools/vara-run.html` (visited on 03/07/2023)
18 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/data.html` (visited on 03/07/2023)
19 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/tables.html` (visited on 03/07/2023)
20 `https://vara.readthedocs.io/en/vara-dev/vara-ts-api/tools/vara-table.html` (visited on 03/07/2023)

3 DETECTING FEATURE TAINTS

Up to now, VaRA's FTA has only been evaluated on small, synthetic test cases [24]. However, to get an idea about the overall performance in actual use cases, the analysis should also been tested on real-world programs. So, we first want run the FTA on a number of projects and later on compare the detected feature taints with a ground truth. In this chapter, we present a way to automatically run the FTA by using VaRA-TS. We create an experiment that generates a report which contains all the relevant information for the evaluation.

3.1 RUNNING THE FEATURE TAINT ANALYSIS

We need to run the FTA on several projects to collect expressive data for the evaluation. In order to do this, we create an experiment named PhASARFeatureTaintAnalysis (PTA) in VaRA-TS. It automatically calls VaRA's Clang and Opt and provides a consistent and convenient way to compile projects to LLVM IR and to run the FTA on them. The output of the PTA-experiment is a report that we discuss in Section 3.2. In case a paper config is selected, we can execute the experiment on its case studies with a simple command-line call:

```
$ vara-run -E PhASARFeatureTaintAnalysis
```

The PTA-experiment uses the modified Clang compiler of VaRA to translate projects to LLVM IR. In the implementation, we specify the following options to call Clang with:

-O1 Generates IR with optimisation level 1. The optimisation levels are predefined sets of options for Clang and Opt. Opt is called implicitly after Clang produced the IR. Optimisation level 1 includes an analysis pass that annotates the IR with high-level type information. These information are used later for an alias analysis that is implicitly executed with the FTA.

-Xclang=-disable-llvm-optzns Suppresses the transformation passes that are included in -O1. This ensures that the generated IR is a direct translation of the source code. Otherwise, Opt may for instance delete instructions that could be relevant for the FTA.

-g Demands debug information that include the high-level source code location of every instruction in the IR. So, every instruction gets assigned the code file and line, it corresponds to. These locations are utilised later during the report generation.

-fvara-feature Detects and annotates the feature variables in the generated IR. The features and their locations in the source code are extracted from the given feature model.

-fvara-fm-path=<path> Specifies the path of the feature model of the current project. In the experiment, the path is automatically determined prior to the specification of the Clang options.

After generating the IR with CLANG, the PTA-experiment calls OPT to optimise and analyse the code. We specify the following options for the execution of OPT:

-enable-new-pm=0 Disables the new pass manager of LLVM. Without this, we would need to hand over the optimisation passes in another format which VARA currently does not support.

-S Tells VARA to use the human-readable LLVM assembly language for the instructions that are printed in the output file.

-vara-FAR Runs the FTA and generates a feature analysis report (FAR) as output. The FAR is subject of Section 3.2.

-vara-report-outfile=<path> Specifies the path of the report file.

3.2 FEATURE ANALYSIS REPORT

The output of the PTA-experiment is a YAML file containing a feature analysis report (FAR). The report consists of a header, metadata, and a body. The header just contains the report type and the version number of the report type. The metadata are general information about the analysed LLVM IR module. Here, the number of functions[1], instructions, and BR/SWITCH instructions in the module is stored. The body is the main part of the report. It contains the actual results of the FTA by mapping every function of the module to their demangled name and their feature-related instructions. Each instruction entry in the report comprises the tainted instruction, along with the location in the high-level source code it was generated from, as well as a list of the features it is related to.

Let us take a look at a FAR by running our experiment on the project FEATUREPERFCSCOLLECTION[2] which is shipped with VARA-TS. We create a paper config named EXAMPLE, add a case study with the latest revision of the FEATUREPERFCSCOLLECTION, and run the PTA-experiment on it:

```
$ vara-pc create example
$ vara-cs gen -p FeaturePerfCSCollection select_latest
$ vara-run -E PhASARFeatureTaintAnalysis
```

The FEATUREPERFCSCOLLECTION is a collection of small C++ code examples, including the one shown in Listing 3.1 named SINGLELOCALSIMPLE. In this program, the variable `Slow` is the feature variable of the feature *Slow*. In Listing 3.2, we can see an excerpt from the FAR of SINGLELOCALSIMPLE which we get as output of our experiment. We can retrieve from its metadata that there are 72 functions in the IR module. The only function with feature taints is `main`. Since `Slow` is the only feature variable in the module, all 4 feature taints are related to *Slow*. The first taint is the memory allocation of the variable and does not directly result from any statement in the C++ file. The other taints are the translation of the IF statement condition in Line 14 of the source file.

1 We do not need to know what functions in LLVM IR exactly are. We only use them to structure the body of the FAR.

2 `https://github.com/se-sic/FeaturePerfCSCollection` (visited on 03/07/2023)

```cpp
#include "fpcsc/perf_util/sleep.h"

#include <string>

int main(int argc, char *argv[]) {

  bool Slow = false;

  if ((argc >= 2 && argv[1] == std::string("--slow")) ||
      (argc >= 3 && argv[2] == std::string("--slow"))) {
    Slow = true;
  }

  if (Slow) {
    fpcsc::sleep_for_secs(5);
  } else {
    fpcsc::sleep_for_secs(2);
  }

  return 0;
}
```

