Master's Thesis

VALIDATING VARA'S FEATURE-REGION DETECTION ON REAL-WORLD PROGRAMS BY APPROXIMATING FEATURE LOCATIONS THROUGH COVERAGE DATA

JAN SCHMITZ

January 18, 2024

Advisors:

Florian Sattler	Chair of Software Engineering
Sebastian Böhm	Chair of Software Engineering

Examiners:

Prof. Dr. Sven Apel	Chair of Software Engineering
Prof. Dr. Andreas Zeller	CISPA Helmholtz Center for Information Security

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

UNIVERSITÄT
DES
SAARLANDES

Science demands patience.

— Arthur C. Clarke

Dedicated to my family and friends.

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

Instead of writing programs for a single purpose only, developers tend to reuse existing software systems, adding new functionality or features to them as needed. Usually they introduce new configuration options that enable, disable, or tune the newly introduced features. Adding new features to existing programs is a reasonable choice from a developer perspective but comes with a hidden cost: The number of possible configurations grows exponentially with every added configuration option. Furthermore, the project gets more complicated because features may interact with one another. This complexity then complicates development and maintenance.

To reason about such complexities, the variability-aware analysis framework VaRA provides a feature-region detection that can be used to help developers manage the increased maintenance burden. The feature-region detection identifies code regions that correspond to specific features or configuration options. By mapping features to code regions, developers gain valuable insights into the program's complex structure. For instance, they can directly observe which configurations are affected when modifying a single line of code.

Although VaRA's feature-region detection has produced promising results in test applications, it has not yet undergone extensive validation. The reason for this is that the baseline needed for validation had to be created manually by hand, which is unfeasible to do for large programs in practice. Therefore, in order to validate VaRA's feature-region detection on a large scale, an automatic method is needed to generate the required baseline for real-world programs.

In this thesis, we create such an automatic method to validate the feature-region detection of VaRA on real-world programs by leveraging coverage data to automatically generate baselines. First, we conduct a qualitative analysis to identify conceptual differences when using our coverage-based baseline for validation and mitigate them if necessary. Then, we quantitatively evaluate how VaRA's feature-region detection performs on real-world programs by using our baseline for result classification. We analyze classification outcomes in detail that cannot be explained based on the insights from our first step to find their cause. In this way, we can either spot bugs in VaRA's feature-region detection or verify that it works as intended.

Our evaluation indicate that VaRA's feature-region detection on the whole works as intended and returns valid results. However, our work also reveals a potential bug and identifies some shortcomings that can be improved upon.

Software engineering is the part of computer science which is too difficult for the computer scientist.

— Friedrich L. Bauer

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to my family and friends for their unwavering support and encouragement during my studies and especially throughout this thesis. Specifically, I thank my parents for always being there for me when I needed them, while also allowing me to develop freely and pursue my interests. Furthermore, I would like to thank the Chair of Software Engineering for the supervision of this thesis. In particular, I am grateful for the collaboration with my advisors Florian Sattler and Sebastian Böhm, who provided valuable feedback and motivated me to surpass my own expectations. Special thanks go to Prof. Dr. Sven Apel and Prof. Dr. Andreas Zeller for their time and effort to examine this thesis. Finally, I would like to express my gratitude to my proofreaders for their helpful remarks and encouraging words.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

BA	Balanced Accuracy
BDD	Binary Decision Diagram
DNF	Disjunctive normal form
FN	False Negative
FP	False Positive
ISA	Instruction Set Architecture
MSMR	MultiSharedMultipleRegions
PPV	Positive Predictive Value
SFI	SimpleFeatureInteraction
TN	True Negative
TNR	True Negative Rate
TP	True Positive
TPR	True Positive Rate
VaRA	Variability-aware Region Analyzer

1 INTRODUCTION

Modern software systems often solve problems of an entire domain rather than just a single problem. To do so, they are usually highly-configurable. Similar to a swiss army knife, modern software systems provide a diverse set of functionality or have features that can be tailored to a specific use case through configuration options. Configuration options represent switches that turn on/turn off user-controllable functionality or features. They are normally parsed at program load time from, for example, command line parameters, environment variables, or configuration files and usually have corresponding variables in the source code. These feature variables influence the control flow of the program to achieve the requested behavior and adapt the program to the desired use case. This means that, depending on the variables' value, different parts of the program that implement the requested functionality of the enabled features are executed.

For developers of the aforementioned software systems, the high configurability poses challenges. With every added configuration option the configuration space grows exponentially, making it unfeasible in general to consider every possible configuration when working on the software system. Additionally, features often influence the behavior of each other in unforeseen ways [6]. For example, when one feature changes the expected program flow, data state, or visible behavior of another feature. Therefore, developers must detect, manage, and resolve these feature interactions to ensure that features and the software system as a whole work correctly [2, p. 214]. Usually that requires special handling in code. This added complexity through configurability complicates development and maintenance because every possible configuration has to be considered whenever making a change.

To illustrate the complexity challenge of configurable systems consider the ZIP example in Listing 1.1. The code on the left may be used in the ZIP[1] utility which provides several command-line options to create various kinds of ZIP files. For simplicity, assume it just provides `--compress` for enabling compression and `--enc` for enabling encryption (with corresponding feature variables in Line 1 and 2). These two command-line options can already be combined in four different ways. This results in four possible configurations of ZIP, each processing the data in a different way (right side of Listing 1.1). Since the used encryption algorithm is a block cipher, the data has to be a multiple of the block size. In this example, we assume that the `compress` function always outputs data that is a multiple of the block size. Therefore, no padding is needed when compressing and encrypting. But for the case that the data should be encrypted but not compressed, padding has to be added. This is a special case where glue code (Lines 8–10) is required that manages this interaction between the *Encryption* and *Compression* feature. Now consider the real-world ZIP utility which has over 80 command-line options, some can even be set to non-binary values. That means, the real-world ZIP utility can be configured in more than 2^{80} ways. This leads to a way higher complexity than that of our example and makes it very unlikely that a developer knows

1 https://man.archlinux.org/man/zip.1.en (visited on January 16, 2024)

```
1  bool UseCompression;                          |
2  bool UseEncryption;                           |
3                                                |
4  if (UseCompression) {                         |  Possible Data Processing:
5    Data = compress(Data);                      |   1. ZIP(Data)
6  }                                             |   2. ZIP(compress(Data))
7  if (UseEncryption) {                          |   3. ZIP(encrypt(addPadding(Data)))
8    if (not UseCompression) {                   |   4. ZIP(encrypt(compress(Data)))
9      Data = addPadding(Data);                  |
10   }                                           |
11                                               |
12   Data = encrypt(Data);                       |
13 }                                             |
```

Listing 1.1: Conceptual example of a ZIP compression tool.

which features are affected when they make a change. Hence, introducing bugs becomes more likely.

Fortunately, the analysis framework Variability-aware Region Analyzer (VaRA) [25] can help with that. VaRA provides a feature-region detection that statically analyzes a program to find code regions dependent on feature variables. The resulting code region to feature mapping can be used by other tools, for example, to annotate the feature-dependent regions in a program to assist developers of configurable software systems. Listing 1.2 depicts a visualization of the code region to feature mapping for our ZIP example.

```
1  bool UseCompression;                          |  Feature Variable: UseCompression
2  bool UseEncryption;                           |  Feature Variable: UseEncryption
3                                                |
4  if (UseCompression){                          |
5    Data = compress(Data);                      |  Compression
6  }                                             |
7  if (UseEncryption){                           |  Encryption
8    if (not UseCompression){                    |
9      Data = addPadding(Data);                  |  Encryption, Compression
10   }                                           |
11                                               |  Encryption
12   Data = encrypt(Data);                       |
13 }                                             |
```

Listing 1.2: ZIP example, feature regions detected by VaRA.

However, VaRA's feature-region detection has not yet been validated on large-scale on real-world programs. This is due to the lack of an easy way to generate the necessary baseline or

reference data for verifying VaRA's feature-region detection results. Typically, generating such a baseline is done manually, which is a very labor-intensive task that is impractical for large code bases [27]. Therefore, research focused on evaluating different feature-region detection approaches [27] or only validated some components of VaRA's feature-region detection [14].

Fortunately, coverage data poses a solution to this impediment because it contains information about the executed source code of a program, including the source code belonging to active features during a run. By leveraging this information, we can determine which code regions belong to which feature [22] and generate a baseline for a program automatically. A coverage-based trace of a program execution contains the code regions that were active during that execution. When the same program is executed again but this time with exactly one configuration option changed, the active code regions change too. The difference in active code regions is the influence of the changed configuration option (i. e., the code that belongs to that configuration option [22]). Therefore, executing the program for all possible configurations and building the difference between the obtained coverage data yields a feature to code region mapping that is based on run-time information, we call it coverage-based baseline. Listing 1.3 depicts our coverage-based baseline for our ZIP example.

In this thesis, we validate VaRA's feature-region detection by comparing its results with our coverage-based baseline. Since both approaches use completely different concepts to generate the same type of data, it is unlikely that they share the same shortcomings. Therefore, comparing the results of both gives us the chance to find bugs in VaRA's feature-region detection or confirm that it works as intended.

```
1  bool UseCompression;                          
2  bool UseEncryption;                           True
3                                                
4  if (UseCompression){                          
5    Data = compress(Data);                      compress
6  }                                             True
7  if (UseEncryption){                           enc
8    if (not UseCompression){                    
9      Data = addPadding(Data);                  ¬compress ∧ enc
10   }                                           
11                                               enc
12   Data = encrypt(Data);                       
13 }                                             
```

Listing 1.3: ZIP example, presence conditions by leveraging coverage data.

1.1 GOAL OF THIS THESIS

The goal of this thesis is to validate VaRA's feature-region detection on large-scale real-world programs. To accomplish this, we provide a way to automatically build a coverage-based

baseline for real-world programs and demonstrate how it can be compared to the results of VaRA's feature-region detection. Since VaRA's feature-region detection technique relies on static code analysis, while the generation of the coverage-based baseline is basically a dynamic analysis, it is uncertain how comparable the results of these two approaches are. Therefore, we first validate small example programs with our coverage-based baseline and analyze the results in detail to draw conclusions about possible differences in the identified feature regions. In a second step, we evaluate how well VaRA's feature-region detection performs on real-world programs by using the coverage-based baseline to measure performance metrics. We base our assessment of these results on the conclusions drawn in step one and examine cases where the two approaches diverge in detail. This way, we determine whether the cause is a shortcoming of one approach or whether we have found an actual bug in the implementation. Therefore, we can either identify areas for improvement in VaRA's feature-region detection or verify that it works as intended.

1.2 OVERVIEW

This thesis is structured as follows: Chapter 2 provides the necessary background information for our work. It introduces important terms and concepts of configurable software systems and explains the tools and analysis approaches that VaRA and our coverage-based baseline use. In Chapter 3, we describe how we build our coverage-based baseline and how we associate VaRA's feature-region information with it. Chapter 4 focuses on our methodology. It includes our research questions, explains how we interpret the information from both approaches to make them comparable, describes our comparison procedure, and presents the operationalization of our experiments. In Chapter 5, we showcase our results, discuss them, answer our research questions and thesis goal, and disclose threats to validity. Chapter 6 provides an overview of related work, specifically previous research on feature detection in general, previous evaluation attempts related to VaRA's feature-region detection, as well as relevant work upon which we base our implementation. Lastly, Chapter 7 concludes what we have accomplished and gives recommendations for immediate possible tool improvements around VaRA and future work.

2 BACKGROUND

We present in this chapter the essential background knowledge needed to understand our work. First, we discuss the concepts and terminology of configurable software systems. Then, we introduce the respective tools and techniques of VaRA and our coverage-based baseline.

2.1 CONFIGURABLE SOFTWARE SYSTEMS

A software system consists of several cooperating components that work together to solve a task. Each component has a specific purpose and contributes to the overall solution of the problem. Components can be implemented in various ways, for example, as an independent program or as a part of a program itself. To avoid developing software systems from scratch, the concept of software product lines has been established.

A software product line is a set of useful, reusable components, which can be combined into a software product or software system tailored to the user's needs [2, Section 1.3]. Each of these components realizes a certain functionality. If this functionality is end-user visible, then it is called a feature [2, Definition 2.1].

The variability of a software product line, that is, which features can be combined to obtain a valid software product, is commonly modeled by a feature model [2, pp. 26–27]. Usually, feature models can be transformed into a Boolean expression that describes all valid feature combinations. A visual representation of a feature model is a feature diagram, a tree structure with features as nodes and possibly additional constraints. The purpose of this representation is to display for each node the conditions that child features must satisfy to obtain valid configurations. As an example, we present the feature diagram for our ZIP example in Figure 2.1, which we extended to include decompression support for the sake of completeness. Feature models are composed of both abstract and concrete features. Abstract features are used to structure the feature model and for documentation purposes, but unlike concrete features, they typically do not have implementation artifacts [2, p. 35]. In our feature diagram, we have two abstract features that group features. *ZIP* acts as the root of the tree and represents the entire functionality of the ZIP example. It forms an "And" group, which requires all conditions of the features in this group to be fulfilled to obtain valid configurations. Furthermore, we have the abstract feature *OperationMode*, which is used to assign the concrete features *Compress* and *Decompress* to an alternative group. This means that *Compress* and *Decompress* cannot be activated at the same time. In addition to group memberships, features can be either optional or mandatory. *Concat* is mandatory in our case, which means that our ZIP example always concatenates several files into a single one. The features *Encrypt* and *OperationMode* are optional, meaning that the encryption and the compression or decompression functionality can be activated as needed depending on the use case. Additional constraints that cannot be expressed directly in the tree are shown below in the feature diagram. In our ZIP example, this is the condition that we cannot encrypt while decompressing. This case is not relevant in practice and is therefore not supported.

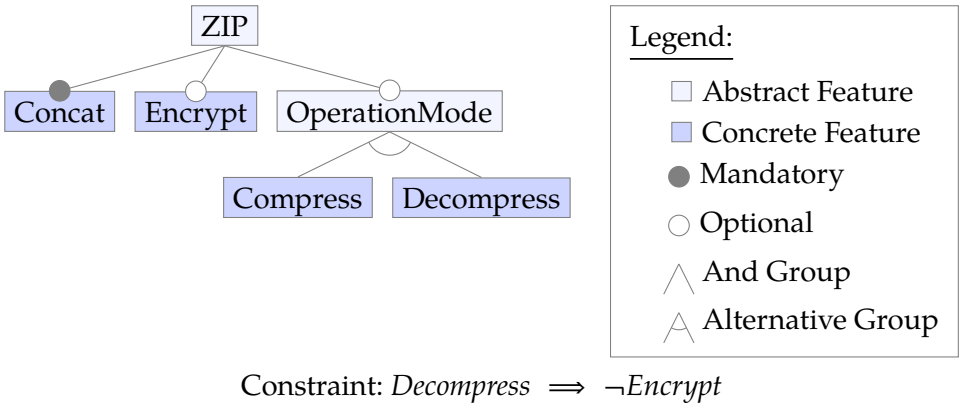


Figure 2.1: Feature model of ZIP example visualized as feature diagram.

Software product lines may have features that are always active and cannot be configured by the user, such as our mandatory *Concat* feature. However, developers often provide users with the ability to customize their software system by implementing configuration options that allow them to enable or disable features or tweak their functionality. The end users can configure the program to their needs by setting the desired configuration options. Developers have three options for implementing variability in their software product line: at compile time, load time or run time [2, Section 3.1.1]. Preprocessor directives or configuration options in the build system allow for compile-time variability. Load-time configuration options are typically implemented through configuration files, environment variables, or command-line options. A program evaluates them at load time, that is, before it starts operating. Run-time configuration options are often also set at load time but unlike load-time configuration options they can be changed during run time. They allow users to reconfigure a program without restarting it.

VaRA's feature-region detection can analyze the use of load-time and run-time configuration options in programs. However, in our experiments, we only examine load-time variable programs configured via command-line options. Therefore, in this work, we define a feature as a configurable feature with a corresponding configuration option. Furthermore, a configuration option is equivalent to a command-line option in the following, unless we state otherwise.

2.2 VARA

The Variability-aware Region Analyzer (VaRA) is an analysis framework based on LLVM [25]. It provides several high-level static or dynamic analyses that work on LLVM IR and allow users to detect code of interest. In the following subsections we explain important terminology and concepts related to VaRA.

