Bachelor's Thesis

FINDING FEATURE-DEPENDENT CODE: A STUDY ON DIFFERENT FEATURE-REGION DETECTION APPROACHES

TOM ZAHLBACH

February 2, 2023

Advisor:
Florian Sattler Chair of Software Engineering

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

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Erklärung

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

Statement

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

Einverständniserklärung

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

Declaration of Consent

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

Saarbrücken, 02.02.2023 Tom Zahlbach

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Today's complex software projects are almost always configurable and allow users to customize their configuration to their liking. However, these highly-configurable software projects put a burden on developers, as every feature contributes exponentially to the complexity of configuration options, and interactions between features can lead to unexpected behavior and bugs.

Hence, developers require specialized analysis tools with configuration-aware methods that analyze these software projects and help developers reason about a project. Many of these methods require a mapping between a configuration option and the code sections it influences to reason about the software project. These code sections, called feature regions, are usually detected with algorithms designed to isolate them.

Considerable time and effort go into evaluating a feature-region detection algorithm's performance, strengths, and weaknesses. Most of which is spent on evaluating whether results generated on test code or projects match desired results. Automatically discovering disagreements between detection results and a fixed ground truth would alleviate a big part of the manual labor involved in evaluating a detection strategy.

We present a straightforward framework to compare feature-region detection algorithms with ground-truth data or alternative detection strategies and evaluate the results. Our framework highlights code locations where the approaches disagree. Highlighting these code patterns in real-world software projects then enables a detailed comparison of each strategy's strengths and weaknesses and helps developers to tune and improve these detection strategies. Additionally, we showcase a comparison between two different detection strategies and present the strengths and weaknesses we uncovered using our framework. Our evaluation is split into two parts. Firstly, we compared the two strategies on small code samples to test for specific edge cases and feature encodings. Secondly, we compared the two strategies on real-world software projects to highlight previously undiscovered disagreements between the strategies.

Our work demonstrates that analysis developers can gain valuable information about their detection strategy by comparing them with alternative strategies on real-world software projects or against a handcrafted ground truth.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

ICFG	interprocedural control-flow graph
DT	dominator tree
LLVM IR	LLVM intermediate representation

1 INTRODUCTION

When working with modern software, it is almost inevitable that one comes across configuration options in the form of command line parameters, environment variables, or configuration files that control sections of the code and enable them only in specific cases. Configuration options can add functionality, leave out optional parts, alter the program's behavior, or influence the program's execution in another way. They generally control and affect the features or functionality built into a program. On Linux systems, for example, the `find` command has the command line parameter `-exec`, which prompts the find program to run a command on all files found. Every configuration option contributes to the complexity of the software, which makes it harder for developers to test their software for bugs or unexpected behavior [13]. So to help out developers, configuration-aware analysis tools have been developed which analyze or reason about a program while being aware of its configuration options to help developers maintain highly configurable software by making them aware of unexpected interactions between configuration options [16, 18, 20]. We can, for example, isolate what code is executed when the `find` command is used with the `-exec` parameter, watch the performance impact it has [18], or monitor how `-exec` interacts with other parameters, such as `-d`, which deletes all files the command finds [7, 20]. These analysis tools require a precise mapping from features, or the functionality of a program, onto the sections of code that perform the functionality. These code regions are called feature regions and are detected by specially-designed detection algorithms.

Evaluating a feature-region detection algorithm usually requires ground-truth information to compare with the detection results. Creating ground-truth information is a human task ranging from very labor-intensive to unfeasible, depending on the size of the code base. Another huge timesink is checking whether the results generated on test code match desired results. By comparing one detection approach with a different approach, we can run the approaches on larger code bases and evaluate them relative to each other without creating ground-truth data for the entire code base. With such a comparison, we can then detect strengths and weaknesses in the detection approaches.

1.1 THESIS GOAL

To address the problem of comparing detection approaches mentioned above, we propose a framework that runs feature-region detection approaches on code and compares their results. In the following thesis, we lay out the structure for such a framework. Then, utilizing this framework, we evaluate two detection approaches to show how the framework can be used. Finally, we showcase the approaches' strengths, weaknesses, and other insights we discovered while analyzing their disagreements on real-world projects. We also showcase the framework's ability to compare the results of detection approaches with ground-truth information. We utilize this feature to isolate interesting code patterns into small test cases to showcase each detection approach's results compared to the desired results in one example.

In the following paragraphs, we introduce the two detection approaches we compare in our evaluation.

The first approach we use for our evaluation we call *if approach*. It identifies code as feature-dependent if wrapped in the if or else case of an if statement that uses feature variables (Section 2.1.3) in its condition statement.

We refer to the second approach as *dominator approach*. It uses dominator trees and domination relationships (Section 2.3) to help identify feature-dependent code. However, before we get started, let us quickly summarize the structure of this thesis.