Listing 3.1: SingleLocalSimple

The FAR offers a useful representation of LLVM IR for our purposes. It restricts the IR code to the feature taints and maps every instruction to their related features, as well as to their corresponding location in the high-level source code. We utilise these information for our evaluation of the FTA as we describe in the next chapter.

```
---
DocType:          FeatureAnalysisReport
Version:          1
...
---
funcs-in-module: 72
insts-in-module: 1011
br-switch-insts-in-module: 83
...
---
result-map:
  _GLOBAL__sub_I_SLSmain.cpp:
    demangled-name:  _GLOBAL__sub_I_SLSmain.cpp
    feature-related-insts: []
  . . .
  __cxx_global_var_init:
    demangled-name:  __cxx_global_var_init
    feature-related-insts: []
  main:
    demangled-name:  main
    feature-related-insts:
      - inst:     '%Slow = alloca i8, align 1, !FVar !1356'
        location: None
        taints:
          - Slow
      - inst:     '%32 = load i8, i8* %Slow, align 1, !dbg !1563, !tbaa !1388, !range !1565'
        location: 'src/SingleLocalSimple/SLSmain.cpp:14'
        taints:
          - Slow
      - inst:     '%tobool = trunc i8 %32 to i1, !dbg !1563'
        location: 'src/SingleLocalSimple/SLSmain.cpp:14'
        taints:
          - Slow
      - inst:     'br i1 %tobool, label %if.then71, label %if.else, !dbg !1566'
        location: 'src/SingleLocalSimple/SLSmain.cpp:14'
        taints:
          - Slow
...
```

Listing 3.2: Excerpt from feature analysis report (FAR) of Listing 3.1

4 OPERATIONALISATION: MEASURING ANALYSIS QUALITY

For every feature taint in the FAR, we need to decide if it should be found by the FTA or not. In addition, there may be taints that are not included in the report but actually should be. So, for every real-world project that we consider in our evaluation, we need to define a ground truth that we can compare with the FAR to collect data about the precision of the FTA. In this chapter, we describe how we find this ground truth and show the data that we get, when we compare it to the FAR. Based on this data, we may need to adjust our initial expectations by revising the ground truth. In the end of the chapter, we present the projects that are subject of our research.

4.1 GROUND TRUTH

After generating an FAR for a project with the PTA-experiment, we need to decide which taints are correct, wrong or missing. The easiest way to do so, is by defining a reference, a ground truth, that contains all the expected feature taints. The ground truth should be conveniently comparable with the FAR to collect data that quantifies the precision of the FTA. In this section, we explain what feature taints we expect the FTA to find and how we proceed to get the ground truth for a project.

In Section 2.4.1.2, we saw that the FTA is part of the feature-region detection pipeline. It is important that the FTA works correct in especially this context, since it is the main task of the analysis to serve as an intermediate step of the feature-region detection. The feature-region detection determines the feature-regions as the branching options of tainted BR and SWITCH instructions. This means if the FTA does not taint these two types of instructions correctly, the feature-region detection will not be able to find the correct feature-regions. Therefore, we will constraint our evaluation on BR and SWITCH instructions, rather than every single instruction of an IR module. With this constraint, we can focus on the feature taints that really matter and do not need to deal with unnecessary details.

So, we could generate the IR of a project and decide for each BR/SWITCH instruction individually if it should be tainted or not, in order to get the ground truth of the project. This would require us to understand the whole IR code and is pretty unreasonable and error prone, as the IR module of a real-world software project is usually very huge and hard to read. Luckily, the FAR maps every feature-related instruction to the location in the high-level code that the instruction was derived from. We exploit this information and relocate our investigations to the high-level code of a project, instead of analysing the IR code of a project by hand. We use an integrated development environment (IDE) for convenient access to the source code.

When we open the project code in an IDE, we need to identify the feature variables. Their locations are specified in the feature model that is used by the PTA-experiment. Next, we trace the data-flow of the feature variables by hand, one after another. With the supporting features provided by the IDE, we look up every direct access to a feature variable and search

```c
 1 int main(int argc, char *argv[]) {
 2
 3   int Foo, Bar; // Feature Variables
 4
 5   if (Foo) {
 6     Foo--;
 7   } else {
 8     Foo++;
 9   }
10
11   if (Bar) {
12     Bar--;
13   }
14
15   if (Foo && Bar) {
16     return 1;
17   }
18   return 0;
19 }
```

Listing 4.1: gt_basic.c

```yaml
Bar:
  - 'gt_basic.c:11'
  - 'gt_basic.c:15'
Foo:
  - 'gt_basic.c:5'
  - 'gt_basic.c:15'
```

Listing 4.2: Ground truth of gt_basic.c

for conditional control flow decisions dependent on the variable. In other words, we look for the code locations where the feature variable is directly accessed insight the condition of the ternary operator, IF-, WHILE-, FOR-, or SWITCH statements. The compiler probably realises these expressions with BR or SWITCH instructions. We collect all these locations for every feature of interest of a project. This collection then contains all the locations of the high-level code where we expect a tainted BR/SWITCH in the IR. This is the ground truth[1] of the project.

Let's have a look at Listing 4.1 to get an idea of this process in practice. `Foo` and `Bar` are the feature variables of the features *Foo* and *Bar*, respectively. We trace the data-flow of both variables and search for conditional control flow decisions dependent on them to get the ground truth of the program. Starting with `Bar`, we see that the IF statements in Line 11 and 15 use the variable in their condition. Therefore, we expect that feature *Bar* taints BR instructions at these locations and put them in the ground truth. Now, we trace the other feature variable `Foo` and identify that it is directly accessed insight the IF conditions in Line 5 and 15. Hence, we put these locations in our ground truth. This gives us the ground truth as presented in Listing 4.2.

We also need to consider that the FTA is an interprocedural analysis. Accordingly, we have to trace feature variables across function calls. If a feature variable is passed to a function via a parameter of a function call, we trace the data-flow of the respective parameter insight the function. For example in Listing 4.3, the feature variable `Foo` is passed to the function `helper` in Line 13. Therefore, we need to trace the flow of the tainted data insight the function. Since `Foo` is passed as the parameter `a`, we search the body of `helper` for conditional control flow decisions dependent on `a`. We identify Line 2 as the only qualifying location and put it in the ground truth. Additionally, in Line 3, `a` is directly accessed in one of the RETURN

1 Strictly speaking, it is an approximated ground truth of the project. In Section 4.3, we talk about this in further detail. For simplicity, we just refer to our approximated ground truth as *ground truth*.

```
1  int helper(int a) {
2    if (a) {
3      return a++;
4    }
5    return 0;
6  }
7
8
9  int main(int argc, char *argv[]) {
10
11   int Foo;      // Feature Variable
12
13   if (helper(Foo)) {
14     return 42;
15   }
16   return 0;
17 }
```

Listing 4.3: gt_func_call.c

```
Foo:
  - 'gt_func_call.c:2'
  - 'gt_func_call.c:13'
```

Listing 4.4: Ground truth of gt_func_call.c

		Analysis Result	
		Positive	Negative
Ground Truth	Positive	True Positive	False Negative
	Negative	False Positive	True Negative

Table 4.1: Types of collected data

statements of `helper`. Therefore, we expect the function to return tainted data. The function is called inside the IF condition in Line 13. Hence, we also expect a tainted BR instruction at this location and get the ground truth as shown in Listing 4.4.