2.2.1 *Static and Dynamic Analysis*

A static analysis works on the source code of a program or the instructions generated from it without executing these. In contrast, a dynamic analysis executes the program and observes the execution. Both static and dynamic analysis methods have advantages and disadvantages.

Static analysis methods often require an entry point in the source code to start from, which requires some preparation before they can be used. Dynamic analysis methods, on the other hand, can usually be used out of the box without any adjustments or markups to the source code. However, they do not scale as well as static analysis methods, as they only collect information about one program execution at a time. To analyze the entire program, dynamic analysis must be performed repeatedly until all executions have covered the entire source code. However, this task requires a significant amount of computational effort and is not feasible for sufficiently complex programs. In contrast, static analysis only needs to analyze the source code once and can then make statements about the entire program.

2.2.2 *LLVM*

LLVM is a compiler framework [17], which allows to compile a program for an arbitrary supported CPU instruction set architecture (ISA). LLVM achieves this ability through its modular design. It has three different types of components, language frontends, the LLVM optimizer, and ISA backends. The only interface between LLVM components is an architecture-independent intermediate representation of a program, referred to as LLVM IR [16].

As shown in Figure 2.2, the first step in the compilation process of an input program with LLVM is to parse the source code by an corresponding language frontend which translates the program into LLVM IR. The resulting LLVM IR may not be optimal yet, for example, it may contain instructions for code that will never be executed. Therefore, as second step, an optimizer can be used that performs various analyses, called LLVM passes, on the LLVM IR. A LLVM pass can alter the LLVM IR in any way. Usually, passes implement optimizations that make the LLVM IR more efficient, for example, by removing unnecessary instructions. The final step in the LLVM compilation process is to process the LLVM IR with an ISA backend that produces the actual machine code that can be executed on the desired CPU architecture.

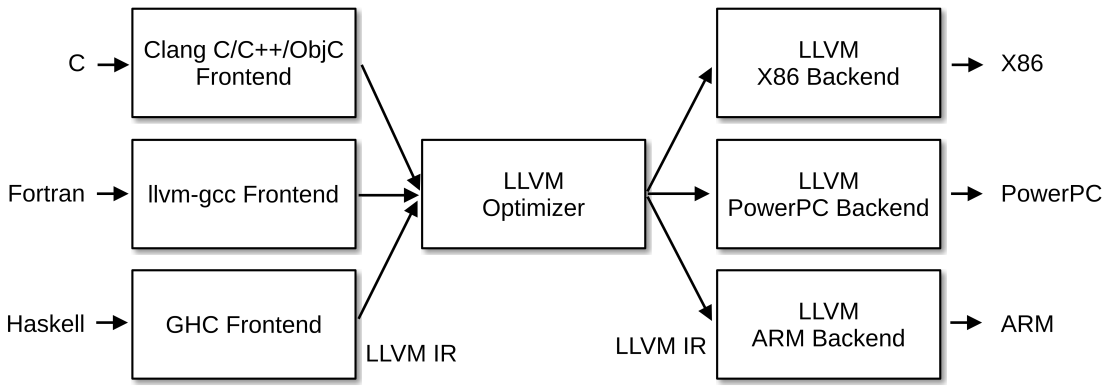


Figure 2.2: LLVM compiler architecture as depicted in [16].

2.2.2.1 *LLVM IR*

LLVM IR is a low-level program representation that is later translated into machine-specific code. The LLVM IR for Lines 1–6 in our ZIP example code is shown in Listing 2.1. One can

clearly recognize the variable declarations of `UseCompression` and `UseEncryption` (Lines 1–2), as well as the `UseCompression` if-case (Lines 5–14).

Declarations and instructions in LLVM IR can be annotated with metadata. This metadata provides additional information about the entity that can be helpful when debugging LLVM passes, for example. Metadata is appended to the end of an entry in the comma-separated format: `!<metadata_name> !<reference_number>`. The reference number refers to the corresponding metadata entry at the bottom of the LLVM IR. We can also see this in Listing 2.1, where we instruct LLVM to generate metadata for debugging. This results in declarations and instructions having `!dbg` metadata that links them to the corresponding locations in the source code (Lines 18–22).

```
1  @UseCompression = internal global i8 0, align 1, !dbg !991
2  @UseEncryption = internal global i8 0, align 1, !dbg !992
3  ...
4  entry:
5    %1 = load i8, i8* @UseCompression, align 1, !dbg !2683
6    %tobool = trunc i8 %1 to i1, !dbg !2683
7    br i1 %tobool, label %if.then, label %if.end, !dbg !2685
8
9  if.then:
10   ; Call compress function
11    ...
12   br label %if.end, !dbg !2691
13
14 if.end:
15 ; Continue with program
16 ...
17
18 !991 = !DILocation(line: 1, column: 1, scope: !42)
19 !992 = !DILocation(line: 2, column: 1, scope: !42)
20 !2683 = !DILocation(line: 3, column: 1, scope: !2684)
21 !2685 = !DILocation(line: 4, column: 21, scope: !2677)
22 !2691 = !DILocation(line: 6, column: 1, scope: !2687)
```

Listing 2.1: ZIP example code (Listing 1.1) Line 1–6 as LLVM IR.

2.2.3 *Control-Flow Graph*

The control flow in a program is determined by conditions. Depending on the conditions, so-called basic blocks are executed or not. “A basic block is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed).” [1] The possible control flows in a program can be represented by a control-flow graph (CFG). “A control flow graph is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths.” [1]

If all control-flow paths to a basic block X go exclusively through a basic block Y, we say basic block X is dominated by Y [1]. The same works in the other direction. Here we speak of post-domination [1].

Such a control-flow graph for the ZIP example is depicted in Figure 2.3. It has the basic blocks A-G as nodes and the control-flow decisions as edges.

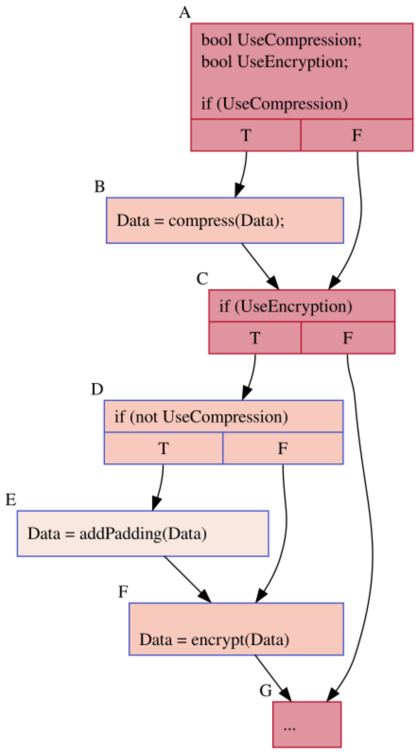


Figure 2.3: Control-flow graph of the ZIP example (Listing 1.1).

2.2.4 *Feature-Taint Analysis*

Taint analysis is a static or dynamic analysis that reveals the information flow in a program. It begins at a labeled start point (taint source) and marks all places in a program that use the taint source as tainted. If such a place is a variable assignment, then the assigned variable is also marked as tainted. The taint analysis continues its search on the freshly marked variables and eventually identifies all places in the program that use information from the taint source.

VaRA implements a static feature-taint analysis that works with feature variables as taint source. It taints LLVM IR instructions that depend on a feature variable as feature-dependent. This way, it creates a mapping from feature variables to the affected instructions.

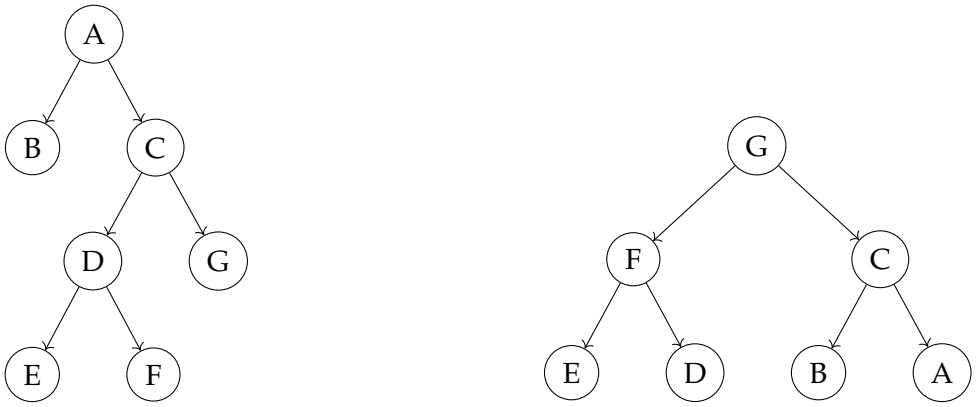
2.2.5 *VaRA's Feature-Region Detection Approach*

VaRA detects feature regions in two steps. First, it uses the feature-taint analysis to obtain feature-dependent instructions. In the second step, it analyzes the last instruction in each basic block to determine which basic blocks depend on a feature. The last instruction in a

basic block of LLVM IR is a branch instruction that determines which basic block is executed next. Therefore, if this last instruction is a feature-dependent conditional branch instruction, the decision depends on the corresponding features. VaRA treats all connected basic blocks affected by this decision as a feature region.

VaRA's feature-region detection determines which basic blocks belong to a feature region through domination and post-domination relationships in the control-flow graph. It uses dominator trees to describe the domination relationship. A dominator tree has the same basic blocks as nodes as the control-flow graph and a node's children are the basic blocks that are directly dominated by it. For our ZIP example, we visualize the dominator trees in Figure 2.4. The nodes *A*-*G* correspond to the basic blocks in Figure 2.3. A feature region consists of basic blocks that are dominated by a feature-dependent condition but are not post-dominated by it. The dominator trees illustrate that, for example, for the basic block *A*. *A* dominates both *B* and *C*, but *C* post-dominates *A*. As a result, *C* and all basic blocks dominated by *C* are not part of the feature region. This leaves only *B* as the basic block that is exclusively affected by the feature-dependent condition in *A*. VaRA's feature-region detection counts the basic block with the feature-dependent condition as part of a feature region by default. Therefore, in this case, the feature region consists of both *A* and *B*.

In total, VaRA's feature-region detection finds three feature regions in our ZIP example: two for the *Compression* feature, consisting of the basic blocks *A* and *B*, as well as *D* and *E*. The third feature region belongs to the *Encryption* feature and consists of *C*, *D*, *E*, and *F*.



Dominator tree Post-dominator tree

Figure 2.4: Dominator trees of the ZIP example.

2.3 COVERAGE-BASED BASELINE

Our coverage-based baseline is a mapping of code regions to features, or more precisely, to their presence conditions. Presence conditions describe feature combinations under which the code regions are executed and can be automatically generated from coverage information. In the following, we explain terms and techniques relevant to our coverage-based baseline.

2.3.1 *Baseline and Ground Truth*

A baseline is reference data that serves as a basis for comparison with other data. It enables the determination of differences between the baseline and other data, and can be used to classify the latter. If the baseline has been verified and is known to be true, it is often referred to as ground truth.

Since, in this thesis, we automatically generate a code region to feature mapping from coverage data without always verifying it, we use the term coverage-based baseline rather than coverage-based ground truth.

2.3.2 *Coverage Data*

Measuring code coverage, that is, what parts of a program are executed during run time, is a common and established technique due to the easy accessibility of tools for most programming languages[1]. Coverage information is often used to estimate the quality of software tests, because the more code of a program is tested, the more likely it is that bugs are found [20].

LLVM can also measure code coverage. We can instruct the compiler to compile a program that generates a coverage log when it is executed. A coverage log contains information about the code regions of the program and the count of how many times they were executed. We use these counts to determine which code regions were active in a program and identify code regions that belong to a feature.

The command-line options listed below take care of compiling an instrumented program:

- **-fprofile-instr-generate:** Enables instrumented code generation that collects execution counts in a `.profraw` file[2].
- **-fcoverage-mapping:** Writes a coverage mapping[3] to the `.profraw` file to enable code coverage analysis[4].

The resulting instrumented program saves raw profiling data during its execution in a `.profraw` file. A `.profraw` file is an append-friendly unstructured LLVM-internal format that is neither forward nor backward compatible and can change with every LLVM version[5]. Therefore, in order to work with the raw data and to create a coverage report from them, they must first be indexed[6]. LLVM provides the `llvm-profdata` tool for this purpose. With the command:

```
llvm-profdata merge name.profraw -o name.profdata
```

1 Java: https://www.jacoco.org/jacoco/ (visited on January 16, 2024); Python: https://coverage.readthedocs.io (visited on January 16, 2024); Many more: https://about.codecov.io/ (visited on January 16, 2024)

2 https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fprofile-instr-generate (visited on January 16, 2024)

3 https://llvm.org/docs/CoverageMappingFormat.html (visited on January 16, 2024)

4 https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fcoverage-mapping (visited on January 16, 2024)

5 https://clang.llvm.org/docs/SourceBasedCodeCoverage.html?highlight=profraw#format-compatibility-guarantees (visited on January 16, 2024)

6 https://clang.llvm.org/docs/SourceBasedCodeCoverage.html?highlight=profraw#creating-coverage-reports (visited on January 16, 2024)

an indexed data profile (`.profdata` file) is created from which coverage information can be extracted by `llvm-cov`[7]. In order to make the coverage data easy to parse, we export it as JSON with:

```
llvm-cov export --instr-profile=name.profdata path/to/instrumented_program
```

2.3.3 *Code Regions and Feature Regions*

Code regions are parts of the source code that are grouped together by a specific syntax and may consist of one or more statements. In C/C++ programs, for example, curly braces or control-flow structures such as if-then-else statements form code regions. In this thesis, code regions refer to the corresponding data structure in LLVM's coverage JSON[8]. From our perspective, the most important fields in this data structure are the start and end locations in the source code, which define the contents of a code region, and the execution count, which tells us how often that code region has been executed. By parsing this data structure, we can determine the entire structure of a program.

From the standpoint of VaRA's feature-region detection, we can define a code region as consisting of one or more basic blocks that are located directly in that code region, given the source-code location of their instructions. Apart from minor deviations, which we describe in Section 5.1.1.2 and Paragraph 5.2.1, this understanding is correct.

In the ZIP example (Listing 1.3), there are essentially four code regions color coded as gray (Lines 1–13), blue (Lines 4–6), orange (Lines 7–13), and brown (Lines 8–10). We depict how these code regions are nested as a code region tree in Figure 2.5. A code region tree consists of nodes that represent code regions. The code region of each child node in a code region tree is entirely contained within the start and end locations of the parent node. We use the labels of the nodes to visualize the basic blocks from Figure 2.3 that are exclusively contained in a code region.

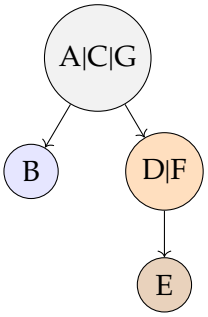


Figure 2.5: Code region tree.

FEATURE REGION In the context of the coverage-based baseline, we call a code region a feature region when the execution of the code region depends on a configuration option. Regarding VaRA's feature-region detection, we define a feature region as a code region whose

7 https://llvm.org/docs/CommandGuide/llvm-cov.html (visited on January 16, 2024)
8 https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports (visited on January 16, 2024)

associated instructions are all detected as feature-dependent. Therefore, the blue, orange, and brown code regions in our ZIP example are feature regions for both approaches. To differentiate between feature regions and regions without features associated, we use the term "normal" region to refer to a code region that is not a feature region.

FLAKY FEATURE REGION Due to the instruction-level granularity of VARA's feature-region detection, it is possible that not all instructions within a code region are identified as feature-dependent by VARA. Therefore, if a code region consists of both feature-dependent and non-feature-dependent instructions, we call the code region a flaky feature region.

2.3.4 *Binary Decision Diagram (BDD)*

Boolean formulas respectively functions can be efficiently represented by a Binary Decision Diagram (BDD) [4, 5], as can be seen for example in Figure 2.6. A BDD is a directed acyclic graph whose nodes represent the variable of the Boolean formula and its edges represent the values true (1; continuous arrow) or false (0; dashed arrow). The graph has a start node and two terminal nodes, 0 and 1, denoting the result of the Boolean function. Each path through the graph represents a possible input and its corresponding output. An interesting property of BDDs is their canonical form, that is, equivalent Boolean functions with the same input variable order have the same unique function graph [4]. This allows for the automatic minimization of Boolean formulas, which is why BDDs can be used for logic minimization, that is, minimizing a formula to an equivalent shorter formula.