1.2 OVERVIEW

This thesis is comprised of three different parts. In the beginning, we provide background information about the basic functionality of the detection approaches and the LLVM environment we use to run the approaches. Next, we describe the setup of our verifier pipeline, which is the framework we propose. It encompasses preparing code for the analysis, running the approaches, and comparing their results instruction by instruction. Then we present our two-part evaluation of two detection approaches. Firstly, we perform a qualitative analysis where we compare the results of both approaches on small code samples with ground-truth information to test our framework. Secondly, we run both detection approaches on larger real-world projects to measure their precision relative to each other and observe their results. Finally, we isolate patterns we find interesting to small code examples, run the detection approaches on them, and compare their results.

2 BACKGROUND

In this chapter, we explain control flow graphs and dominator trees, which are important concepts used by the dominator-based detection approach. Additionally, we introduce the compiler framework LLVM, which we use to create a pipeline to handle the comparison of detection approaches.

2.1 CONFIGURABLE SYSTEMS

A configurable system is a software system that can be configured or modified in predefined ways by developers or the end user to change the software's functionality or appearance. A concrete example is the chrome web browser, allowing the user to configure things like its appearance, autofill options, and default search engine.

2.1.1 *Configuration Option*

A configuration option represents a predefined modification that can be made to a software system. Configurations can be made in many different ways. A program might accept configuration options via the command line, a specific configuration file, or environment variables. Every configuration option modifies a software system's functionality or appearance by, for example, setting the compression level in the `zip` command. The `zip` command, for example, accepts the configuration option -0 to -9 when used in a terminal to set the compression level.

2.1.2 *Feature*

Many definitions have been proposed to describe a feature[2]. What most of them have in common is that a feature represents some quality or aspect of a software system that is visible to the user and provides a solution to a problem.

A feature is an abstract concept or quality of a software system that has been purposefully designed and implemented in a software system. Features could be anything. Some examples are a login form on a website that allows oneself to authenticate to the website's backend servers or a scrollbar on a web browser that allows scrolling overflowing content into view. How about deciding whether a compression program should compress some files or decompresses an already compressed bundle? That would be a feature too. Features can even be made up of other features. For example, the feature of a login form is a combination of features such as making HTTP requests and listening to responses, accepting user input through the UI, and more.

To keep it simple, think of features as the tools built into a piece of software to solve a problem, no matter how small it may seem.

2.1.3 *Feature Variable*

A feature variable represents the state of a program's feature in its code. It is a variable that, for instance, can indicate what mode of operation a program takes or whether or not an optional feature is enabled.

A feature variable might, for example, represent the compression level in a compression algorithm as an integer and whether or not the algorithm is compressing or decompressing data as an enum with the values *COMPRESS* and *DECOMPRESS* indicating the algorithm's mode. In that way, both of these features of the compression algorithm are represented by their respective feature variable. Categorizing a variable as a feature variable is a somewhat arbitrary decision we make from the outside. A link between a variable and a feature might be explicitly encoded in the program by naming the variable in a way that helps identify its purpose. However, what ultimately counts is what the variable represents.