4.2 COLLECTING EVALUATION DATA

Now, we want to collect actual data about the quality of the FTA based on the ground truth and the FAR of a project. The FAR is the true analysis result and the ground truth is the expected analysis result. So, we can find unexpected behaviour of the FTA by comparing both and looking for differences in the locations of feature taints.

For this purpose, we implement the Python class `FeatureAnalysisReportEval` in VaRA-TS. Given the FAR and the ground truth of a project, this class contrasts both inputs with each other. For every location in the ground truth, it searches for a corresponding entry of a BR/SWITCH instruction of the same feature in the FAR. Additionally, it checks if there are any tainted BR/SWITCH instructions in the report whose locations are not part of the ground truth.

With this process, the class computes the four types of data that are depicted in Table 4.1. We have a true positive in case a location of a feature is part of the ground truth and there is a feature taint of the respective feature at a BR/SWITCH instruction with the same location

Feature	True Positive	True Negative	False Positive	False Negative
Bar	2	30	1	0
Foo	2	30	0	1
Total	**4**	**60**	**1**	**1**

Table 4.2: Exemplary evaluation results

in the FAR. If a BR/SWITCH location is neither part of the ground truth, nor in the report, we have a true negative. In these two cases, the FTA works as expected and there is no need for further investigations.

More interesting are the situations in which ground truth and FAR differ from one another. If a location is part of the ground truth but not in the FAR, we have a false negative. This means that the FTA does not taint an instruction which is expected to be tainted. The other way around, we speak of a false positive if an instruction is not in the ground truth but in the report. In this case, the FTA finds a taint where no taint is expected. So, false negatives and false positives reflect unexpected behaviour of the analysis and need to be further investigated.

Table 4.2 shows an exemplary evaluation result of the made up FAR in Listing 4.5 and the ground truth in Listing 4.6. The ground truth contains expected taints of the two features *Bar* and *Foo*. For *Bar*, there are tainted BR instructions at both locations of the ground truth in the FAR in Line 2 and 30, respectively. Hence, we have 2 true positives. However, in Line 6 is an entry of a BR instructions whose location is not in the ground truth. Therefore, we also have a false positive for the feature. All other BR/SWITCH instructions in the IR are not tainted and not expected to be tainted. This way, we get the 30 true negatives. For the other feature *Foo*, we have two matching taints at the first location that is specified in the ground truth, with their entries in Line 18 and 30 of the FAR. So, we again have two true positives for the feature. But as their is also a location specified in the ground truth of which no corresponding entry can be found in the FAR, we also have one false negative for feature *Foo*.

This example exposes an interesting detail of our evaluation process. Namely, that we do not distinguish between instructions at the same location. A location in the ground truth means that we expect at least one tainted BR/SWITCH instruction at this location but it is not specified how this instruction looks like. For instance, in Listing 4.5, two BR instructions with the same location are tainted by the feature *Foo*. Both are considered true positive as the respective location is specified in the ground truth. However, it may be the case that actually only one of the instructions should be tainted. This leaves space for inaccuracies in our evaluation. Nonetheless, verifying that really the correct instructions are tainted would produce a lot of overhead. For every location in the ground truth, we would need to check in the IR if there are multiple BR/SWITCH instructions with this location. And in this case, we would need to decide for each of these instructions individually if the taint is correct or not. This requires us to comprehend the IR and backtrack the data-flow by hand. We consider this overhead as inappropriate compared to the expected inaccuracies of our current approach. Therefore, we accept this inaccuracy of our evaluation process.

```
...
- inst: 'br i1 %tobool, label %if.then42, label %if.else, !dbg !1524'
  location: 'example.c:24'
  taints:
    - Bar
- inst: 'br i1 %tobool, label %if.then49, label %if.else, !dbg !1531'
  location: 'example.c:31'
  taints:
    - Bar
- inst: '%32 = load i8, i8* %Foo, align 1, !dbg !1562, !tbaa !1388, !range !1562'
  location: 'example.c:42'
  taints:
    - Foo
- inst: '%tobool = trunc i8 %32 to i1, !dbg !1563'
  location: 'example.c:42'
  taints:
    - Foo
- inst: 'br i1 %tobool, label %if.then71, label %if.else, !dbg !1566'
  location: 'example.c:42'
  taints:
    - Foo
- inst: '%33 = load i8, i8* %Bar, align 1, !dbg !1568, !tbaa !1388, !range !1568'
  location: 'example.c:42'
  taints:
    - Bar
- inst: '%tobool = trunc i8 %33 to i1, !dbg !1569'
  location: 'example.c:42'
  taints:
    - Bar
- inst: 'br i1 %tobool, label %if.then72, label %if.else, !dbg !1570'
  location: 'example.c:42'
  taints:
    - Foo
    - Bar
...
```

Listing 4.5: Exemplary feature analysis report (FAR) excerpt. Entries of BR instructions are highlighted red.

```
Bar:
  - 'example.c:24'
  - 'example.c:42'
Foo:
  - 'example.c:42'
  - 'example.c:66'
```

Listing 4.6: Ground truth of Listing 4.5

After collecting the mentioned evaluation data, we use the following formulas to draw conclusions about the overall accuracy of VaRA's FTA:

$$precision = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$sensitivity = \frac{TP}{TP + FN} \tag{4.2}$$

$$specificity = \frac{TN}{TN + FP} \tag{4.3}$$

The precision (4.1) reflects the proportion of correct analysis results to correct and incorrect analysis results. If the FTA behaves completely as expected, i.e, there are no false positives and no false negatives, the precision is 100%. The precision decreases as the unexpected behaviour increases. The sensitivity (4.2), also known as the true positive rate, expresses how certain we can be to have a feature taint at a location where we expect one. It is 100% in case the FTA taints a BR/SWITCH instruction at every location in the ground truth, i.e., there are no false negatives. The specificity (4.3), also known as the true negative rate, is the percentage of instructions that are not tainted among the instructions that are not expected to be tainted. A specificity of 100% means that the FTA does not unexpectedly taint any BR/SWITCH instruction, i.e., there are no false positives.