Figure 2.6: Example BDDs.

2.3.5 *Presence-Condition Simplification*

If certain states of a Boolean expression cannot occur, they can be removed to simplify the expression. In our case, we know from the feature model which valid configurations a program has. Therefore, we can use this knowledge to minimize presence conditions. Rhein et al. [24]

introduced the term "Presence-Condition Simplification" for this approach and formally described the concept with the formula:

$$m \implies (\text{simp}(p, m) \iff p)$$

In this formula, m represents the context, which in our case is the feature model, p represents the presence condition, and "simp" stands for a simplification algorithm. The formula states that, given a context m and a presence condition p, the presence-condition-simplification algorithm must output a presence condition that is equivalent to the original presence condition p in the context of m. Ideally, the new presence condition is shorter than the original. In other words, presence-condition simplification minimizes the presence condition in context of the feature model.

We follow the recommendation of Rhein et al. [24] and apply the RESTRICT algorithm to simplify presence conditions represented as BDDs [5, 8]. Admittedly, this only provides us a minimized presence condition, not necessarily the most minimal one. To obtain a minimal presence condition, we would have to use the QUINE-MCCLUSKY algorithm [9], which scales poorly because it is NP-complete. However, for our purpose, a guaranteed optimal solution is not necessary since we only aim to reduce the complexity of presence conditions.

3 IMPLEMENTATION

In the following sections, we explain how the coverage-based baseline, our code region to features mapping used to validate VaRA's feature-region detection, is generated by leveraging coverage data and how VaRA's information about feature regions can be exported and associated with the baseline.

3.1 GENERATING THE COVERAGE-BASED BASELINE

To create the coverage-based baseline for a program, we require coverage logs first. To generate these, we proceed as described in Section 2.3.2 and instrument the program with profiling instructions. In addition, we generate all valid configurations from the feature model. We then run the instrumented program for each configuration on all matching workloads. The execution of all workloads matching the configuration is important to increase the coverage because in real-world programs parts of the program execution may depend on the type of workload. The more code we execute for a configuration, that is, the more workloads we run, the more complete our coverage-based baseline will be. Ideally, our workloads execute all possible code for each configuration, giving us the optimal baseline.

3.1.1 *Initial approach: Diffing coverage data*

Initially, we tried to generate the coverage-based baseline by computing the difference in coverage data. To do this, we divide the coverage logs into two groups. Group A comprises the coverage logs in which a feature X is deactivated, while group B comprises the logs with feature X activated. By merging the coverage logs in group A and those in group B, we obtain coverage log a and b. The sole dissimilarity between coverage log a and b is feature X. Thus, the difference in covered code regions between a and b corresponds to the code regions affected by feature X. By repeating this procedure for further features, we can determine the features that affect each code region.

While this approach is straightforward and sufficient for a direct comparison with the features identified by VaRA, it can only determine features that affect a code region, not the presence condition of the code region. For instance, regarding Line 9 in the ZIP example presented in Listing 1.3, we just know that it depends on *compress* and *enc*, but not that it will only be executed if $\neg compress \wedge enc$. However, this presence condition is helpful when analyzing possible inconsistencies of the feature regions detected by VaRA in the LLVM IR, as it provides us with additional context. Therefore, we changed our approach to the following in order to obtain presence conditions.

```
class CodeRegion:
    start: RegionStart
    end: RegionEnd
    count: int # How often this code region was executed
    parent: tp.Optional[CodeRegion] # Code region in which this is located
    children: tp.List[CodeRegion] # Subregions of this code region
    presence_condition: tp.Optional[Function] # Formula in BDD
    vara_instrs: tp.List[VaraInstr] # LLVM IR instruction associated with this node
    ...
```

Listing 3.1: Definition of a node in the code region tree.

3.1.2 *Better approach: Building presence conditions*

We can build presence conditions for every code region by utilizing the coverage logs recorded for each configuration. A coverage log contains all code regions of our program, their corresponding source code file and how often they were executed. We parse this information and use it to create a code region tree per file. The code region tree consists of nested code regions. A node in this tree is a code region depicted in Listing 3.1. The children of a node are all code regions that lie directly in the code region. If a node has no children, it is a leaf. The root of the code region tree is a code region that spans the whole file. It contains all other code regions.

Since all coverage logs come from the same program, the locations of all code regions are identical, and thus the structure of the code region trees is also identical. They only differ in the execution counts of the code regions. We use this to determine the presence conditions, that is, under which conditions the code region was executed. To do so, we proceed with the three steps shown in the pseudocode in Listing 3.2.

```
# Create code region trees for every configuration.
trees = []
for configuration in valid_configurations(feature_model):
    tree = record_coverage(configuration)

    # Step 1: Annotating conditions
    tree.annotate(configuration)
    trees.append(tree)

# Step 2: Building presence condition
feature_tree = merge(trees)

# Step 3: Presence-condition simplification
for code_region in feature_tree:
    presence_condition = code_region.presence_condition
    presence_condition.restrict(feature_model)
```

Listing 3.2: Pseudocode for creating a coverage-based baseline.

3.1.2.1 *Step 1: Annotating conditions*

Initially, we convert the configuration corresponding to the coverage log to a Boolean formula with command-line options as variables. To do this, we set the command-line options that are specified in the configuration as positive literals and the others as negative literals. We then connect the literals using conjunctions to create the formula. For example, if our ZIP example was executed only with the *compress* command-line option, we get the condition $compress \wedge \neg enc$. We set this condition as initial presence condition for all executed code regions in our code region tree. For all code regions that are not executed, we set the presence condition to *False*.

3.1.2.2 *Step 2: Building presence condition*

We now have structurally identical code region trees with initial presence conditions for each coverage log. The actual presence conditions for each code region are determined by merging all code region trees per source code file. This happens in the second step. When merging two code region trees, we simply build the disjunction of both presence conditions. In this way, we build a formula in disjunctive normal form (DNF) for each code region. Through using a BDD as the underlying data structure for our Boolean expressions, the presence condition is minimized with each merge. We merge the code region trees until we have one tree per source code file left. These trees contain the presence condition for each code region.

3.1.2.3 *Step 3: Presence-condition simplification*

We now have the presence condition for each code region, but not yet in the context of the feature model. Let's assume that at least one of the features *Compression* or *Encryption* must always be enabled in our ZIP example. So after the second step, all code regions that are consistently executed independently of a feature, would have the presence condition $compress \vee enc$. This, of course, distorts our coverage-based baseline because it seems that these code regions are affected by the *Compression* and *Encryption* feature. To correct this error, we apply presence-condition simplification to obtain the presence conditions in the context of the feature model. To do so, we restrict the presence conditions with the feature model. Since the feature model itself is $compress \vee enc$, the affected presence conditions reduce to *True*.

The changes in the presence conditions of the ZIP example are visualized in Listing 3.3. The column *Annotated Conditions* contains the conditions that we build from the configurations as described in the first step. *Presence Condition* is the Boolean formula, automatically minimized by the BDD. The last column *Simplification* contains the presence condition, after the restriction to the feature model.

3.2 EXPORTING VARA'S FEATURE REGIONS

The feature-region detection of VaRA is based on the dominator approach described in Section 2.2.5 and implemented as a LLVM pass. It operates at an instruction-level granularity, that is, for each instruction in the LLVM IR, we can see what features it depends on. However, this data is only accessible internally to other LLVM passes. Therefore, to compare the feature regions identified by VaRA with those from our coverage-based baseline, we must first export

	Annotated Conditions	Presence Condition	Simplification
1 `bool UseCompression;` 2 `bool UseEncryption;` 3	$c \wedge \neg e, \neg c \wedge e, c \wedge e$	$c \vee e$	$True$
4 `if (UseCompression){` 5 `  Data = compress(Data);` 6 `}`	$c \wedge \neg e, c \wedge e$	c	c
7 `if (UseEncryption){`	$c \wedge \neg e, \neg c \wedge e, c \wedge e$	$c \vee e$	$True$
8 `  if (not UseCompression){`	$\neg c \wedge e, c \wedge e$	e	e
9 `    Data = addPadding(Data);`	$\neg c \wedge e$	$\neg c \wedge e$	$\neg c \wedge e$
10 `  }` 11			
12 `  Data = encrypt(Data);` 13 `}`	$\neg c \wedge e, c \wedge e$	e	e

Listing 3.3: ZIP example, generating the coverage-based baseline in three steps. The feature model is $compress \vee enc$. *compress* is abbreviated with c, *enc* with e.

all instructions and their features, and then assign them to code regions. To accomplish this task, we perform the following steps.

At first, we generate LLVM IR from the program's source code, containing the information needed for VaRA's feature-region detection and our LLVM pass called *FeatureDebugLocationExport*. We accomplish this by utilizing the *CFLAGS*:

- **-O0:** To disable almost all other LLVM passes. This gives us the unoptimized LLVM IR.
- **-g:** To annotate the LLVM IR with source-level debug information[1] (i. e., line and column number as shown in Listing 2.1).
- **-fno-exceptions:** To disable generating exception handling code for C++[2]. This is necessary to get comparable feature regions from VaRA for C++ programs. Omitting it causes the instructions following exception handling code within feature regions to also depend on the previously identified features, resulting in non-intuitive feature regions as demonstrated in Listing A.2.
- **-fvara-fm-path=<value>:** Path to feature model for delivering VaRA the locations of feature variables in the source code.
- **-fvara-feature:** To mark feature variables in LLVM IR by metadata.

1 https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-g (visited on January 16, 2024)

2 https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-fexceptions (visited on January 16, 2024)

The resulting LLVM IR is similar to Listing 2.1, but the feature variables are now linked to the corresponding feature by the metadata `!FVar`, as depicted in Listing 3.4.

```
1 @UseCompression = internal global i8 0, align 1, !dbg !991, !FVar !1932
2 @UseEncryption = internal global i8 0, align 1, !dbg !992, !FVar !1933
3
4 ...
5 !1932 = !{!"Compression"}
6 !1933 = !{!"Encryption"}
```

Listing 3.4: VaRA's feature variable metadata annotations in the LLVM IR of the ZIP example.

Next, we run VaRA's feature-region detection to generate feature-region information for instructions in the LLVM IR and then our LLVM pass called *FeatureDebugLocationExport* to export this information. This is done by invoking the LLVM optimizer OPT with the following options:

- **-vara-PTFDD:** Runs VaRA's feature-region detection pass.
- **-vara-export-feature-dbg:** Runs the *FeatureDebugLocationExport* pass that exports LLVM IR instructions together with relevant information.
- **-vara-report-outfile=<value>:** To which file the exported data is written.

The FeatureDebugLocationExport pass iterates over all instructions in the LLVM IR and exports instructions with debug locations in CSV format. Instructions without debug metadata are not directly related to the source code and can be ignored. For each instruction, we export the path to the source file, the line and column number, the features annotated by VaRA, the consecutive instruction index, and the instruction itself. The instruction index and the instruction itself are not required for comparison with the coverage-based baseline, but are useful for analyzing inconsistencies. The relevant data from the export of our ZIP example, namely the location and the annotated features, can be seen in Table 3.1.

The exported instructions can be associated to the code regions in our code region trees based on their location. This is done by searching the code region tree belonging to the source file for the smallest code region in which the instruction is located. That is the deepest code region in the code region tree, between whose start and end point the location lies. After finding this code region, the corresponding instructions and their features are mapped to it. In this way, we get a code region to feature mapping for VaRA. The pseudocode for this can be seen in Listing 3.5.

We have demonstrated how coverage data can be utilized to create a coverage-based baseline, which consists of code regions and their corresponding presence conditions, and how to map LLVM IR instructions, including feature information from VaRA, to these code regions. Based on this data, we validate VaRA's feature-region detection.

Table 3.1: Columns of the data export for comparison with the coverage-based baseline. Every row contains the location of an instruction in the ZIP example and the associated features.

Source File	Line	Column	Features
ZIP.cpp	1	1	-
...	...	...	...
	4	21	Compression
	5	3	Compression
	...	...	...
	7	1	-
	...	...	...
	7	20	Encryption
	...	...	...
	8	27	Encryption, Compression
	...	...	...
	12	3	Encryption
	...	...	...
	13	2	-

```
1 for instr in data_export:
2     source_file = instr[”source_file”]
3     code_region_tree = trees[source_file]
4     line = instr[”line”]
5     column = instr[”column”]
6     code_region = code_region_tree.find(line, column)
7     code_region.vara_instrs.append(instr)
```

Listing 3.5: Assigning exported instructions to code regions to create a feature to code region mapping for VaRA.

4 METHODOLOGY

We validate the results of VaRA's feature-region detection by comparison with our coverage-based baseline. In Chapter 3, we already described the creation of our coverage-based baseline and how VaRA's feature-region information can be exported and mapped to code regions. In this chapter, we focus on the research questions we answer in order to validate VaRA's feature-region detection. Furthermore, we discuss our comparison process and the analyses of the comparison results we perform to answer the research questions.

4.1 RESEARCH QUESTIONS

We use our coverage-based baseline to validate VaRA's feature-region detection and answer our thesis goal—whether VaRA's feature-region detection yields valid results. To do this, we compare the results of the two approaches. Since both approaches are implemented independently and are conceptually different, it is unlikely that feature-region detection errors occur in the same place. That is, by examining the deviations of VaRA's feature-region detection results from the coverage-based baseline, we can identify these errors or verify that VaRA's feature-region detection works as intended. In order to understand the deviations and the performance of VaRA's feature-region detection, we have two research questions.

Conceptual Differences

Our coverage-based baseline relies on run-time information, whereas VaRA's feature-region detection is based on static analysis. As a consequence, not all differences between the results necessarily indicate an error in either approach. Rather, they may be caused by *conceptual differences* that lead to different detected feature regions. Consequently, in order to correctly classify and interpret the comparison results, we need to understand how the two approaches differ conceptually. Therefore, we formulate our first research question:

> **RQ1:** What conceptual differences in detected feature regions exist between the coverage-based baseline and VaRA's feature-region detection?

Performance of VaRA's Feature-Region Detection on Real-World Programs

Once we understand how the coverage-based baseline and VaRA's feature-region detection differ conceptually, we can mitigate these differences, which makes it easier for us to evaluate the performance of VaRA's feature-region detection with the coverage-based baseline. We evaluate the performance using real-world programs, as this gives us more general results than if we were to evaluate only individual example programs or synthetic benchmarks. Therefore, our second research question is:

> **RQ2:** How well does VaRA's feature-region detection perform on real-world programs?

To assess VaRA's feature-region detection performance, we determine how many of the detected feature regions are correct or incorrect from the perspective of the coverage-based baseline. Precision and recall are relevant performance metrics that provide insight into this matter. Precision is the percentage of feature regions that VaRA's feature-region detection identified correctly. However, it does not provide information about the feature regions that VaRA does not identify. That is the purpose of recall (i. e., the percentage of valid feature regions that VaRA's feature-region detection identifies). Therefore, we divide the second research question into the two sub-questions:

> **RQ2.1:** What is the fraction of detected feature regions that are genuine? (Precision)
> **RQ2.2:** What is the fraction of genuine feature regions that are detected as such? (Recall)

Based on the research questions and any discovered shortcomings or bugs in one of the approaches, we can determine the validity of VaRA's feature-region detection results. The following sections describe the methodology we use to answer the research questions and to achieve our thesis goal.

4.2 COMPARISON PROCESS

This section describes how we interpret the data of the coverage-based baseline and VaRA's feature-region detection such that we can compare feature regions of both approaches, classify feature regions of VaRA using our coverage-based baseline as reference, and compute relevant performance metrics.

4.2.1 *Interpreting Feature Regions*

The raw data from the coverage-based baseline consists of code regions associated with a presence condition. Our data of VaRA's feature-region detection consists of code regions and corresponding LLVM IR instructions. However, presence conditions and instructions cannot be compared directly. Therefore, to be able to compare them, it is necessary to determine whether a code region qualifies as a feature region based on our coverage-based baseline or VaRA.

For our coverage-based baseline, this is straightforward. A code region is a feature region if its execution depends on a command-line option. This is the case if its presence condition contains command-line options as variables, that is, the presence condition is neither *True* nor *False*.