In this section, we saw that true positives and true negatives reflect expected behaviour of the FTA, while false positives and false negatives reflect unexpected behaviour. However, it is important to note that *unexpected* does not necessarily mean *wrong*, as we explain in the next section.

4.3 FURTHER GROUND TRUTH APPROXIMATION

We mentioned before that the ground truth is just an approximation. This is because we encounter some problems when we create and evaluate a ground truth. We probably need to revise an initial ground truth and approximate it further.

So far, we only consider direct accesses to feature variables in the ground truth. However, VaRA's FTA should also be able to identify indirect accesses. Therefore, we also need to consider them in order to get the real ground truth of a project. Nevertheless, we decided that we do not expect indirect accesses to be found in the first place. We came up with this decision as the data-flow of real world software is usually very complex. We may expect taints at wrong locations or oversee them. Accordingly, the risk of putting wrong locations in the ground truth is high. Hence, identifying every indirect access to a feature variable by hand is very error prone and leaves a high risk of getting wrong evaluation results.

Nevertheless, we need to respect indirect accesses to feature variables. Therefore, we say that the FTA can find them but does not necessarily has to. Our goal is to underapproximate the ground truth. Initially, we only put direct accesses in the ground truth. Then, we use this ground truth and the FAR of the respective project to generate evaluation data. In case

```
1 int main(int argc, char *argv[]) {
2
3   int Foo;      // Feature Variable
4
5   int local = Foo ? Foo++ : Foo--;
6
7   return Foo || local;
8 }
```

Listing 4.7: Uncertain LLVM instructions. The ternary operator in Line 5 is not translated with a BR instruction. The logical-OR operator in Line 7 is translated with a BR instruction.

the FTA finds any indirect accesses, they would be part of the false positives. So, we identify the locations of false positives in the source code and add all indirect accesses to the ground truth. This way, we approximate the ground truth further by adding the taints that could be found by the FTA.

Indirect accesses are not the only situations in which we may have to revise the ground truth subsequently. Remember that we infer from the high-level code to the IR code when we create the ground truth. For every location in the ground truth, we expect that the respective high-level code is translated with a BR or SWITCH instruction. Yet, it could happen that there are BR/SWITCH instructions where we would not expect them intuitively, or, the other way around, that there are no BR/SWITCH instructions where we would expect them.

For instance, in Listing 4.7, the only tainted BR/SWITCH instruction we would expect is in Line 5 because the feature variable is used inside the condition of the ternary operator. However, the only tainted BR/SWITCH instruction that the FTA finds in the program is in Line 7. So, our evaluation generates one false positive and one false negative. Nevertheless, we see that the FTA actually works perfectly fine when we look in the IR of the program. The logical-OR operator in Line 7 is realised with a BR and the instruction should be tainted as the operator uses the feature variable. On the other hand, the ternary operator in Line 5 is translated with a SELECT[2] instruction into IR. This instruction does not produce control flow and is therefore not interesting for our evaluation. Accordingly, we have to remove Line 5 and add Line 7 to the ground truth.

This means in general, that we look up every finding, i.e., every false positive and false negative, that we generate with our initial ground truth. Then, we decide individually if the outcome of the FTA makes sense in the specific situation and, if necessary, revise the ground truth to approximate it further.

4.4 CONSIDERED PROJECTS

We use real-world C projects of medium size for the evaluation. Overall, we collect data for 39 features from 4 projects. The revised ground truths of these 4 projects contain 290 expected taints in total. Table 4.3 provides an overview of all considered projects.

2 https://llvm.org/docs/LangRef.html#select-instruction (visited on 03/07/2023)
3 https://github.com/libarchive/bzip2.git (visited on 03/07/2023)
4 https://github.com/vulder/gzip.git (visited on 03/07/2023)
5 https://github.com/bnico99/picoSAT-mirror (visited on 03/07/2023)
6 https://github.com/xz-mirror/xz.git (visited on 03/07/2023)

Project	Domain	#Features	#Expected-Taints	Commit
bzip2[3]	Compression	8	89	1ea1ac1
gzip[4]	Compression	8	48	23a870d
PicoSAT[5]	Solver	12	55	33c685e
xz[6]	Compression	11	98	4773608

Table 4.3: Considered projects

Note that we dropped 7 features of GZIP because VaRA did not annotate the respective feature variables in the generated IR. For these features, the FTA is not able to analyse the feature taints correctly. The missing feature variable annotations would result in false negatives for every entry of the affected features in the ground truth. However, these false negatives do not reflect wrong behaviour of the FTA. Therefore, we would produce incorrect results if we consider these features in the evaluation.

Initially, we also wanted to collect data for C++ projects. However, VaRA currently has problems with annotating feature variables that are implemented as STRUCT members. We tried to include projects with this type of feature variables in the evaluation, but had to drop them, eventually, since the annotation of the variables did not work. Especially in object-oriented programming languages like C++, it is a common pattern to implement feature variables as STRUCT members. So, we decided to not include C++ projects in general.

5 EVALUATION

We created ground truths for the subject projects and generated the corresponding FARs with the PTA-experiment in order to collect evaluation data. In this chapter, we present and discuss the evaluation results, as well as the threats to validity of the evaluation.

5.1 RESULTS

Table 5.1 shows the results of the evaluation. The corresponding precision, sensitivity, and specificity are listed in Table 5.2. Our results are based on VaRA[1] commit cbc6c8f and vara-llvm-project[2] commit 84c3e84. A detailed overview of the evaluation results of all considered projects can be found in Appendix A.