For VaRA, the situation is more complicated. Since the feature-region detection of VaRA works at instruction-level granularity, it is possible that some instructions in a code region have features assigned to them but others have not. Thus, it is not always straight-forward to determine whether VaRA recognizes a code region as feature region and on which features the region depends on. In fact, there are only two clear cases. First, if a code region has no instructions or only instructions without assigned features, then it is just a *normal* code region. Second, if all instructions of a code region have exactly the same assigned features, then the code region is obviously a feature region with said features. For all other cases in between, we require a decision boundary (i. e., a threshold).

We say that VaRA detected a code region R as feature region with the features X, Y, Z if the proportion of instructions that have annotated the features X, Y, Z to the remaining instructions is greater than or equal to the threshold T, according to the following formula:

$$\text{feature\_region}(R, F, T) = \frac{\#\text{ instructions in } R \text{ with features } F}{\#\text{ all instructions in } R} \geq \text{Threshold } T$$

This allows us to try out different thresholds for our analysis and examine their impact on the outcome.

4.2.2 *Mapping Features to Command-Line Options*

We can now determine for both approaches whether a code region is a feature region. However, a direct comparison of feature regions is not yet possible. The coverage-based baseline associates command-line options to its feature regions, while VaRA associates features from the feature model to its feature regions. That is, they are named differently. Additionally, abstract features have no corresponding command-line options on the coverage side. Thus, in order to compare command-line options to features, we need a mapping between command-line options and their corresponding features. We store this mapping explicitly as part of the feature model. To maintain the integrity of the results, the comparison implementation omits features that our coverage data has no information about, such as abstract features without a corresponding command-line option.

4.2.3 *Classification and Performance Assessment*

Now that we are able to decide if feature regions are affected by the same feature, we can compare the feature regions detected by our coverage-based baseline and VaRA. The actual comparison is a classification, that is, the coverage-based baseline is used as a reference to determine if the feature regions identified by VaRA are indeed feature regions. This is realized by iterating over all code regions and comparing the annotated features of both approaches as follows: For each feature and code region, we compare whether the two approaches consider that code region as feature region with the same feature.

There are four potential outcomes when comparing code regions. True positives (TPs) are code regions that VaRA correctly identifies as being affected by feature X. False positives (FPs) are code regions that VaRA associates with feature X but our coverage-based baseline does not. False negatives (FNs) are code regions that our coverage-based baseline associates with feature X while VaRA does not. Lastly, true negatives (TNs) refer to code regions that are not associated with feature X by both approaches. The four cases can be displayed as confusion matrix as shown in Table 4.1. We use the confusion matrix to calculate the metrics precision, recall, specificity, and balanced accuracy to assess the performance of VaRA's feature-region detection.

Table 4.1: Confusion matrix to classify results of VaRA's feature-region detection.

		VaRA's Feature-Region Detection	
		Feature Region	No Feature Region
Coverage-Baseline	Feature Region	True Positive (TP)	False Negative (FN)
	No Feature Region	False Positive (FP)	True Negative (TN)

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

$$\text{Precision} = \text{Positive Predictive Value (PPV)} = \frac{TP}{TP + FP} \tag{4.2}$$

$$\text{Recall} = \text{True Positive Rate (TPR)} = \frac{TP}{TP + FN} \tag{4.3}$$

$$\text{Specificity} = \text{True Negative Rate (TNR)} = \frac{TN}{TN + FP} \tag{4.4}$$

$$\text{Balanced Accuracy (BA)} = \frac{TPR + TNR}{2} \tag{4.5}$$

Accuracy (Equation 4.1) is the fraction of code regions correctly identified by VaRA's feature-region detection. However, accuracy alone cannot sufficiently assess the feature-region detection performance of VaRA as it is susceptible to imbalanced data. For instance, if there are 10 feature regions and 90 non-feature regions, but VaRA fails to detect any feature region, the accuracy would still be 90%. This looks promising at first, but in reality it means that VaRA's feature-region detection does not work at all! Thus, it is crucial to consider other metrics to gain further insights. For our purposes, precision (Equation 4.2) and recall (Equation 4.3) are the most important metrics to assess VaRA's feature-region detection performance, since they tell us how accurate feature regions get identified. Precision, also known as positive predictive value (PPV), is the fraction of feature regions identified by VaRA that were genuine feature regions. If VaRA does not classify code regions incorrectly as feature regions, then the precision is 100%. Recall, also known as true positive rate (TPR), measures the proportion of genuine feature regions identified as such by VaRA. If VaRA detects all actual feature regions, recall will reach 100%. As additional metric the specificity (Equation 4.4) can be considered. Specificity, also referred to as true negative rate (TNR), is the percentage of non-feature regions that are correctly identified as such by VaRA. It is related to the recall, as it measures the opposite case. To express recall and specificity together as one value, the balanced accuracy (Equation 4.5) is used. Balanced accuracy (BA) is the sum of recall and specificity divided by two to stay in the 0–100% range. In contrast to the normal accuracy, balanced accuracy is not susceptible to imbalanced data.

4.3 OPERATIONALIZATION

Here, we present the experiments we use to address our research questions. As we use the coverage-based baseline as ground truth for classification, we first need to find out how well it approximates VaRA's results. To do this, we conduct a qualitative analysis on reasonable examples, through which we can identify conceptual differences between the two approaches and learn how to mitigate them if possible. Subsequently, we conduct a quantitative analysis where we evaluate the performance of VaRA's feature-region detection on real-world programs and investigate anomalous results. The common procedure for both analyses and the analyses themselves are explained in the following sections.

4.3.1 *Experiment Design*

Our analyses follow the same experiment design. We use our comparison process to classify the results of VaRA's feature-region detection for a program and examine the outcome. We especially pay attention to false positives and false negatives, because in these cases the coverage-based baseline and VaRA's feature-region detection disagree.

To identify inconsistencies not only in detected feature regions, but also in individual instructions, we run the comparison process twice. Once with the threshold $> 0\%$ and another time with the threshold 100%. The former means that a single instruction annotated with a feature makes the code region a feature region, while the latter means that all instructions must have this feature annotated, such that the code region is considered as a feature region. In this way, we can detect flaky feature regions, that is, code regions that we cannot unambiguously assign, by the differences in the corresponding confusion matrices.

We investigate the differences we find in detail to decide whether there is a bug in the implementation of one approach that can be fixed, or whether this is intended behavior. This enables us to validate the results of VaRA's feature-region detection step-wise, or to figure out where problems exist and how we can address them.

4.3.2 *Qualitative Analysis*

The goal of our qualitative analysis is to identify conceptual differences between the feature regions found by VaRA and our coverage-based baseline and, if possible, mitigate their impact on the results. Through this analysis we answer RQ1. By comprehending the differences between the two approaches, we can assess how well VaRA's feature-region detection results can be approximated by our coverage-based baseline.

To identify conceptual differences, analyzing small but reasonable examples is sufficient. We have chosen the example programs MultiSharedMultipleRegions (MSMR) and SimpleFeatureInteraction (SFI) from VaRA's test suite for this purpose. MSMR has four feature variables that are stored in different ways and checked independently one after the other. Therefore, it is well-suited for detecting basic differences. For edge cases, such as feature interactions, we analyze SFI. It resembles the feature interaction in our ZIP example. In fact, our ZIP example is based on SFI. An overview of the analyzed programs is available in Table 4.2.

Since the data we are analyzing is manageable, we manually verify it by tracking the classification of each code region. This ensures that we can rule out errors in our implementation and do not overlook any differences that arise in our example programs.

Table 4.2: Examined example programs[1] in qualitative analysis.

Language	Program	# Configurations	# Features	Commit
C++	MultiSharedMultipleRegions	16	4	`4300ea495e`
	SimpleFeatureInteraction	4	2	

4.3.3 *Quantitative Analysis*

The goal of the quantitative analysis is to assess the performance of VaRA's feature-region detection by examining the performance metrics, precision, and recall and interpreting them based on the insights gained from the qualitative analysis. This allows us to answer RQ2. Additionally, we analyze in detail any results that cannot be explained by the previous findings. That enables us to identify further conceptual differences between the two approaches, reveal bugs in their implementation, or find potential areas for improvement.

We evaluate the performance of VaRA's feature-region detection on the real-world C programs Bzip2, gzip, and XZ. These are well-known compression tools that are also part of previous research [14, 27]. All three programs have similar features, but they implement some of them differently. For instance, Bzip2 uses macros to implement its various operation modes, while gzip uses an `int` variable, and XZ uses enums. Therefore, they are well-suited for comparing VaRA's feature-region detection performance between similar features or assessing the detection performance of different feature implementations. Originally, we planned to analyze the SAT solver PicoSAT and the file optimizer ECT. PicoSAT is of particular interest because it differs from the previous real-world programs as it does not come from the compression domain and therefore has distinct features. ECT is also of interest to us because it is implemented in C++, which allows different programming patterns than C, such as object-oriented programming. However, both programs use fields in structs as an essential part of their program logic. For the case of ECT, the entire load-time configuration (i. e., all feature variables) are saved in one struct. At the time of our testing, VaRA's field-sensitivity support, which is necessary to track individual fields of a struct in LLVM IR, was not mature enough to be tested on real-world programs. Therefore, we abstain from analyzing PicoSAT and ECT to avoid incomplete or falsified results. As compensation, we analyze six of VaRA's synthetic benchmarks that test feature-region detection in various real-world scenarios. We provide an overview of the real-world programs and synthetic benchmarks analyzed in Table 4.3.

1 https://github.com/se-sic/FeaturePerfCSCollection/tree/master/src (visited on January 16, 2024)

Table 4.3: Examined real-world programs and synthetic benchmarks[2] quantitative analysis.

Language	Program	# Configurations	# Features	Commit
C	Bzip2	96	8	`1ea1ac188a`
	gzip	448	10	`23a870d14a`
	XZ	160	13	`4773608554`
	~~PicoSAT~~	1024	11	`33c685e822`
C++	~~ECT~~	2304	12	`e98502a01b`
	SynthDADynamicDispatch	3	3	`04de0642af`
	SynthDARecursion	4	2	`daf81de073`
	SynthIPRuntime	5	4	`7930350628`
	SynthOVInsideLoop	4	2	`51d3c768e5`
	SynthSAContextSensitivity	12	2	`06eac0edb6`
	SynthSAFlowSensitivity	4	2	

[2] https://github.com/se-sic/FeaturePerfCSCollection/tree/master/projects (visited on January 16, 2024)

5 EVALUATION

In this chapter, we perform the qualitative and quantitative analysis described in Section 4.3 to answer the research questions. Subsequently, we derive conclusions from the obtained results and discuss them in relation to our thesis goal: Does VaRA's feature-region detection yield valid results? Lastly, we disclose potential threats to result validity and our efforts to mitigate them.

5.1 RESULTS

In this section, we first report the conceptual differences between our coverage-based baseline and VaRA's feature-region detection that we found through our qualitative analysis. We then present the results of our quantitative analysis on the precision and recall of VaRA's feature-region detection on real-world programs. All results are based on the VaRA version `437b4a40c1`, which was the most recent commit at the time of our testing.

5.1.1 *Qualitative Analysis*

For our two example programs, MSMR and SFI, we conduct a qualitative analysis to answer our first research question:

> **RQ1:** What conceptual differences in detected feature regions exist between the coverage-based baseline and VaRA's feature-region detection?

To achieve this, we classify the detected feature regions of VaRA with these from the coverage-based baseline. The results can be seen in Table 5.1 and Table 5.2. Our tables have two rows per feature, one for each threshold. Each row displays the confusion matrix classifications and their corresponding metrics for the feature and threshold. The classifications are listed by count (TP, FN, FP, TN), while the metrics precision (PPV), recall (TPR), specificity (TNR), and balanced accuracy (BA) are presented as percentages. The total rows at the end of the table provide a summary of all features. They are calculated by summing the count of each classification for the corresponding threshold and then using these sums to compute the performance metrics.

Our first observation is that the values for the two thresholds do not differ. This means that VaRA's feature-region detection marks either all instructions in a code region as feature-dependent or none. Consequently, MSMR and SFI do not have flaky feature regions. In the following tables, we therefore omit the threshold if it is not relevant.

However, both tables show false negatives and false positives. That is, the coverage-based baseline and VaRA's feature-region detection find different feature regions. In the following sections, we examine these differences in detail.

Table 5.1: Results for MSMR without conceptual differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
cpp	>0 %	1	7	1	57	50.00	12.50	98.28	55.39
	100 %	1	7	1	57	50.00	12.50	98.28	55.39
extern	>0 %	1	7	1	57	50.00	12.50	98.28	55.39
	100 %	1	7	1	57	50.00	12.50	98.28	55.39
header	>0 %	2	6	1	57	66.67	25.00	98.28	61.64
	100 %	2	6	1	57	66.67	25.00	98.28	61.64
slow	>0 %	2	6	1	57	66.67	25.00	98.28	61.64
	100 %	2	6	1	57	66.67	25.00	98.28	61.64
TOTAL	>0 %	6	26	4	228	60.00	18.75	98.28	58.51
	100 %	6	26	4	228	60.00	18.75	98.28	58.51

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

Table 5.2: Results for SFI without conceptual differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
compress	>0 %	2	8	2	38	50.00	20.00	95.00	57.50
	100 %	2	8	2	38	50.00	20.00	95.00	57.50
enc	>0 %	3	8	1	38	75.00	27.27	97.44	62.35
	100 %	3	8	1	38	75.00	27.27	97.44	62.35
TOTAL	>0 %	5	16	3	76	62.50	23.81	96.20	60.01
	100 %	5	16	3	76	62.50	23.81	96.20	60.01

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

5.1.1.1 *Conceptual differences*

Some of these differences can be attributed to conceptual differences between the coverage-based baseline and VaRA's feature-region detection. Because of their conceptual nature, we can devise mitigations that exclude these differences from our results. We identify the following conceptual differences and mitigate them.

CONDITION HANDLING The false positives in the results are exclusively control flow decisions in which the corresponding feature variable is checked, as shown in Listing 5.1. From the perspective of VaRA, these conditions are part of the feature regions by default, since all instructions in them depend on the feature variable. However, they are not part of the feature regions from the perspective of our coverage-based baseline, because the instructions are always executed regardless of the feature variable. Both views are valid, as this is a conceptual difference in how the two approaches operate. To mitigate this conceptual difference in our analysis results, we account for it as follows. VaRA is able to tell us whether instructions are part of a condition. We use this information in our *FeatureDebugLocationExport* LLVM pass to mark the feature introduced by the condition in instructions belonging to the condition with a prefix. During classification, we then simply ignore the features found by VaRA with such a prefix. In this way, we align the condition handling of VaRA with that of our coverage-based baseline. This mitigation results in the conditions being classified as true negatives, as shown in Table 5.3, which increases the precision and specificity of VaRA's feature-region detection results for MSMR and SFI to 100 %.

		Coverage	VaRA
1	`bool UseCompression;`		
2	`bool UseEncryption;`		
3			
4	`if (UseCompression) {`		
5	`  Data = compress(Data);`	*compress*	*Compression*
6	`}`		
7	`if (UseEncryption) {`	*enc*	*Encryption*
8	`  if (not UseCompression) {`		
9	`    Data = addPadding(Data);`	$\neg$*compress* $\wedge$ *enc*	*Encryption*, *Compression*
10	`  }`		
11		*enc*	*Encryption*
12	`  Data = encrypt(Data);`		
13	`}`		

Listing 5.1: ZIP example, feature region difference: condition handling. The conditions highlighted using darker colors are considered as feature region by VaRA's feature-region detection but not by the coverage-based baseline.

OPTION PARSING CODE As can be seen in Table 5.3, MSMR and SFI still exhibit a significant number of false negatives, that is, code regions that are only identified as feature regions by the

Table 5.3: Threshold-independent results for MSMR and SFI with condition handling differences mitigated.