VaRA's FTA found 304 true positives and 1 false negative. This gives us a total sensitivity of 99.67%. The only false negative appears in xz, all other projects have a perfect sensitivity of 100%. Additionally, we found 1044 false positives and measured a total specificity of 99.16%. Most of the false positives appear in gzip, which has a specificity of 95.30%. The projects bzip2 and PicoSAT have a nearly perfect specificity of more than 99.90%, resulting from their few false positives. xz has a perfect specificity of 100%. The overall precision that we measured is 99.16%. By far the worst precision, through its relatively high amount of false positives, has gzip with 95.31%.

Table 5.3 provides a summary of all features in the evaluation with unexpected behaviour. The one false negative in our results is related to the feature *Mode* of xz. In Listing 5.1, we see the function of xz where this false negative occurs. The problem arises in Line 4 when the parameter `mode` is accessed. There are several calls of the function specified in the source code of xz. In two of these calls, the feature variable `opt_mode` is passed to the function. Therefore, the access to the function parameter `mode` in Line 4 should be tainted. However, the FTA does not find this taint.

Besides this single false negative, we found many false positives. The false positives of PicoSAT's *NoPrintAssignment* arise due to the fact that VaRA does annotate an additional feature variable in the IR that is not specified in the feature model. The FTA traces the data-flow of the wrong variable correctly and therefore, the false positives occur. We decided to keep this feature in the evaluation, since the actual feature variable is correctly annotated. This finding may serve as a starting point for future improvements of VaRA's feature variable detection.

All other false negatives occur due to wrong behaviour of the FTA. At some point in the program, the analysis treats variables as indirect feature variable accesses although the respective feature variable does neither explicitly, nor implicitly flow to the variable. This causes potentially huge overapproximations of the feature taints of single features. In Listing 5.2, we see a code excerpt of PicoSAT where this behaviour occurs. In Line 10,

1 `https://github.com/se-sic/VaRA` (visited on 03/07/2023)
2 `https://github.com/se-sic/vara-llvm-project` (visited on 03/07/2023)

Project	True Positive	True Negative	False Positive	False Negative
bzip2	97	25568	23	0
gzip	53	19634	969	0
PicoSAT	55	61297	52	0
xz	99	17049	0	1
Total	304	123548	1044	1

Table 5.1: Evaluation results

Project	Precision	Sensitivity	Specificity
bzip2	99.91%	100%	99.91%
gzip	95.31%	100%	95.30%
PicoSAT	99.92%	100%	99.92%
xz	99.99%	99%	100%
Total	99.16%	99.67%	99.16%

Table 5.2: Precision, sensitivity, specificity

the accesses to `err` and `i` in the FOR loop condition appear to be tainted by the feature *Verbose*. However, the feature variable `verbose` flows at no point to one of these variables. Alongside other variables that are tainted like this, the variables propagate feature taints of *Verbose* through the program. This way, the 50 false positives come about. The same behaviour occurs for BZIP2 and GZIP. For GZIP however, it is on a much wider scale as it effects 3 features and variables are tainted across the whole IR module, not just in a single function. Therefore, the total amount of false positives in GZIP is bigger compared to BZIP2 and PicoSAT. It is also worth to mention that the random feature taints of *no_name* are a subset of the random feature taints of *decompress*, which are again a subset of the random feature taints of *force*. This means that the FTA starts by randomly identifying taints of *force*. As the control flow of the program proceeds, the random taints first become taints of *force* and *decompress*, and later of *force*, *decompress*, and *no_name*.

We failed to extract and reproduce the unexpected behaviour from above in the form of minimal examples. However, while we tried to reproduce these bugs, we found some other problems that can not be mapped to the findings in Table 5.3. One of these problems is exposed by the code fragment in Listing 5.3. Here, the feature variable `Foo` is directly accessed in the RETURN statement of the function `get_foo` in Line 4. As expected, the FTA correctly identifies the RETURN value of the function as tainted. In Line 11, this function is now called by the `main` function inside an IF condition. Accordingly, the analysis should also taint the corresponding BR instruction. However, the FTA does not do this since the returned data of the function call is not identified as a feature taint. Interestingly enough, the taint gets identified as soon as the analysis finds another feature taint in the `main` function. In Listing 5.4, the feature variable BAR causes feature taints inside `main`. In this example, the FTA taints the BR instruction in Line 11 correctly.

```c
extern uint64_t
hardware_memlimit_get(enum operation_mode mode)
{
  const uint64_t memlimit = mode == MODE_COMPRESS
    ? memlimit_compress : memlimit_decompress;
  return memlimit != 0 ? memlimit : UINT64_MAX;
}
```

Listing 5.1: Function of xz with false negative for the feature *Mode*. The false negative is highlighted red.

```c
int
picosat_main (int argc, char **argv)
{
  int res, done, err, print_satisfying_assignment, force, print_formula;
  ...
  int i, decision_limit;
  ...
  done = err = 0;
  ...
  for (i = 1; !done && !err && i < argc; i++) {...}
}
```

Listing 5.2: Excerpt of PicoSAT with false positives for the feature *Verbose*. The false positives are highlighted red.

```c
 1 int Foo;         // Feature Variable
 2
 3 int get_foo() {
 4   return Foo;
 5 }
 6
 7 int main(int argc, char *argv[]) {
 8
 9   int Bar;       // No Feature Variable
10
11   if (get_foo()) {
12     return 42;
13   }
14
15   return 0;
16 }
```

Listing 5.3: The function call of `get_foo` in Line 11 does not return tainted data. The only feature taint of *Foo* appears in Line 4.

```c
 1 int Foo;         // Feature Variable
 2
 3 int get_foo() {
 4   return Foo;
 5 }
 6
 7 int main(int argc, char *argv[]) {
 8
 9   int Bar;       // Feature Variable
10
11   if (get_foo()) {
12     return 42;
13   }
14
15   return 0;
16 }
```

Listing 5.4: The function call of `get_foo` in Line 11 returns tainted data. Feature taints of *Foo* appear in Line 4 and 11.