Program	Feature	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
MSMR	cpp	1	7	0	58	100.00	12.50	100.00	56.25
	extern	1	7	0	58	100.00	12.50	100.00	56.25
	header	2	6	0	58	100.00	25.00	100.00	62.50
	slow	2	6	0	58	100.00	25.00	100.00	62.50
	TOTAL	6	26	0	232	100.00	18.75	100.00	59.38
SFI	compress	2	8	0	40	100.00	20.00	100.00	60.00
	enc	3	8	0	39	100.00	27.27	100.00	63.64
	TOTAL	5	16	0	79	100.00	23.81	100.00	61.90

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

coverage-based baseline. Most of these false negatives originate from the command-line option parsing code. As illustrated in Listing 5.2, the coverage-based baseline identifies code regions that depend on the command-line option as feature regions, while VaRA's feature-region detection only finds feature regions that depend on the feature variable. VaRA's detection method is not conceptually capable of recognizing feature regions in command-line option parsing code, since the feature variables are only assigned their value provided from the outside (e. g., via command-line arguments) there, but are not used. Static analysis cannot detect individual features earlier, as this is the earliest point in the information flow from `argv` into the feature variables where the features become distinguishable. However, the command-line option parsing code is usually relatively easy to identify by hand in most programs. Therefore, we manually locate the parsing code and ignore its code regions during classification for our further analysis. This reduces the number of false negatives in MSMR to zero, as shown in Table 5.4. For MSMR, the coverage-based baseline and VaRA's feature-region detection now agree on the detected feature regions.

FEATURE-DEPENDENT FUNCTIONS By applying the previous mitigations, the number of false negatives in SFI has been significantly reduced, as can be seen in Table 5.4. The remaining false negatives originate from feature-dependent functions, that is, functions that are exclusively called within feature regions. As demonstrated in Listing 5.3, the coverage-based baseline as a dynamic analysis can identify these, whereas VaRA's feature-region detection with its static approach currently cannot. VaRA could attempt to construct a call graph to determine if a function is solely invoked in feature regions. This approach would work for the static calls in our SFI example, but the effort would be only useful to a limited extent. If dynamic calls are added through pointers or language features like dynamic dispatch, where a decision regarding which implementation to use is made at run time, the static call graph is not useful. In fact, it might worsen the results of VaRA's feature-region detection

```
1  bool HelloWorld = false;
2
3  if (isFeatureEnabled(argc, argv, std::string("--hello-world"))){
4    HelloWorld = true;
5  }
6
7  if (HelloWorld){
8    std::cout << "Hello ";
9  }
10
11 if (isFeatureEnabled(argc, argv, std::string("--hello-world"))){
12   std::cout << "World!";
13 }
```

Coverage	VaRA
✓	✗
✓	✓
✓	✗

Listing 5.2: Difference in detected command-line option dependent and feature variable dependent feature regions.

Table 5.4: Threshold-independent results for MSMR and SFI with condition handling and parsing code differences mitigated.

Program	Feature	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
MSMR	cpp	1	0	0	48	100.00	100.00	100.00	100.00
	extern	1	0	0	48	100.00	100.00	100.00	100.00
	header	2	0	0	47	100.00	100.00	100.00	100.00
	slow	2	0	0	47	100.00	100.00	100.00	100.00
	TOTAL	6	0	0	190	100.00	100.00	100.00	100.00
SFI	compress	2	2	0	28	100.00	50.00	100.00	75.00
	enc	3	2	0	27	100.00	60.00	100.00	80.00
	TOTAL	5	4	0	55	100.00	55.56	100.00	77.78

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

if a feature-dependent function is called both statically and dynamically. We handle this conceptual difference similarly to the previous one, by manually detecting and ignoring feature-dependent functions. In this way, we also achieve a complete agreement in identified feature regions of both approaches for SFI, as can be seen in Table 5.5.

```
1  bool UseCompression;
2
3  PackageData compress(PackageData Data){
4    ...
5  }
6  ...
7  if (UseCompression){
8    Data = compress(Data);
9  }
```

Coverage	VaRA
✓	✗
✓	✓

Listing 5.3: Feature-dependent functions are detected as feature-region by the coverage-based baseline but not by VaRA's feature-region detection.

Table 5.5: Threshold-independent results for SFI with condition handling, parsing code, and feature-dependent function differences mitigated.

Feature	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
compress	2	0	0	27	100.00	100.00	100.00	100.00
enc	3	0	0	26	100.00	100.00	100.00	100.00
TOTAL	5	0	0	53	100.00	100.00	100.00	100.00

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

5.1.1.2 *Flaky feature regions*

During the evaluation of both synthetic and real-world programs for RQ2, we encountered flaky feature regions, that is, feature regions whose detection depends on the threshold. We describe these flaky feature regions here because they reveal more subtle differences between the two feature detection approaches.

CLASSES To overcome the limitation of static analysis in dynamic dispatching, VaRA offers an additional feature detection process called *FunctionBasedFeatureDetection*. This pass can be enabled with the `-vara-FBFD` command-line option and allows us to assign features to classes or functions. We make use of this functionality to assign the three classes, `BrutForceSearcher`, `SortingSearcher`, and `HashSearcher`, in SynthDADynamicDispatch their respective feature. As

a result, VaRA's feature-region detection also annotates the instructions belonging to the class with the feature. There is a high agreement in detected feature regions, as shown in Table 5.6. However, every feature has a flaky false positive, namely the root of our code region tree. The root code region is flaky because it contains some of the feature-dependent class initialization instructions together with other non-feature-dependent instructions. Additionally, it is a flaky false positive because, from the perspective of the coverage-based baseline, the root code region cannot be feature-dependent since its instructions are always executed. Hence, whenever VaRA marks a class as feature-dependent, we will observe it as a flaky false positive in our comparison results.

Table 5.6: Results for SynthDADynamicDispatch with condition handling, parsing code, and feature-dependent function differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
brut-force	>0 %	11	0	1	71	91.67	100.00	98.61	99.31
	100 %	11	0	0	72	100.00	100.00	100.00	100.00
hashing	>0 %	9	0	1	73	90.00	100.00	98.65	99.32
	100 %	9	0	0	74	100.00	100.00	100.00	100.00
sorting	>0 %	10	0	1	72	90.91	100.00	98.63	99.32
	100 %	10	0	0	73	100.00	100.00	100.00	100.00
TOTAL	>0 %	30	0	3	216	90.91	100.00	98.63	99.32
	100 %	30	0	0	219	100.00	100.00	100.00	100.00

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

SHORT-CIRCUITING CONDITIONS Another cause of flaky feature regions are short-circuiting conditions, that is, conditions with short-circuiting operators like the logical AND (`&&`) or the logical OR (`||`) operator. The term "short-circuit" pertains to their behavior of ceasing the evaluation of other operands once the final outcome is fixed. Such a short-circuit condition occurs, for example, in SynthIPRuntime and causes the threshold difference in Table 5.7. The relevant code is displayed in Listing 5.4. As we can see, in the threshold >0 % case the `smallmode` variable in Line 2 gets assigned the smallmode feature by VaRA, even though we actually ignore this code region through our condition handling mitigation. This issue is caused by the last instruction in Listing 5.5, which LLVM still attributes to the first operand (Column 7), although the `br` instruction evaluates the second operand, the `decompress` variable (Column 20). That means, in a basic block, there may be `br` instructions that belong to different code regions than the rest of the basic block. This cannot be easily mitigated by us.

TERNARY OPERATORS The same problem of counting the location of the second `br` instruction towards the beginning of the condition also causes threshold differences in SynthSAFlowSensitivity, as can be seen in Table 5.8. However, this time it is caused in

Table 5.7: Results for SynthIPRuntime with condition handling, parsing code, and feature-dependent function differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
1	>0 %	2	0	0	84	100.00	100.00	100.00	100.00
	100 %	2	0	0	84	100.00	100.00	100.00	100.00
2	>0 %	5	0	1	80	83.33	100.00	98.77	99.38
	100 %	5	0	0	81	100.00	100.00	100.00	100.00
c	>0 %	6	11	15	54	28.57	35.29	78.26	56.78
	100 %	6	11	15	54	28.57	35.29	78.26	56.78
d	>0 %	11	1	3	71	78.57	91.67	95.95	93.81
	100 %	11	1	3	71	78.57	91.67	95.95	93.81
TOTAL	>0 %	24	12	19	289	55.81	66.67	93.83	80.25
	100 %	24	12	18	290	57.14	66.67	94.16	80.41

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy
1 = fastmode, 2 = smallmode, c = compress, d = decompress

		Coverage	VaRA
1	`int getBufsize(bool smallmode, bool decompress){`		
2	`  if (smallmode || decompress){`	*True* || ¬2 ∨ ¬*c*	
3	`    return 100 * 1024;`	2 ∨ ¬*c*	*smallmode*
4	`  }`		
5	`  return 10 * 1024 * 1024;`	¬2 ∧ *c*	*smallmode*,
6	`}`		*decompress*

Listing 5.4: Threshold difference in SynthIPRuntime marked in red.

		Line	Column	VaRA
1	`entry:`			
2	`  ...`			
3	`  %0 = load i8, i8* %smallmode.addr, align 1`	2	7	
4	`  %tobool = trunc i8 %0 to i1`	...	...	
5	`  br i1 %tobool, label %if.then, label %lor.lhs.false`		17	
6			...	
7	`lor.lhs.false: ;preds = %entry`			
8	`  %1 = load i8, i8* %decompress.addr, align 1`		20	
9	`  %tobool2 = trunc i8 %1 to i1`		...	*smallmode*
10	`  br i1 %tobool2, label %if.then, label %if.end`		7	

Listing 5.5: Threshold difference in SynthIPRuntime's LLVM IR.

a different place, namely the ternary operator. The relevant part of the program is depicted in Listing 5.6.

Table 5.8: Results for SYNTHSAFLOWSENSITIVITY with condition handling, parsing code, and feature-dependent function differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
cutoff 4	>0 %	2	0	12	52	14.29	100.00	81.25	90.62
	100 %	2	0	10	54	16.67	100.00	84.38	92.19
rec	>0 %	2	6	0	58	100.00	25.00	100.00	62.50
	100 %	2	6	0	58	100.00	25.00	100.00	62.50
TOTAL	>0 %	4	6	12	110	25.00	40.00	90.16	65.08
	100 %	4	6	10	112	28.57	40.00	91.80	65.90

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

```
1 CutOff = CutOff < MAX_CUTOFF ? CutOff : MAX_CUTOFF;
2 unsigned TargetN = UserInput > CutOff ? CutOff : UserInput;
```

Listing 5.6: Threshold difference in SYNTHSAFLOWSENSITIVITY marked in red.

EARLY RETURNS The previous feature regions are flaky because only a few instructions have a feature annotated, but the rest has not. We discover the opposite case in SYNTHDARECURSION for the limit option, as can be seen in Table 5.9. In SYNTHDARECURSION the limit option limits the recursion depth of a function similar to the one depicted in Listing 5.7. In this recursive function the last feature region is flaky, but the cause for this can only be seen in the LLVM IR in Listing 5.8. The instruction `ret void` (Line 9) belongs to the feature region located in Lines 5–7 of the source code. However, it is correctly not annotated with the *RecursionLimit* feature, as it can be reached from both the `if.then` and `if.end` blocks, while the former is not feature-dependent. The `if.then` block can also be reached through the `N == 0` condition, which is also not feature-dependent. This conceptual difference is not restricted to recursive functions. It may also affect normal functions that return from feature regions as well as normal regions.

5.1.1.3 *Summary*

As shown above, the feature-region detection of the coverage-based baseline and VARA differ conceptually in the detection of conditions, command-line option parsing related code, and feature-dependent functions. Additionally, flaky feature regions may arise from classes, short-circuiting conditions, ternary operators, and early returns.

Table 5.9: Results for SynthDARecursion with condition handling, parsing code, and feature-dependent function differences mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
limit `0`	>0 %	8	1	0	35	100.00	88.89	100.00	94.44
	100 %	7	2	0	35	100.00	77.78	100.00	88.89
rev	>0 %	1	0	0	43	100.00	100.00	100.00	100.00
	100 %	1	0	0	43	100.00	100.00	100.00	100.00
TOTAL	>0 %	9	1	0	78	100.00	90.00	100.00	95.00
	100 %	8	2	0	78	100.00	80.00	100.00	90.00

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

```
1 void recursion(vector<int> &Result, int N, int Depth = 0){
2   if (N == 0 || Depth == RecLimit){
3     return;
4   }
5   Result.push_back(N)
6   recursion(Result, N * N, Depth + 1);
7 }
```

Coverage	VaRA
True	
¬limit 0	*RecursionLimit*

Listing 5.7: Threshold difference in SynthDARecursion visualized as simplified recursive function.

```
1 if.then: ;preds = %lor.lhs.false, %entry
2   br label %return
3
4 if.end: ;preds = %lor.lhs.false, %if.then
5   ...
6   br label %return
7
8 return: ;preds = %if.end, %if.then
9   ret void
```

Line	Column	VaRA
3	5	
5–7	-	*RecursionLimit*
7	1	
7	1	

Listing 5.8: Threshold difference in SynthDARecursion's LLVM IR.

Table 5.10: Totals for synthetic programs.

Program	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
DADynamicDispatch	>0 %	30	0	3	216	90.91	100.00	98.63	99.32
	100 %	30	0	0	219	100.00	100.00	100.00	100.00
DARecursion	>0 %	9	1	0	78	100.00	90.00	100.00	95.00
	100 %	8	2	0	78	100.00	80.00	100.00	90.00
IPRuntime	>0 %	24	12	19	289	55.81	66.67	93.83	80.25
	100 %	24	12	18	290	57.14	66.67	94.16	80.41
OVInsideLoop	>0 %	2	0	0	102	100.00	100.00	100.00	100.00
	100 %	2	0	0	102	100.00	100.00	100.00	100.00
SAContextSensitivity	>0 %	24	102	32	255	42.86	19.05	88.85	53.95
	100 %	24	102	32	255	42.86	19.05	88.85	53.95
SAFlowSensitivity	>0 %	4	6	12	110	25.00	40.00	90.16	65.08
	100 %	4	6	10	112	28.57	40.00	91.80	65.90

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

5.1.2 *Quantitative Analyis*

We conduct a quantitative analysis on synthetic benchmarks and real-world programs in order to address RQ2:

> **RQ2:** How well does VaRA's feature-region detection perform on real-world programs?
> **RQ2.1:** What is the fraction of detected feature regions that are genuine? (Precision)
> **RQ2.2:** What is the fraction of genuine feature regions that are detected as such? (Recall)

The results of the synthetic benchmarks are presented in Table 5.10. For the real-world programs, the results are shown in Table 5.11, 5.12, and 5.13. However, for the latter, we no longer mitigate the feature-dependent functions, as they cannot be clearly identified by hand anymore.

To assess performance, we need to take a closer look at precision and recall. Since we found that some types of features are recognized better than others, we evaluate RQ2.1 and RQ2.2 per feature group. The group membership of the features is indicated in the real-world program tables by the superscript number at the feature name.

Table 5.11: Results for Bzip2 with condition handling, parsing code differences and alternative group issues mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
1^3	>0 %	0	0	60	5558	0.00	-	98.93	-
	100 %	0	0	56	5562	0.00	-	99.00	-
9^3	>0 %	0	1	60	5557	0.00	0.00	98.93	49.47
	100 %	0	1	56	5561	0.00	0.00	99.00	49.50
compress[1]	>0 %	0	1050	0	4568	-	0.00	100.00	50.00
	100 %	0	1050	0	4568	-	0.00	100.00	50.00
decompress[1]	>0 %	0	696	0	4922	-	0.00	100.00	50.00
	100 %	0	696	0	4922	-	0.00	100.00	50.00
force[6]	>0 %	4	10	32	5572	11.11	28.57	99.43	64.00
	100 %	4	10	27	5577	12.90	28.57	99.52	64.04
keep[6]	>0 %	6	0	2	5610	75.00	100.00	99.96	99.98
	100 %	6	0	0	5612	100.00	100.00	100.00	100.00
quiet[5]	>0 %	0	0	19	5599	0.00	-	99.66	-
	100 %	0	0	18	5600	0.00	-	99.68	-
small[6]	>0 %	2	136	45	5435	4.26	1.45	99.18	50.31
	100 %	2	136	41	5439	4.65	1.45	99.25	50.35
test[1]	>0 %	0	615	0	5003	-	0.00	100.00	50.00
	100 %	0	615	0	5003	-	0.00	100.00	50.00
TOTAL	>0 %	12	2508	218	47824	5.22	0.48	99.55	50.01
	100 %	12	2508	198	47844	5.71	0.48	99.59	50.03

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy, Feature^n = Group n

Table 5.12: Results for GZIP with condition handling, parsing code differences and alternative group issues mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
6^3	>0 %	0	2	93	4075	0.00	0.00	97.77	48.88
	100 %	0	2	90	4078	0.00	0.00	97.84	48.92
9^3	>0 %	1	3	92	4074	1.08	25.00	97.79	61.40
	100 %	1	3	89	4077	1.11	25.00	97.86	61.43
decompress^4	>0 %	370	474	578	2748	39.03	43.84	82.62	63.23
	100 %	342	502	538	2788	38.86	40.52	83.82	62.17
force^6	>0 %	14	0	872	3284	1.58	100.00	79.02	89.51
	100 %	14	0	809	3347	1.70	100.00	80.53	90.27
keep^6	>0 %	13	11	6	4140	68.42	54.17	99.86	77.01
	100 %	13	11	5	4141	72.22	54.17	99.88	77.02
list^4	>0 %	2	81	118	3969	1.67	2.41	97.11	49.76
	100 %	2	81	114	3973	1.72	2.41	97.21	49.81
no-name^6	>0 %	7	5	732	3426	0.95	58.33	82.40	70.36
	100 %	6	6	677	3481	0.88	50.00	83.72	66.86
quiet^5	>0 %	1	1	59	4109	1.67	50.00	98.58	74.29
	100 %	1	1	40	4128	2.44	50.00	99.04	74.52
stdout^6	>0 %	43	131	97	3899	30.71	24.71	97.57	61.14
	100 %	43	131	93	3903	31.62	24.71	97.67	61.19
test^4	>0 %	8	508	25	3629	24.24	1.55	99.32	50.43
	100 %	8	508	19	3635	29.63	1.55	99.48	50.52
verbose^5	>0 %	16	5	27	4122	37.21	76.19	99.35	87.77
	100 %	16	5	23	4126	41.03	76.19	99.45	87.82
TOTAL	>0 %	475	1221	2699	41475	14.97	28.01	93.89	60.95
	100 %	446	1250	2497	41677	15.15	26.30	94.35	60.32

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy, Feature^n = Group n

Table 5.13: Results for XZ with condition handling, parsing code differences and alternative group issues mitigated.

Feature	Threshold	TP	FN	FP	TN	PPV (%)	TPR (%)	TNR (%)	BA (%)
6^{3}	>0 %	0	639	0	3139	-	0.00	100.00	50.00
	100 %	0	639	0	3139	-	0.00	100.00	50.00
9^{3}	>0 %	0	0	0	3778	-	-	100.00	-
	100 %	0	0	0	3778	-	-	100.00	-
$\text{compress}^{2/4}$	>0 %	97	241	269	3171	26.50	28.70	92.18	60.44
	100 %	91	247	259	3181	26.00	26.92	92.47	59.70
$\text{decompress}^{2/4}$	>0 %	80	215	195	3288	29.09	27.12	94.40	60.76
	100 %	77	218	186	3297	29.28	26.10	94.66	60.38
force^{6}	>0 %	14	496	13	3255	51.85	2.75	99.60	51.17
	100 %	13	497	10	3258	56.52	2.55	99.69	51.12
format=auto^{4}	>0 %	0	4	18	3756	0.00	0.00	99.52	49.76
	100 %	0	4	18	3756	0.00	0.00	99.52	49.76
format=xz^{4}	>0 %	3	14	167	3594	1.76	17.65	95.56	56.60
	100 %	3	14	162	3599	1.82	17.65	95.69	56.67
keep^{6}	>0 %	6	175	10	3587	37.50	3.31	99.72	51.52
	100 %	6	175	10	3587	37.50	3.31	99.72	51.52
$\text{list}^{2/4}$	>0 %	4	231	273	3270	1.44	1.70	92.29	47.00
	100 %	2	233	261	3282	0.76	0.85	92.63	46.74
robot^{6}	>0 %	16	127	25	3610	39.02	11.19	99.31	55.25
	100 %	16	127	23	3612	41.03	11.19	99.37	55.28
stdout^{6}	>0 %	28	488	58	3204	32.56	5.43	98.22	51.82
	100 %	28	488	53	3209	34.57	5.43	98.38	51.90
suffix=.su^{6}	>0 %	4	24	16	3734	20.00	14.29	99.57	56.93
	100 %	4	24	16	3734	20.00	14.29	99.57	56.93
$\text{test}^{2/4}$	>0 %	71	106	204	3397	25.82	40.11	94.33	67.22
	100 %	68	109	195	3406	25.86	38.42	94.58	66.50
v^{5}	>0 %	26	166	30	3556	46.43	13.54	99.16	56.35
	100 %	25	167	27	3559	48.08	13.02	99.25	56.13
TOTAL	>0 %	349	2926	1278	48339	21.45	10.66	97.42	54.04
	100 %	333	2942	1220	48397	21.44	10.17	97.54	53.85

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy, Feature^{n} = Group n

5.1.2.1 *Features implemented as preprocessor macros*

The first group of features we would like to discuss are those that are defined through preprocessor macros. In Table 5.11 this applies to the features *compress*, *decompress* and *test*. These have no precision and a recall of 0 % because they are only recognized by the coverage-based baseline. VaRA is currently unable to assign macros to a feature as macro expansion occurs before the creation of LLVM IR and thus before VaRA is able to mark its feature variables. In principle, it should be possible in LLVM to associate LLVM IR instructions with a macro, since the code regions in LLVM's coverage JSON contain information about the macro from which they are expanded. However, it is debatable whether the effort would be justified because as can be seen in Listing 5.9, VaRA's feature-region detection recognizes the abstract feature *opMode* for the corresponding feature regions. Since the feature-dependent macros are always compared to the `opMode` feature variable, it is currently possible to detect operation mode dependent feature regions in Bzip2, albeit not that granular.

```
1 /*-- operation modes --*/                          | Coverage | VaRA
2 #define OM_Z       1 //compress                   |          |
3 #define OM_UNZ     2 //decompress                 |          |
4 #define OM_TEST    3 //test                       |          |
5 Int32 opMode; //either OM_Z, OM_UNZ or OM_TEST     |          |
6 ...                                                |          |
7 if (opMode == OM_Z){                               |          |
8   ...                                              | compress | OpMode
9 }                                                  |          |
```

Listing 5.9: Features as preprocessor macros in Bzip2.

5.1.2.2 *Features implemented as enum values*

The second group contains features whose feature variables are part of an enum. This is the case for the features *compress*, *decompress*, *list*, and *test* in Table 5.13. These features have a high number of false positives compared to the other features. As can be seen in Listing 5.10, this is because VaRA's feature region detection often detects all the features in an enum for a feature region. As a result, the precision is not as high as it could be. There are feature regions similar to the one in Lines 9–10 where VaRA only recognizes *Compr* and *Mode* as expected. However, we could not identify any differences in the corresponding instructions that could cause this varying behavior.

Furthermore, VaRA's feature-region detection has identified alleged feature regions that are definitely not feature regions. For instance, the function `hardware_threads_set` in XZ, as illustrated in Listing 5.11, is not a feature region. Here, the variable `n` seems to depend on the `operation_mode` enum, which probably causes that all features of the enum are annotated to the code region in Lines 3–5. We have traced the information flow to `n` in the LLVM IR, but without finding an explanation. In total, there are three instances where `hardware_threads_set`

```
1  enum operation_mode {
2    MODE_COMPRESS, //Compr
3    MODE_DECOMPRESS, //Decompr
4    MODE_TEST, //Test
5    MODE_LIST, //Info
6  };
7  enum operation_mode opt_mode; //Mode
8  ...
9  if (opt_mode == MODE_COMPRESS){
10   ...
11 }
```

Coverage	VaRA
compress	*Compr*, *Decompr*, *Info*, *Mode*, *Test*

Listing 5.10: Feature variables as enum in XZ.

```
1  extern void
2  hardware_threads_set(uint32_t n){
3    if (n == 0){
4     ...
5    }
6  }
7  ...
8  hardware_threads_set(str_to_uint64("threads", optarg, ...));
9  ...
10 hardware_threads_set(1);
```

VaRA
Compr, *Decompr*, *Info*, *Test*
Compr, *Mode*

Listing 5.11: Wrongly annotated feature region in XZ.

is called: one in the parsing code at Line 8, and two more from feature regions annotated with *Compr* and *Mode*, shown in Line 10. The passed argument is not feature-dependent for any call. Therefore, it appears to be a bug in VaRA's feature-region detection or the underlying feature-taint analysis, which may also trigger the behavior shown in Listing 5.10, at the expense of precision and recall.

5.1.2.3 *Features with numeric arguments*

Thirdly, we discuss features with numeric arguments as used in Bzip2 (Table 5.11), gzip (Table 5.12) and XZ (Table 5.13). There, the compression level can be adjusted using the numerical options `-1` to `-9`, a tradeoff between computational effort and resulting file size. Both the coverage-based baseline and VaRA's feature-region detection had difficulties identifying these features.

In the real-world programs we test, numeric options are primarily utilized to determine the number of times a loop should be executed (GZIP and XZ) or the block size into which the data should be split for compression (BZIP2). Both cases have little influence on the executed code regions in the coverage data. Consequently, our coverage-based baseline is unable to identify these usages of numeric options. Precision and recall, therefore, hold little significance, and we must examine the false negatives and false positives to evaluate VARA's feature-region detection performance.

For BZIP2 and GZIP, there exist false positives, which means that VARA identifies feature regions for the compression level. We analyzed these feature regions and discovered that they all depend on the compression level variable. Hence, they are legitimate feature regions.

Only in XZ, VARA does not find the compression level, as we see from the missing false positives and true positives. However, this is not caused by a design flaw, but by missing instructions in the LLVM IR that prevent continuous tracking of the feature variable's information flow. Specifically, this affects the function `lzma_lzma_preset`, which is called with the feature variable `preset_number` as an argument, as seen in Listing 5.12. One would anticipate that the output also depends on `preset_number`, and therefore the feature region in Lines 5–7 is found. In the LLVM IR, however, the function is only declared as in Listing 5.13. The complete body of the function is absent. This is because the implementation of `lzma_lzma_preset` occurs in the library LIBLZMA, which is linked dynamically instead of statically by default. By changing this behavior, VARA's feature taint analysis should be capable of tracking information flow through the LIBLZMA library.

```
1 extern LZMA_API(lzma_bool)
2 lzma_lzma_preset(lzma_options_lzma *options, uint32_t preset)
3 ...
4
5 if (lzma_lzma_preset(&opt_lzma, preset_number)) {
6   // Should be detected as feature region.
7 }
```

Listing 5.12: Definition and usage of `lzma_lzma_preset`.

```
1 ; Function Attrs: nounwind
2 declare i8 @lzma_lzma_preset(%struct.lzma_options_lzma* noundef, i32 noundef) #3
```

Listing 5.13: Declaration of `lzma_lzma_preset` in LLVM IR.

The numerous false negatives observed for compression level `-6` in XZ are due to the fact that `-6` is the default compression level for XZ. Since the compression level is a mandatory feature in XZ's feature model, all configurations have one specified as command-line option. Even configurations that activate other modes such as *decompress*, *list*, and *test*, although they do not compress, have `-6` specified. Therefore, our coverage-based baseline assumes that these feature regions also depend on compression level `-6`. However, this is not the case.

5.1.2.4 *Features in an alternative group*

The fourth group contains features that are part of an alternative group in the feature model, that is, only one feature in this group can be enabled at a time. This applies to GZIP and

XZ, where the different operation modes (*compress*, *decompress*, *list*, and *test*) fall under this category. It would make no sense to compress and decompress data at the same time. In Table 5.12 and 5.13, we observe that these features exhibit high levels of precision and recall simultaneously. Therefore, there is a relatively high agreement on them between the coverage-based baseline and VaRA's feature region-detection. Nonetheless, these particular features have a high number of false negatives and false positives compared to the other features. The reason for this is that these features make up larger parts of the program than the other features, which means that the conceptual differences between VaRA and the coverage-based baseline also accumulate. Some of these differences may originate from the feature-dependent functions that we could not mitigate for the tested real-world programs as they were hard to identify. However, there are also false negatives and false positives that are specifically caused through our implementation of the coverage-based baseline.

Our implementation can generate different presence conditions for features in alternative groups. For instance, in XZ, a code region executed only when the compression feature is active can have two corresponding presence conditions: *compress* and $\neg$*decompress* $\wedge$ $\neg$*list* $\wedge$ $\neg$*test*. Both conditions are equivalent because *compress* cannot coexist with the other features in the alternative group. Since we are using a BDD that automatically minimizes our presence conditions, we have no direct control over whether the former or the latter presence condition results. If the latter occurs, false negatives and false positives will be the consequence. We could mitigate the problem if we had information from the feature model about alternative groups for our presence-condition simplification, but this is not currently implemented. As a workaround, in step 1 (Section 3.1.2.1) of our approach where we construct the initial presence condition from the configuration, we omit appropriate negated literals if a positive literal from the same alternative group occurs. That biases the BDD in the right direction. This workaround reduces the amount of false negatives and false positives and improves precision and recall by low single-digit percentages. However, it does not solve the problem completely.

5.1.2.5 *Features managing verbosity*

As a fifth group, we discuss features that affect the verbosity of a program. In other words, features such as *quiet*, *verbose*, or *v* in Table 5.11, 5.12, and 5.13. For these, precision and recall depend heavily on the extent to which the features are used in the error handling code. In our real-world programs, these features are often used in the error-handling code to decide whether or how detailed an error should be output. However, the coverage-based baseline is generated by executing only valid configurations with valid workloads, so error handling code is usually not covered. Therefore, our coverage-based baseline can only detect feature regions that are not in error-handling code. As a result, verbosity features tend to have lower precision and higher recall. We examined the false positives again to verify the feature regions not covered by the coverage-based baseline, without detecting anomalies.

5.1.2.6 *Remaining Features*

The remaining features in group six are rather hit-or-miss regarding their precision and recall values. Some like *keep* in Table 5.11 and 5.12 have particularly high precision and recall values, others like *small* in Table 5.11 have particularly low values. The remaining features, such as *stdout* (Table 5.12 and 5.13) or *suffix* (Table 5.13) have mediocre precision and recall values.

Despite the variability in their values, these features share the common trait of having higher precision than recall. This means that the coverage-based baseline identifies a greater number of feature regions compared to VaRA for those features.

However, with *force* and *no-name* in Table 5.12 there are two outliers in this group with low precision, high recall and a large number of false positives. Due to the sheer quantity, we are unable to individually validate all of the false positives. However, the majority seems to be indeed incorrectly detected feature regions by VaRA. Since numerous sections of GZIP have *force* and *no-name* annotated without a clear rationale, it appears that at some point VaRA's taint analysis marks a wrong instruction or variable.

5.1.2.7 *Summary*

As shown, features from different categories have different performance characteristics. The reasons for this are diverse. In some cases, it is due to detection limitations of VaRA (macros) or the coverage-based baseline (numeric and verbosity features). Sometimes, it is due to detection anomalies in one of the approaches, like for enums or features in alternative groups. However, in other cases it works as expected, like for most features in the remaining feature group.

5.2 DISCUSSION

In this section, we discuss the results of our quantitative and qualitative analysis in order to answer RQ1 and RQ2. Furthermore, we discuss the obtained insights in relation to our thesis goal.

5.2.1 *RQ1: What conceptual differences in detected feature regions exist?*

As seen in Section 5.1.1, there are two types of conceptual differences that cause the coverage-based baseline and VaRA to identify different feature regions.

SOURCE-CODE-LEVEL DIFFERENCES The first category are source-level differences, that is, different parts of the source code that are considered feature-dependent. These are conditions, command-line option parsing related code, and feature-dependent functions. They cause the largest deviations between the coverage-based baseline and VaRA's feature-region detection, but can be mitigated well. Our mitigations are not intended to improve the performance metrics, that is only a side effect. Rather, we implement these measures in order to reduce the over-approximation of the coverage-based baseline to avoid already known false negatives and false positives in the results. This reduces our manual analysis effort. The significance of the results is not affected by our mitigations, as we only mitigate the conceptual differences that are always present. This does not change the big picture.