Project	Feature	TP	TN	FP	FN
bzip2	verbosity	17	3171	23	0
gzip	decompress	19	2238	325	0
	force	9	2234	339	0
	no_name	2	2275	305	0
PicoSAT	NoPrintAssignment	3	5112	2	0
	Verbose	22	5045	50	0
xz	Mode	24	1534	0	1

Table 5.3: Evaluation findings

```
 1 int main(int argc, char *argv[]) {
 2
 3   int i;         // Feature Variable
 4   int arr[3];    // No Feature Variable
 5
 6   if (arr[i]) {
 7     return 42;
 8   }
 9
10   return 0;
11 }
```

Listing 5.5: The array access in Line 6 returns tainted data. The corresponding BR instruction is tainted by *i*.

In Listing 5.5, we can see some other problematic corner case. The feature variable `i` is used to access the array `arr` in the IF condition in Line 5. The analysis now identifies the accessed data of the array as a feature taint of *i*, and, consequently, taints the BR instruction. However, this behaviour is arguable as the feature variable does not explicitly flow to the accessed data. The index `i` is only used for address calculation. The accessed data is then determined based on the calculated address, but the feature variable itself is not explicitly assigned to the value that is accessed. Intuitively, this pattern matches more to an implicit flow, but formally, this is also not the case. Per definition, we would need some kind of control flow decision to be involved in the process of the array access in order to have an implicit flow. Because of its ambiguity, this edge case deserves attention when it comes to discussing future improvements of VaRA's FTA.

Listing 5.6 and the corresponding feature model in Figure 5.1 present a pattern of feature implementation that we faced several times during research. Here, the variable `Foo` is the feature variable of the feature *Foo*, which has the sub-features *A*, *B*, and *C*. These features are not implemented with the help of feature variables, but with the help of an ENUM. Each of the sub-features can be mapped to a different value of the enumeration type `e`. Accordingly, VaRA's FTA should be able to identify feature taints of *A* and *B* in Line 7. However, this

```c
enum e { A, B, C };

int main(int argc, char *argv[]) {

  enum e Foo;   // Feature Variable

  if (Foo == A || Foo == B) {
    return 1;
  }

  return 0;
}
```

Listing 5.6: Features as ENUM values

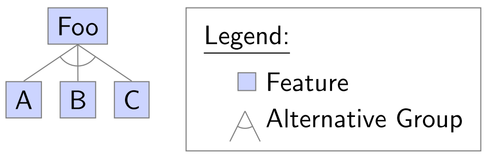


Figure 5.1: Feature model for Listing 5.6

pattern of feature implementation is currently not supported by VaRA and may be an interesting extension in the future.

5.2 DISCUSSION

Based on the evaluation data that we collected, we are now able to answer the central and eponymous research question of this thesis:

> How precise can VaRA track the influence of feature variables in real-world programs?

The precision that we measured for VaRA's FTA is 99.16%. The total sensitivity is 99.67%, the total specificity is 99.16%. Note that the precision is way more influenced by true negatives and false positives than by true positives and false negatives. This is because the number of expected taints is usually way smaller than the number of instructions that should not be tainted. Therefore, it is especially important to separate the sensitivity from the precision.

The precision expresses that we can be 99.16% sure that the FTA taints a conditional control flow instruction correctly. This means that the analysis works pretty accurate in the pipeline of VaRA's feature-region detection. With a probability of 99.16%, the FTA provides the feature-region detection with correct information.

However, there also are inaccuracies. We never encountered false positives and false negatives for the same feature, i.e., in case of unexpected behaviour, the FTA either overapproximates, or underapproximates the feature taints. The specificity is lower than the sensitivity. Hence, we can conclude that the FTA tends to overapproximate the feature taints. These overapproximations rather occur for single features than being scattered across all features of a project.

But still, there are situations in which the analysis overlooks feature taints. If the FTA does not find feature taints, the feature-region detection is not able to fully determine the feature-regions. With an overapproximation of feature taints, it is at least guaranteed that the feature-region detection can find all feature-regions. Therefore, false negatives are to a certain extend more problematic than false positives. We have 1 false negative in our results, i.e., the FTA found 289 of 290 expected feature taints. Note that the number of true positives is 304, and thus bigger than 289. This is due to the fact that we do not specify the number of expected taints at a location in the ground truth. In case there is more than 1 tainted BR/SWITCH instruction at a location, all of those are counted as true positive.

It is also worth to mention again that we dropped 7 features of GZIP from our evaluation. We discussed this in Section 4.4. VaRA did not annotate any feature variables for those features. Respectively, the FTA was not able to analyse the feature taints of the features, resulting in a high amount of false negatives. Therefore, the sensitivity of our results would have been significantly lower. However, the missing feature taints were induced by the feature variable annotation and not by the FTA. Therefore, we decided to drop the features as their evaluation results do not reflect wrong behaviour of the analysis.

5.3 THREATS TO VALIDITY

A threat to the internal validity of our results is that we trace the data-flow of the subject projects by hand in order to get the ground truths. We reduce the risk of errors to a minimum by analysing the high-level code, rather than the IR, and by restricting our expectations to direct accesses to feature variables that are translated with BR/SWITCH instructions in the IR. Yet, there is still room for human errors. For instance, we may have overseen a BR instruction that is not obvious from the high-level code perspective. In case that the instruction actually qualifies for the ground truth and the FTA also did not find it, we would consider it as a true negative even if it should be a false negative.

It is also a threat to the internal validity of our research that we do not distinguish between instructions at the same location. We only specify that we expect at least one tainted BR or SWITCH instruction when we put a location in the ground truth. If the high-level source code at this location is translated into IR with multiple BR/SWITCH instructions, we do not specify which of these instructions actually should be found by the FTA. This leaves room for inaccuracies of the analysis that we are not able to identify with our evaluation approach.

The focus on direct accesses increases the internal validity of our results, but at the same time threatens their external validity. This is because the FTA is also supposed to find indirect accesses. However, these are not reflected by the results that we gathered. We underapproximated the ground truth of the analysis by declaring indirect accesses as optional.

The restriction to BR and SWITCH instructions of C projects is another threat to the external validity of our results. The precision that we measured for VaRA's FTA is only valid in the context of the feature-region detection for C software. We can not say how good the analysis performs as a whole, i.e., with all feature taints under consideration. We are also not able to draw conclusions about the precision for C++ projects. The behaviour of the FTA may be different for these projects due to the complex language features of C++.

6 RELATED WORK

In 2017, Sattler [22] introduced the variability-aware region analyser VaRA, an open-source analysis framework that provides developers an interface for all kinds of code region detections in LLVM IR. In his work, Sattler showcased his framework by implementing the first revision of VaRA's static feature-region detection. Timm [24] extended the feature-region detection later on with PhASAR support in order to implement the FTA into VaRA. The open-source static analysis framework PhASAR provides convenient access to implementations for solving data-flow problems in LLVM IR [23].

Other research also investigated the data-flow problem of tracking configuration options and feature variables in configurable systems. Both; static [10, 26] and dynamic [21, 27] analyses; were presented to achieve this goal. For instance, Velez et al. [26] introduced ConfigCrusher, a tool that analyses the performance of feature-regions in order to build so-called performance-influence models. The feature-regions are determined with a static data-flow analysis which also considers implicit flows. The static analysis, however, does not scale beyond small systems. Therefore, Velez et al. [27] developed Comprex. This tool builds performance-influence models with an iterative dynamic taint analysis. Comprex maps features to regions by repeatedly executing a system in different configurations and analysing the feature taints. This approach also considers implicit flows and, in contrast to ConfigCrusher, scales to large systems.

In general, static analyses are a broad research field. For instance, they are examined in the context of identifying software vulnerabilities [14, 18, 19, 25], or coding errors [5, 11, 12]. Another example was presented by Haas et al. [13]. They investigated if static analyses can be used in order to detect unnecessary source code. VaRA's FTA adapts the concept of static taint analysis. These are typically used to identify security leaks in software [14, 18, 25].

In their recent work, Nachtigall, Schlichtig, and Bodden [20] evaluated the usability of static analysis tools. They evaluated 46 mainly non-proprietary static analysis tools for different languages against 36 usability criteria. Their research exposes huge chances of VaRA as an open-source framework providing static analyses of C/C++ source code. A total of 69 tools for C/C++ code were considered, but only 5 could be included in the evaluation. The other 64 were excluded for different reasons, e.g. because they were proprietary or could not be installed. The 5 tools that were tested on the usability criteria did not perform very well. For instance, only one tool provides some kind of fix support, and warning messages are, in general, pretty poor. This means that the usability of open-source static analysis tools for C/C++ is usually neglected. VaRA has good chances to stand out from other tools if it provides a good usability.

7 CONCLUDING REMARKS

In this chapter, we summarise and conclude our work. We also provide an outlook for future work at the FTA, the feature-region detection, and VaRA in general.

7.1 CONCLUSION

In this thesis, we evaluated VaRA's FTA on real-world programs. For 4 C projects, we underapproximated the analysis' ground truth by tracing the data-flow of feature variables by hand. We determined conditional control flow decisions dependent on the feature variables with the aim to ensure that the analysis finds the information that are important in the feature-region detection pipeline.

We implemented an experiment in VaRA-TS that automatically configures and runs the FTA. For every subject project, we compared the output of the experiment with the created ground truth. We encountered unexpected behaviour of the analysis for some features, and failed to extract and reproduce these bugs. However, while we tried to extract these findings, we found some other problematic corner cases that deserve further investigation in the future.

Overall, we measured a high precision of the FTA. According to our results, the analysis provides the feature-region detection with correct information with a probability of 99.16%. Inaccuracies either occur as overapproximations, or as underapproximations of the feature taints, whereat the case of an overapproximation is more likely.

7.2 FUTURE WORK

The problematic corner cases that we mined can be utilised to improve VaRA's FTA in the future. The respective code fragments expose analysis bugs that need to be fixed, or at least deserve special attention. The same holds for the unexpected behaviour of the real-world projects that we were unable to reproduce in the form of minimal examples. For the affected features, the behaviour of the analysis can be further analysed in the future for the purpose of determining shortcomings.

We only considered C projects in our research. However, VaRA also supports C++ projects. A detailed evaluation of the FTA on C++ real-world software is currently not feasible due to problems with the feature variable annotation in case that feature variables are implemented as STRUCT members. These problems are well known and currently worked on. But as soon as they are fixed, an evaluation of VaRA's FTA on C++ projects would be appropriate. The more complex language features of C++ are likely to uncover additional shortcomings of the analysis.

Besides the known bugs, we also encountered unknown problems of the feature variable annotation. These need to get fixed in order to ensure that the FTA can work correctly. The respective features of GZIP and PICOSAT can be analysed to identify the bugs, and to further improve VARA in the future.

The FTA is an intermediate part of VARA's feature-region detection. In a next step, it needs to be assured that the feature-regions are correctly detected. For the features without any false positives and false negatives in our evaluation results, we can be sure that the detection pipeline works correct to the point when it comes to detecting the feature-regions. Hence, these features serve as a good starting point for a future evaluation of the feature-region detection.

A APPENDIX

Feature	True Positive	True Negative	False Positive	False Negative
blockSize100k	5 (5)	3206 (3206)	0 (0)	0 (0)
forceOverwrite	7 (7)	3204 (3204)	0 (0)	0 (0)
keepInputFiles	2 (2)	3209 (3209)	0 (0)	0 (0)
opMode	8 (8)	3203 (3203)	0 (0)	0 (0)
quiet	15 (15)	3196 (3196)	0 (0)	0 (0)
smallMode	5 (5)	3206 (3206)	0 (0)	0 (0)
srcMode	38 (38)	3173 (3173)	0 (0)	0 (0)
verbosity	17 (17)	3171 (3171)	23 (23)	0 (0)
Total	97 (97)	25568 (25568)	23 (23)	0 (0)

Table A.1: Evaluation results for project BZIP2. The brackets contain the results from the initial and unrevised ground truth.