INSTRUCTION-LEVEL DIFFERENCES The second category are instruction-level differences where parts of a basic block belong to different code regions, causing flaky feature regions. We observe these differences in classes, short-circuiting conditions, ternary operators, and early returns. However, these kinds of differences cause only a small deviation between the

coverage-based baseline and VaRA's feature-region detection, as evident by the small number of flaky feature regions. Mitigating them does not make a big difference.

Therefore, we answer our first research question as follows.

> **Answer RQ1:** There are conceptual differences in the identified feature regions between the coverage-based baseline and VaRA's feature-region detection, both at the source code level and at the instruction level. The former are differences in the detection of conditions, command-line-option-related code, and feature-dependent functions. The latter concern classes, short-circuit conditions, ternary operators, and early returns.

5.2.2 *RQ2: How well does VaRA's feature-region detection perform on real-world programs?*

As we have seen in Section 5.1.2, the performance of VaRA's feature-region detection is very different for certain kinds of features. For a better overview, we present the averaged metrics of the features in a group in Table 5.14. Since these feature groups are not uncommon and are found in other real-world programs, we assume that the performance results for other real-world programs are similar. In the following, we discuss the reasons behind the varying performance results of each group.

Table 5.14: Arithmetic mean of performance metrics per feature group (for threshold 0 %).

ID	Group	PPV (%)	TPR (%)	TNR (%)	BA (%)
1	Macro	-	0.00	100.00	50.00
2	Enum	20.71	24.41	93.30	58.86
3	Numeric	0.27	6.25	98.90	52.44
4	Alternative	16.61	18.12	94.15	56.13
5	Verbosity	21.33	46.58	99.19	72.80
6	Remaining	31.08	33.68	96.15	64.91

Legend: PPV = Precision, TPR = Recall, TNR = Specificity, BA = Balanced Accuracy

MACROS Since macros are preprocessor directives that get expanded before generating the LLVM IR, it is currently impossible for VaRA to track feature variables defined as macros. Therefore, VaRA is unable to detect any associated feature regions. However, the coverage-based baseline can identify these feature regions because it does not depend on the concrete implementation of the feature variables. This is why there are only false negatives and true negatives in the macro feature group. Given the absence of true positives and false positives, calculating the precision is not possible. However, the recall is 0 %.

ENUMS VaRA's feature-region detection can assign specific enum values to feature regions. The results show that this is not always working. There are some feature regions that get

assigned all enum values for no apparent reason. This leads to a high number of false positives, which reduces the precision. We suspect that this is a bug in VaRA's feature-taint analysis, which underlies the feature-region detection. For the *Enum* feature group, we determine a precision of 20.71 % and a recall of 24.41 %. However, these values are not exclusively due to the bug, because the features implemented as enum value in XZ are also part of an alternative group. This causes additional false positives and false negatives that negatively impact precision and recall average of the *Enum* feature group.

NUMERIC FEATURES Our coverage-based baseline has a detection problem with numeric features. This is due to the fact that numerical features often do not change the regions of code that are executed. Numerical features that are only used in computations or passed to the operating system (e. g., the size of data to read into a block) have no effect on coverage and therefore cannot be detected. However, even if numeric features occur in conditions, for example, to determine how often a loop is executed, they are very difficult to detect. This is because we generally have to test all possible values to find those that satisfy the condition, or no longer satisfy the condition, in order to achieve a change in executed code regions. This significantly increases the effort, which is why we avoid it. For static analyses, such as VaRA's feature-region detection, numeric features do not pose a challenge, as we can see from the false positives. However, due to the scarcity of feature regions detected by the coverage-based baseline, we barely have any true positives. This absence explains the extremely low precision and recall values of 0.27 % and 6.25 %. Both of these values are caused by the single true positive for the feature *9* in GZIP (Table 5.12). Without it, precision and recall would be 0 %.

ALTERNATIVE GROUPS For the features that are part of an alternative group in the feature model, our implementation of the presence-condition simplification causes problems. The presence condition of a feature in an alternative group can be represented either as a positive literal of the feature itself or as a conjunction of the negated literals of the other features in the alternative group. Our implementation lacks information about alternative groups in the feature model, which means we cannot influence the specific case to which our BDD minimizes the presence condition. This leads to a large number of false positives and false negatives for alternative groups when the wrong case occurs. In Section 5.1.2.4, we previously explained our workaround that partially mitigates this issue but fails to resolve it entirely. Despite the workaround, due to the many false negatives and false positives, we obtain an average precision of 16.61 % and a recall of 18.12 % for features in an alternative group. We anticipate that resolving this issue will lead to an increase of another 5–15 % in precision and recall for these features. This is because feature regions of this kind are actually correctly identified by the coverage-based baseline, but poorly interpreted by our implementation.

VERBOSITY FEATURES Features in the verbosity group are frequently used in error-handling code in the real-world programs we tested. As we do not execute invalid configurations or invalid workloads to create our coverage-based baseline, this error-handling code is not executed. As a result, our coverage-based baseline does not find any feature regions there, but VaRA does. This imbalance results in an average precision of 21.33 % and a more than twice as high recall of 46.58 %. Assuming we would also execute the error-handling code, the precision and recall would probably be about the same as for the other feature groups. However, the

effort to execute all the error handling code is enormous. This is because we would have to run not only all invalid configurations of a program, but also every configuration with not only valid but also many invalid workloads, since an invalid workload is likely to cause exactly one error. In addition, we would have to run the program with different resources, for example, to trigger errors if there is not enough memory available. Thus, if we try to run error-handling code, we do not scale. For this reason, we have decided not to pursue this approach.

REMAINING FEATURES As noted in Section 5.1.2.6, the remaining group of features contains two outliers with very low precision and high recall (*force* and *no-name* in Table 5.12). These influence our average precision and recall of 31.08 % and 33.68 % strongly. Because if we exclude them from the average, we get a different picture with 37.02 % precision and 24.59 % recall. Higher precision than recall means that the coverage-based baseline is more likely to over-approximate the feature regions found by VaRA. Empirical data from our example programs, MSMR (Table 5.1) and SFI (Table 5.2), generally support this assumption. This property is welcome because it means that the coverage-based baseline is more likely to find too many of VaRA's feature regions than too few. That this is still the case despite our condition handling and option parsing code mitigations confirms that they are not too restrictive.

As we have seen significant differences between the individual feature groups, we answer our two sub-questions per feature group.

> **Answer RQ2.1:** The fraction of detected feature regions that are genuine is: Not measurable for macros, 20.71 % for enums, 0.27 % for numeric features, 16.61 % for features in an alternative group, 21.33 % for features managing verbosity, and 31.08 % (37.02 % without outliers) for remaining features.
> **Answer RQ2.2:** The fraction of genuine feature regions that are detected as such is: 0 % for macros, 24.41 % for enums, 6.25 % for numeric features, 18.12 % for features in an alternative group, 46.58 % for features managing verbosity, and 33.68 % (24.59 % without outliers) for remaining features.

In absolute values, the performance of VaRA's feature region detection on real-world programs is not great. The average metrics of the individual feature groups, as well as the totals of the real-world programs, reveal this. Nonetheless, we cannot conclude that VaRA itself performs poorly, only that we could not measure better performance stats with our coverage-based baseline. The concepts of the coverage-based baseline and VaRA are too different for that. On a small scale, we were able to ignore most differences in our example programs and synthetic benchmarks with our mitigations. However, that is not the case anymore for real-world programs where we do not mitigate feature-dependent functions anymore. Additionally, our coverage-based baseline has its shortcomings, particularly with numeric features, alternative groups, and error-handling code. As a result, we experience a high number of false positives and false negatives, that negatively affect our performance metrics. Therefore, we answer our second research question as follows.

Answer RQ2: The precision and recall of VaRA's feature-region detection on real-world programs is not very high, but this does not mean that VaRA is bad at detecting feature regions in real-world programs. There are areas where VaRA's feature-region detection could be improved, such as the lack of macro support or the feature-taint analysis, which seems to be buggy in some places. However, a large part of the low performance metrics is due to differences between the coverage-based baseline and VaRA, which add up and become more significant the larger the code base.

5.2.3 *Thesis Goal: Does VaRA's feature-region detection yield valid results?*

We tested VaRA's feature-region detection on small (example and synthetic programs) and large (real-world programs) scale. By classifying the results using the coverage-based baseline, we validate most of the results automatically. Unfortunately, our coverage-based baseline also has its shortcomings regarding numeric features, alternative groups, and error handling code, so we still need to validate these areas manually. However, we found no new anomalies during our validation of the corresponding source code in the real-world programs. VaRA's feature-region detection performed as expected. Therefore, we stick to our findings from the previous two research questions regarding the validity of VaRA's feature-region detection results.

Answer thesis goal: The feature-region detection results from VaRA are valid on the whole. However, there seems to be a bug in the underlying feature-taint analysis, which manifests itself in enum comparisons, for example, and causes too large parts of a program to be considered feature-dependent.

5.3 THREATS TO VALIDITY

There are several threats to the validity of our results, both internal and external. Internal threats to validity can arise from faulty measurements or improper processing of the measured data during analysis. External validity threats impact the generalizability of our results. In this section, we discuss what these threats are and how we mitigate them when possible.

5.3.1 *Internal validity*

Mühlbauer et al. [22] demonstrated, among others, that they could identify feature regions with coverage data. Therefore, we do not have to prove anymore that this is possible. Hence, the only question remaining is whether our methodology is valid. Our results are only reliable if the classification of VaRA's feature-region detection results with our coverage-based baseline works as intended. To obtain a good coverage-based baseline, we first need valid coverage data that covers a large portion of the program under investigation. When the coverage data is not valid or of poor quality, we cannot find all the coverage-based feature regions.

For valid and precise coverage data, we use LLVM's source-based coverage measurements, a mature functionality that has proven reliable in practice for many years. These measurements involve profiling instructions added at the beginning of every code region, which track the number of times that code region is executed. To prevent the profiling instructions from interfering with VaRA's feature-region detection, we compile our programs twice. Once with profiling instructions and once without. Apart from the profiling instructions, the resulting LLVM IR is identical, as the other compiler options remain the same.

In order to obtain high-quality coverage data, we execute all valid configurations with workloads that match their features. In other words, we make sure that the workload is capable of executing the feature-relevant code. This is especially crucial for our real-world programs, which have multiple operation modes with different input requirements. For instance, if we want to decompress data, we require a compressed file as the workload. Using an uncompressed file as input would only result in an error. Furthermore, we ensure that our coverage data is deterministic, meaning that two executions of a program with the same inputs result in the same coverage data. This is important because code regions in non-deterministic programs that are sometimes executed and sometimes not would be falsely identified as feature regions by our coverage-based baseline. To avoid this, we only test deterministic configurations of programs. For instance, it is feasible to execute XZ with several threads for faster compression. However, that causes non-deterministic executions, which will likely result in non-deterministic coverage data. Hence, we always run XZ in single-threaded mode.

In addition to the coverage data, the generated LLVM IR may also cause problems. To be able to uniquely assign instructions to code regions, each instruction must have exactly one location in the source code, effectively a one-to-many mapping. However, compiler optimizations or language features like preprocessor macros may lead to a many-to-many mapping, where multiple source code locations are represented by a single instruction. In this situation, it is unclear to which section of the source code the instruction belongs, since instructions in LLVM can only have one debug location. We would overlook that instruction for the other relevant code regions. While this possibility cannot be completely eliminated, we greatly reduce its likelihood by disabling compiler optimizations when compiling our programs. As a result, many-to-many mappings in the LLVM IR should occur rarely, if at all, which in the worst case could slightly affect our results. However, this does not render our results invalid per se.

Despite the correctness of the coverage data and LLVM IR, our implementation could still introduce errors when processing them. We have identified potential risks in the parsing of LLVM's coverage data JSON format to code region trees, the simplification of presence conditions, and the classification of feature regions. These areas are relatively complex to implement. However, we anticipate a low probability of bugs in our implementation, as we thoroughly test these areas with our test suite and continuously check many of our assumptions with assertions. Through the latter, we discovered, for example, that the coverage data also contains information about compile-time generated source files (e. g., `config.h` generated by GNU autotools), which we have not included in our code region trees because they are not part of the actual source code of the program. Furthermore, we found a bug in the LLVM version we use. LLVM 14 erroneously reports the end column of a single code region in ECT to be higher than the length of the corresponding line. However, this issue

occurs only in this place. All other code regions are correctly represented in the coverage data.

Even with correct implementation, we might make mistakes in our evaluation. For example, we may miss feature regions in our quantitative analysis if they are not detected by both the coverage-based baseline and VaRA. Such instances are categorized as true negatives. The only way to confirm these true negatives is through manual verification, which can be a significant undertaking, especially for the real-world programs, due to their large number. However, we do not assume that there are any missed feature regions among the true negatives. First, the coverage-based baseline and VaRA's feature region detection differ conceptually, making it likely that at least one approach will identify the feature region. Second, in our qualitative analysis, we explicitly validate the true negatives and found them to be correct, so it is unlikely to be a general problem. Also, when we examined the false positives and false negatives in both the synthetic and real-world programs, we did not notice anything out of the ordinary. Therefore, even if we missed feature regions in the true negatives, their occurrence would be rare and have low impact on the validity of our findings.

Another potential factor affecting internal validity is our choice of features to test. Due to the need to execute all valid configurations to establish a coverage-based baseline – and the exponential growth in the number of configurations based on the number of tested features – it is not feasible to test all available features. Hence, we must strategically select the features we wish to evaluate in real-world programs. However, there is a risk that the selection of features affects our results because, as we saw in Section 5.1.2, features part of distinct feature groups have different performance characteristics. The overall performance metrics could vary if different features are chosen, which may no longer accurately represent VaRA's detection of feature regions across the entire program. To minimize the number of configurations required for testing, we limit the values of numerical features in the real-world programs to two: the default value, if available, and one other. We also refrain from testing rather insignificant features such as *help* or *version*, which only print some text but do not execute the primary logic of the program. These measures enable us to test nearly all features of Bzip2 and gzip, thus avoiding the problem. However, with XZ, we cannot test all features because XZ has more than twice as many features as Bzip2 or gzip. Therefore, we restrict testing to the most common features that have the potential to interact with each other. Since the chosen features incorporate various functionalities, many of which overlap with those present in Bzip2 and gzip, we still get a broad view of VaRA's feature-region detection performance for XZ.

Finally, it is worth noting that our chosen methodology affects our results. Our approach to detect feature regions via coverage data is, of course, conceptually different from the static analysis of VaRA's feature-region detection. How the two approaches differ, how we mitigate these differences, and how this affects the results has already been described in detail in Section 5.1.1 and 5.2.1.

5.3.2 *External validity*

Although we examine several real-world programs, the generalizability of our results is limited. This is because all of the real-world programs we test have three things in common: they are from the compression domain, are written in C/C++, and do not implement their feature variables in config structs. As a result, our performance metrics may look quite

different for programs in other domains, other compiled programming languages like Rust, or when config structs are used. However, we do not consider this an issue for this work as our intention is to validate the current state of VaRA's feature-region detection. Currently, VaRA only supports C/C++, and the necessary field sensitivity support to correctly handle config structs is not yet fully developed. However, some of the reasons behind the different feature groups (lack of macro support, numeric features, error handling code) are generalizable, as they are more fundamental issues that occur regardless of the used programming language and implementation specifics.

6 RELATED WORK

Feature detection tools map features to source code, which helps developers or maintainers in locating the implementation of a feature in a program. An overview of the subject and different approaches is provided by Dit et al. in their paper "Feature location in source code: a taxonomy and survey" [10]. Essentially, there are two methods for locating features in source code. One can collect information during program execution (dynamic analysis) or analyze a program statically, that is, without executing it.

Wilde et al. [26] are the first to solve the problem of feature detection with dynamic analysis in 1992. They gather coverage data from executions with and without activated features and compare the two sets to identify feature-related code. This concept of using coverage data to locate features still exists today and is used in modified forms [7, 12, 22].

In 1994, Biggerstaff, Mitbander, and Webster [3] introduce the first static approach to feature detection. Their static analysis is text-based, that is, the source code itself is analyzed. For example, among other things, they search for the occurrences of identifier names that can be related to a feature. Over time, various static concepts were developed to enhance detection accuracy. A noteworthy concept is the analysis of preprocessor directives for locating compile-time variable features in programs [15, 21]. However, we will not delve into the specifics as this thesis focuses on load-time variability only. The static analysis tool most akin to VaRA is Lotrack [19]. Just like VaRA's feature-region detection, Lotrack uses a taint analysis to track the flow of data from configuration options or feature variables through the program. It also considers instructions whose control flow depends on a tainted value to be feature-dependent. However, Lotrack is designed specifically for detecting features in Android applications, and thus only supports the Java ecosystem. VaRA, on the other hand, supports feature detection in C/C++ programs.