Feature	True Positive	True Negative	False Positive	False Negative
decompress	19 (19)	2238 (2233)	325 (325)	0 (5)
do_lzw	1 (1)	2581 (2581)	0 (0)	0 (0)
force	9 (9)	2234 (2234)	339 (339)	0 (0)
keep	3 (3)	2579 (2578)	0 (0)	0 (1)
list	12 (12)	2570 (2570)	0 (0)	0 (0)
no_name	2 (2)	2275 (2275)	305 (305)	0 (0)
no_time	3 (3)	2579 (2579)	0 (0)	0 (0)
recursive	4 (4)	2578 (2578)	0 (0)	0 (0)
Total	53 (53)	19634 (19628)	969 (969)	0 (6)

Table A.2: Evaluation results for project GZIP. The brackets contain the results from the initial and unrevised ground truth.

Feature	True Positive	True Negative	False Positive	False Negative
All	7 (7)	5110 (5110)	0 (0)	0 (0)
ClausalCoreFile	2 (2)	5115 (5115)	0 (0)	0 (0)
CompactTrace	2 (2)	5115 (5115)	0 (0)	0 (0)
Defaultphase_val	3 (3)	5114 (5114)	0 (0)	0 (0)
ExtendedTrace	2 (2)	5115 (5115)	0 (0)	0 (0)
FileListingCoreVariables	2 (2)	5115 (5115)	0 (0)	0 (0)
IgnoreInvalidHeader	3 (3)	5114 (5114)	0 (0)	0 (0)
NoPrintAssignment	3 (3)	5112 (5112)	2 (2)	0 (0)
Partial	3 (3)	5114 (5114)	0 (0)	0 (0)
PrintToFile	3 (3)	5114 (5114)	0 (0)	0 (0)
ReverseUnitPropagationProof	3 (3)	5114 (5114)	0 (0)	0 (0)
Verbose	22 (22)	5045 (5045)	50 (50)	0 (0)
Total	55 (55)	61297 (61297)	52 (52)	0 (0)

Table A.3: Evaluation results for project PicoSAT. The brackets contain the results from the initial and unrevised ground truth.

Feature	True Positive	True Negative	False Positive	False Negative
FlushTO_val	2 (2)	1557 (1557)	0 (0)	0 (0)
ForceOverwrite	5 (4)	1554 (1554)	0 (1)	0 (0)
Format	17 (16)	1542 (1541)	0 (1)	0 (1)
KeepIn	4 (2)	1555 (1555)	0 (2)	0 (0)
Mode	24 (24)	1534 (1534)	0 (0)	1 (1)
NoAdjust	2 (2)	1557 (1557)	0 (0)	0 (0)
Robot	7 (7)	1552 (1552)	0 (0)	0 (0)
SingleStream	2 (2)	1557 (1557)	0 (0)	0 (0)
Stdout	7 (4)	1552 (1552)	0 (3)	0 (0)
Suffix	5 (5)	1554 (1554)	0 (0)	0 (0)
Verbosity	24 (17)	1535 (1535)	0 (7)	0 (0)
Total	99 (85)	17049 (17048)	0 (14)	1 (2)

Table A.4: Evaluation results for project xz. The brackets contain the results from the initial and unrevised ground truth.

BIBLIOGRAPHY

[1] Sven Apel and Christian Kästner. "An overview of feature-oriented software development." In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[2] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. "An algebra for features and feature composition." In: *International Conference on Algebraic Methodology and Software Technology*. Springer. 2008, pp. 36–50.

[3] Andrew W. Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge university press, 2002.

[4] Don Batory. "Feature models, grammars, and propositional formulas." In: *International Conference on Software Product Lines*. Springer. 2005, pp. 7–20.

[5] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. "Hairball: Lint-inspired static analysis of scratch projects." In: *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, pp. 215–220.

[6] Jan Bosch. *Design and use of software architectures: adopting and evolving a product-line approach*. Pearson Education, 2000.

[7] Andreas Classen, Patrick Heymans, and Pierre-Yves Schobbens. "What's in a feature: A requirements engineering perspective." In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2008, pp. 16–30.

[8] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley Boston, 2002.

[9] Krzysztof Czarnecki and Ulrich W. Eisenecker. "Generative programming." In: (2000).

[10] Zhen Dong, Artur Andrzejak, David Lo, and Diego Costa. "Orplocator: Identifying read points of configuration options via static analysis." In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2016, pp. 185–195.

[11] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. "Extended static checking for Java." In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. 2002, pp. 234–245.

[12] Gordon Fraser, Ute Heuer, Nina Körber, Florian Obermüller, and Ewald Wasmeier. "Litterbox: A linter for scratch programs." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE. 2021, pp. 183–188.

[13] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. "Is static analysis able to identify unnecessary source code?" In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29.1 (2020), pp. 1–23.

[14] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities." In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE. 2006, 6–pp.

[15] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[16] Dave King, Boniface Hicks, Michael Hicks, and Trent Jaeger. "Implicit flows: Can't live with 'em, can't live without 'em." In: *International Conference on Information Systems Security*. Springer. 2008, pp. 56–70.

[17] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.

[18] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis." In: *arXiv preprint arXiv:1404.7431* (2014).

[19] V Benjamin Livshits and Monica S Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis." In: *USENIX security symposium*. Vol. 14. 2005, pp. 18–18.

[20] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. "Evaluation of Usability Criteria Addressed by Static Analysis Tools on a Large Scale." In: *Software Engineering 2023* (2023).

[21] ThanhVu Nguyen, Ugur Koc, Javran Cheng, Jeffrey S Foster, and Adam A Porter. "iGen: Dynamic interaction inference for configurable software." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 655–665.

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

[23] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "Phasar: An inter-procedural static analysis framework for c/c++." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2019, pp. 393–410.

[24] Lauritz H. Timm. "Improving VaRA's Feature Region Detection by Using an Inter-procedural Context-Sensitive Taint Analysis." BA Thesis. Saarland University, Nov. 2021.

[25] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. "TAJ: effective taint analysis of web applications." In: *ACM Sigplan Notices* 44.6 (2009), pp. 87–97.

[26] Miguel Velez, Pooyan Jamshidi, Florian Sattler, Norbert Siegmund, Sven Apel, and Christian Kästner. "Configcrusher: Towards white-box performance analysis for configurable systems." In: *Automated Software Engineering* 27 (2020), pp. 265–300.

[27] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. "White-box analysis over machine learning: Modeling performance of configurable systems." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 1072–1084.