During the development of VaRA's feature-region detection, it became evident that an automatically generated ground truth is needed, against which the results of the feature region detection can be compared, for example, for evaluation or validation purposes. Until now, the ground truth for a program had to be laboriously created manually, which limited the evaluation possibilities in scale and scope. Two evaluations related to VaRA's feature-region detection have been performed so far. One was conducted by Keller [14]. In his bachelor thesis, he evaluates the underlying feature-taint analysis of VaRA's feature-region detection on four real-world programs. To accomplish this, he had to create the necessary ground truth data by hand. The other was undertaken by Zahlbach [27] also in his bachelor thesis. He compares different feature-region detection approaches with each other on real-world programs. However, he could only validate the results of the detection approaches on small code examples because it was not feasible to generate the necessary ground truth data on such large scale. Our work builds on his proposal to automatically generate ground truth from coverage data in order to be able to validate VaRA's feature-region detection approaches on a large scale on real-world programs.

Although there have been hybrid approaches to feature detection that attempt to combine the strengths of dynamic and static analysis [11, 13, 23], to the best of our knowledge, we are the first to attempt to evaluate and validate a static feature-region detection analysis with a dynamic analysis. Our initial approach for creating the coverage-based baseline is very similar to the one used by Mühlbauer et al. [22] in their work. As explained in Section 3.1.1, it is possible to detect feature regions by computing the difference of coverage data. The sole difference is that their implementation uses lines of code as the granularity, while we use code regions. Using code regions is more precise, because it is possible that a line of code consists of multiple code regions, some of which are covered and some of which are not. In such cases, the coverage tools usually display the maximum count of all code regions in the line, which can result in code regions being marked as covered when they are not. Thereby, the accuracy of the generated baseline decreases.

However, as also described in Section 3.1.1, the coverage diffing approach only gives us the features that affect a code region, but not the presence condition of the code region. The key idea for implementing presence conditions for our coverage-based baseline came from Rhein et al. In their paper "Presence-Condition Simplification in Highly Configurable Systems" [24], they describe how to minimize presence conditions in the context of a feature model and test the efficiency of different algorithms for doing so. Our implementation follows their recommendation to use a BDD for presence-condition simplification, as this is currently the best scaling technique.

7 CONCLUDING REMARKS

To conclude this thesis, we summarize our findings, make recommendations for immediate improvements, and provide an outlook for future research.

7.1 CONCLUSION

The goal of this thesis is to validate VaRA's feature-region detection on real-world programs. This has been impractical so far, since the baseline required for validation could only be created by hand on a small scale. We solve this problem by implementing a method that extracts feature regions from coverage data and use it to automatically generate baselines for real-world programs.

We evaluate VaRA's feature-region detection on programs by comparing its results with the coverage-based baseline. Since both approaches are based on different concepts, we first work out their conceptual differences so that we are able to correctly assess the comparison results. We found that VaRA's feature-region detection and our coverage-based baseline differ on the source code level in terms of condition handling, option parsing code, and feature-dependent functions. However, these differences can be mitigated without compromising the meaningfulness of the results. Some minor discrepancies exist at the instruction level, but they are not significant.

Based on our findings, we assessed the performance of VaRA's feature-region detection on three real-world programs: Bzip2, gzip, and XZ. We discovered that various feature groups exhibit different performance characteristics. This is due to shortcomings of both the coverage-based baseline and VaRA. The coverage-based baseline has problems finding numeric features, features in error handling code, and, due to our implementation, features that are part of an alternative group. VaRA cannot track feature variables defined as macros due to the absence of macro support and might have a bug in the feature-taint analysis, which manifests when enum values are involved in conditions, among others.

Apart from the aforementioned issues, we did not find any systematic errors during our validation of the results, so we conclude that VaRA's feature region-detection provides valid results on the whole.

That is, we are confident that our coverage-based baseline will substantially decrease the manual labor needed to evaluate VaRA's feature-region detection or comparable tools in the future. This will ultimately expedite further development efforts.

7.2 POSSIBLE TOOLING IMPROVEMENTS

Albeit not being the main goal of this thesis, while working towards the evaluation, we have developed tools around the coverage-based baseline that could be of immediate use in the further development of VaRA. One of them is a side-by-side view of the source code and the features associated with each line. This visualization provides an initial overview of whether

corresponding feature-region detection approaches are finding the correct feature regions. Such an visualization has been missing from VaRA until now. With little effort, we could also visualize the feature taints in this view, for example, by color coding the tainted variables in the source code according to the corresponding feature. This could facilitate bug finding in the feature-taint analysis.

Furthermore, our tools could help to detect and correct deviations of our feature models from reality. During our experiments, we computed and executed all valid configurations of a program based on the feature model. However, some of these configurations were in reality invalid and the program returned an error. We corrected the feature models for the case studies investigated in our evaluation, but it is possible that other feature models are also incorrect. Therefore, we recommend to check whether other feature models correctly model the reality. Note that validating feature models is a separate research topic with its own challenges [18]. However, for smaller systems, the simple approach of running all valid configurations and checking the exit code should suffice.

Finally, as we have seen in the case of XZ, dynamically linked libraries cause problems for VaRA's feature-taint analysis. Their instructions are not present in the analyzed LLVM IR, which hinders the feature-taint analysis from continuously analyzing the LLVM IR. Therefore, to prevent unintentional omitted instructions, VaRA's tooling should display a warning for dynamically linked libraries during the compile process.

7.3 FUTURE WORK

As we have seen through our experiments, there is potential for improvement in our coverage-based baseline, VaRA's feature-region detection, as well as in our mitigations to reduce the conceptual differences between the two.

For the coverage-based baseline, it is possible to evaluate the coverage data at a more fine-grained level. Currently, we only distinguish between executed and non-executed code regions to assign initial presence conditions. As a result, our coverage-based baseline rarely detects any numeric features. If we could determine for how many executions of a feature region a feature is responsible, we could improve numeric feature detection in loops, since we would know how many loop executions a feature caused. However, there is a risk that the coverage-based baseline may become more susceptible to noise, such as when loops wait for an event to happen, which could degrade the results. Furthermore, the coverage-based baseline could be extended to support compile-time variability. We assume that the locations of code regions in the coverage data remains the same across compile-time variants, so our code region trees should still have the same structure. Thus, to support compile-time variability, it should suffice to combine the corresponding compile-time configuration with the run-time configuration and set the outcome as the initial presence condition for all executed code regions.

VaRA's feature-region detection could implement macro and field-sensitivity support to be able to handle features implemented as macros and enable feature detection in config structs. Furthermore, VaRA could also implement presence conditions for feature regions. To accomplish this, symbolic execution of control-flow paths that lead to feature regions could be performed. The validation of this presence condition implementation could be easily achieved by comparing to our coverage-based baseline.

Regarding our mitigations for the conceptual differences between the coverage-based baseline and VaRA's feature-region detection, we had to manually find and ignore feature-dependent functions. However, we were unable to clearly identify feature-dependent functions in the real-world programs, so we decided to not mitigate them. As explained in Paragraph 5.1.1.1, we could automatically ignore these functions by statically detecting them with a call graph. However, this approach carries the risk of overlooking dynamic calls, which would falsely declare functions as feature-dependent. Therefore, it would be interesting to find out how prevalent this problem is in practice and whether the call graph approach is useful.

Finally, it could be beneficial to combine our coverage-based baseline and VaRA's feature region detection to overcome each other's shortcomings and increase feature detection accuracy. Similar approaches have been used successfully in research [23]. However, the specifics of how this could work in our case are still unclear.

A

APPENDIX

A.1 FEATURE INTERACTION IN ECT

During our preparations to evaluate ECT, we encountered a feature interaction between the `-gzip` command-line option and `--disable-png` or `--disable-jpg`. As demonstrated in Listing A.1, enabling the *gzip* feature reactivated the previously disabled features. The problem has been fixed by the developer[1].

```
1 ./ect --disable-png --disable-jpg image.png image.jpg
2 No compatible files found
3
4 ./ect -gzip --disable-png --disable-jpg image.png image.jpg
5 Processed 2 files
6 Saved 40.77KB out of 276.14KB (14.7637%)
```

Listing A.1: Feature interaction in ECT.

A.2 DISABLED EXCEPTION HANDLING

In C++ programs, exceptions that are not handled usually trigger an abort of the program. Although these exceptions are not handled explicitly in source code, the compiler implicitly handles them in LLVM IR. For example, in Listing A.2 the operations `compress`, `encrypt`, and `addPadding` try to allocate memory, which can fail. Since this exception is not explicitly handled, the program can abort at these locations early. However, this changes the control flow of the program because it causes the execution of subsequent instructions to depend on the success of previous operations. Consequently, the features identified by VaRA's feature-region detection begin to accumulate, as can be seen in the right column of Listing A.2. Unfortunately, this causes VaRA's detected feature regions to deviate significantly from our coverage-based baseline. To avoid this phenomenon, we compile all programs with disabled exception handling.

1 https://github.com/fhanau/Efficient-Compression-Tool/issues/133 (visited on January 16, 2024)

		Coverage	VaRA
51	⋯|		
52	`if (UseCompression){`	*True*	
53	`  Data = compress(Data);`	*compress*	*Compression*
54	`}`	*True*	*Compression*
55	`if (UseEncryption){`	*enc*	*Encryption*, *Compression*
56	`  if (not UseCompression){`		
57	`    Data = addPadding(Data);`	$\neg compress \wedge enc$	*Encryption*, *Compression*
58	`  }`		
59		*enc*	*Encryption*, *Compression*
60	`  Data = encrypt(Data);`		
61	`}`		
62			
63	`//Sending`		
64	`fpsc::sleep_for_secs(2);`	*True*	*Encryption*, *Compression*
65	`send(&Data);`		
66	⋯		

Listing A.2: VaRA's detected feature regions in SFI compiled without `-fno-exceptions`.

BIBLIOGRAPHY

[1] Frances E. Allen. "Control Flow Analysis." In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois: Association for Computing Machinery, 1970, pp. 1–19. ISBN: 9781450373869. DOI: 10.1145/800028.808479. URL: https://doi.org/10.1145/800028.808479.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. "Software Product Lines." In: *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 3–15. ISBN: 9783642375217. DOI: 10.1007/978-3-642-37521-7_1. URL: https://doi.org/10.1007/978-3-642-37521-7_1.

[3] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. "Program Understanding and the Concept Assignment Problem." In: *Commun. ACM* 37.5 (1994), pp. 72–82. ISSN: 0001-0782. DOI: 10.1145/175290.175300. URL: https://doi.org/10.1145/175290.175300.

[4] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. "Efficient Implementation of a BDD Package." In: *Proceedings of the 27th ACM/IEEE Design Automation Conference*. DAC '90. Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 40–45. ISBN: 0897913639. DOI: 10.1145/123186.123222. URL: https://doi.org/10.1145/123186.123222.

[5] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation." In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[6] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. "Feature interaction: a critical review and considered forecast." In: *Computer Networks* 41.1 (2003), pp. 115–141. ISSN: 1389-1286. DOI: https://doi.org/10.1016/S1389-1286(02)00352-3. URL: https://www.sciencedirect.com/science/article/pii/S1389128602003523.

[7] Bruno Castro, Alexandre Perez, and Rui Abreu. "Pangolin: An SFL-Based Toolset for Feature Localization." In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 1130–1133. DOI: 10.1109/ASE.2019.00119.

[8] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. "Verification of synchronous sequential machines based on symbolic execution." In: *Automatic Verification Methods for Finite State Systems*. Ed. by Joseph Sifakis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 365–373. ISBN: 978-3-540-46905-6.

[9] Olivier Coudert and Tsutomu Sasao. "Two-Level Logic Minimization." In: *Logic Synthesis and Verification*. Ed. by Soha Hassoun and Tsutomu Sasao. Boston, MA: Springer US, 2002, pp. 1–27. ISBN: 978-1-4615-0817-5. DOI: 10.1007/978-1-4615-0817-5_1. URL: https://doi.org/10.1007/978-1-4615-0817-5_1.

[10] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. "Feature location in source code: a taxonomy and survey." In: *Journal of Software: Evolution and Process* 25.1 (2013), pp. 53–95. DOI: https://doi.org/10.1002/smr.567. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.567.

[11] Marc Eaddy, Alfred V. Aho, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. "CERBERUS: Tracing Requirements to Source Code Using Information Retrieval, Dynamic Analysis, and Program Analysis." In: *2008 16th IEEE International Conference on Program Comprehension*. 2008, pp. 53–62. DOI: 10.1109/ICPC.2008.39.

[12] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. "Feature-driven program understanding using concept analysis of execution traces." In: *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 2001, pp. 300–309. DOI: 10.1109/WPC.2001.921740.

[13] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. "Locating features in source code." In: *IEEE Transactions on Software Engineering* 29.3 (2003), pp. 210–224. DOI: 10.1109/TSE.2003.1183929.

[14] Janik Keller. "Feature Taint Analysis: How Precise can VARA Track the Influence of Feature Variables in Real-World Programs?" Bachelor's thesis. Saarland Informatics Campus, Saarland University, 2023.

[15] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. "Where is my feature and what is it about? A case study on recovering feature facets." In: *Journal of Systems and Software* 152 (2019), pp. 239–253. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2019.01.057. URL: https://www.sciencedirect.com/science/article/pii/S0164121219300184.

[16] Chris Lattner. *The Architecture of Open Source Applications (Volume 1) LLVM*. 2011. URL: https://aosabook.org/en/v1/llvm.html (visited on 05/23/2023).

[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.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[18] Duc Minh Le, Hyesun Lee, Kyo Chul Kang, and Lee Keun. "Validating consistency between a feature model and its implementation." In: *Safe and Secure Software Reuse: 13th International Conference on Software Reuse, ICSR 2013, Pisa, June 18-20. Proceedings 13*. Springer. 2013, pp. 1–16.

[19] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking Load-Time Configuration Options." In: *IEEE Transactions on Software Engineering* 44.12 (2018), pp. 1269–1291. DOI: 10.1109/TSE.2017.2756048.

[20] Yashwant K. Malaiya, Michael Naixin Li, James M. Bieman, and Rick Karcich. "Software reliability growth with test coverage." In: *IEEE Transactions on Reliability* 51.4 (2002), pp. 420–426. DOI: 10.1109/TR.2002.804489.

[21] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton G. Assunção, Paul Grünbacher, and Alexander Egyed. "Locating Feature Revisions in Software Systems Evolving in Space and Time." In: *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*. SPLC '20. Montreal, Quebec, Canada: Association for Computing Machinery, 2020. ISBN: 9781450375696. DOI: 10.1145/3382025.3414954. URL: https://doi.org/10.1145/3382025.3414954.

[22] Stefan Mühlbauer, Florian Sattler, Christian Kaltenecker, Johannes Dorn, Seven Apel, and Norbert Siegmund. "Analyzing the impact of workloads on modeling the performance of configurable software systems." In: *Proceedings of the International Conference on Software Engineering (ICSE), IEEE*. 2023.

[23] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. "Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval." In: *IEEE Transactions on Software Engineering* 33.6 (2007), pp. 420–432. DOI: 10.1109/TSE.2007.1016.

[24] Alexander von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. "Presence-Condition Simplification in Highly Configurable Systems." In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. 2015, pp. 178–188. DOI: 10.1109/ICSE.2015.39.

[25] Florian Sattler. "A Variability-Aware Feature-Region Analyzer in LLVM." Master's thesis. Department of Informatics and Mathematics, University of Passau, 2017.

[26] Norman Wilde, Juan A. Gomez, Thomas Gust, and Douglas Strasburg. "Locating user functionality in old code." In: *Proceedings Conference on Software Maintenance 1992*. 1992, pp. 200–205. DOI: 10.1109/ICSM.1992.242542.

[27] Tom Zahlbach. "Finding Feature-Dependent Code: A Study on Different Feature-Region Detection Approaches." Bachelor's thesis. Saarland Informatics Campus, Saarland University, 2023.