Listing 2.1: Example: Feature Model XML describing taint sources for Listing 3.1

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE vm SYSTEM "vm.dtd">
<vm name="abc" root="test/path/to/root/" commit="dabadfh">
  <binaryOptions>
A:    <configurationOption>
B:       <name>fv</name>
        <parent></parent>
        <children></children>
C:        <locations>
          <sourceRange>
D:            <path>main.c</path>
E:            <start>
              <line>3</line>
              <column>8</column>
            </start>
F:            <end>
              <line>3</line>
              <column>25</column>
            </end>
          </sourceRange>
        </locations>
        <optional>False</optional>
      </configurationOption>
  </binaryOptions>
  <numericOptions></numericOptions>
  <booleanConstraints/>
</vm>
```

2.1.4 *Feature Model*

A feature model describes the features of a software system and their relationships with each other. In Listing 2.1, we can see a feature model written in XML. Each feature we want to track is capsuled in a `configurationOption` (`A:`) tag. They are nested within the `<binaryOptions>` tag or the `<numericOptions>` tag, depending on the type of the feature. The name of the feature is in the `<name>` tag (`B:`), and we use it as an identifier to track its influence through the code. The `<locations>` and `<sourceRange>` tags (`C:`) are additions to the feature model, which provide information on where to find feature variables associated with a feature in a project. A `<locations>` can hold multiple `<sourceRange>` tags, and every `<sourceRange>` represents one feature variable associated with the feature. Part of each `<sourceRange>` is the path to the source file in the `<path>` tag (`D:`), as well as the starting and ending line and column of the variable name used in the declaration of the variable we want to track. These line and column values can be found in the `<start>` (`E:`) and `<end>` (`F:`) tags, respectively.

2.2 INTERPROCEDURAL CONTROL FLOW GRAPHS

An interprocedural control-flow graph (ICFG) is a directed graph representing all possible program execution paths [5]. It shows groups of instructions called basic blocks as nodes, and the graph's edges represent the control flow between these basic blocks. A basic block is a sequence of instructions with an entry label at the top and only one control-flow-changing instruction at the end, which we refer to as terminator. The terminator determines a node's outgoing edges, and the entry label acts as the entry point into a basic block [14]. It is important to note that a node can have multiple incoming and outgoing edges. Thus branches, loops, function calls, and other jumps in the control flow can be represented in an ICFG by adding multiple outgoing or incoming edges. We handle function calls differently from regular instructions. For each function call, we add an edge from the calling node to the start of the function's first basic block, and from every returning node of the function, we add an edge to the instruction after the function call.

Figure 2.1 shows an example of anICFG representing a function call. The ICFG begins with the first instruction in basic block A, calling the function $func$. Note that the dotted lines correspond to edges added by function call and return instructions. At B inside $func$, we define the variable $v1$ and then check whether or not $v1 > 25$. Depending on the result, we jump to basic block C or D. In the body of the if statement C, we add 5 to the variable $v1$ and jump to basic block E, while in the else case D, we subtract 5 from the variable $v1$ and then jump to basic block E. Next, the return statement gets called in basic block E, we jump out of $func$ back to the instruction after the call F, and the ICFG ends. [1] [2]

1 https://www.cs.cmu.edu/~aldrich/courses/15-8190-13sp/resources/interprocedural.pdf
2 http://cs.rpi.edu/~milanova/csci4450/Lecture12.pdf

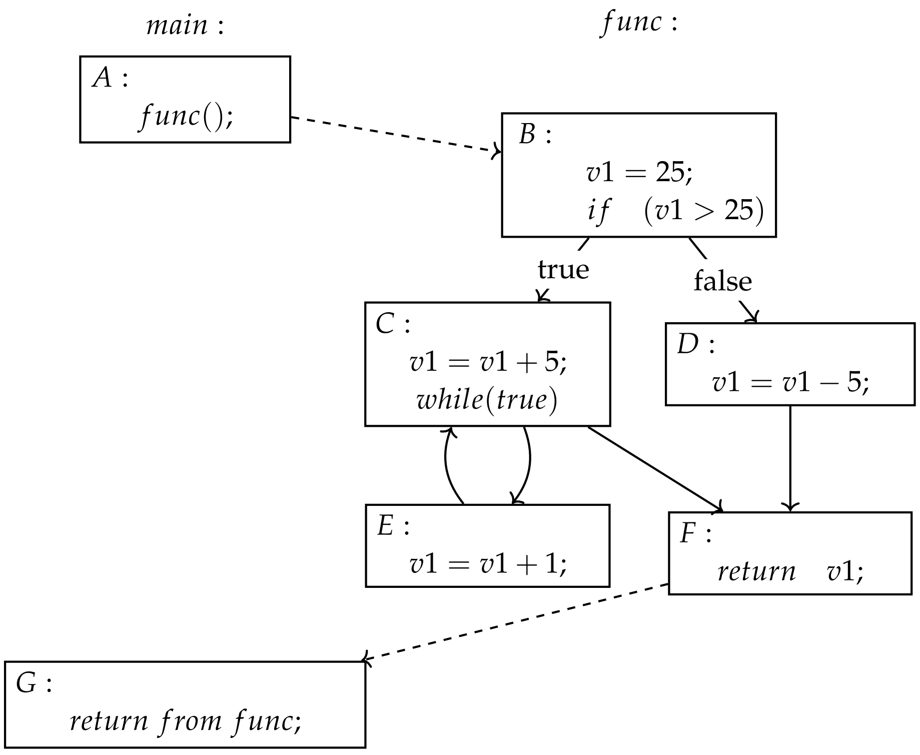


Figure 2.1: ICFG of a main function calling the function `func`

2.3 DOMINATOR TREE

A dominator tree (DT) of a program is a directed graph with the same nodes as the ICFG of the program. The DT is constructed such that every node d dominates its child nodes $n_1, \ldots, n_k$. A node d dominates node n if, in the corresponding ICFG of the program, every path from the entry node to node n has to pass through node d [1, 4].

Figure 2.2 depicts the DT of the ICFG from Figure 2.1 shows that the entry node A is dominating every other node. Node B dominates nodes C, D, and F. In particular, node F is neither dominated by C nor D because there is always a path to F over the other node. So C does not dominate F because we can reach F by going over D ($A \rightarrow B \rightarrow D \rightarrow F$), and D does not dominate F because we can reach F by going over C ($A \rightarrow B \rightarrow C \rightarrow F$). E is dominated by C, and G is dominated by F in this example. [3]

3 https://en.wikipedia.org/wiki/Dominator'_(graph_theory)

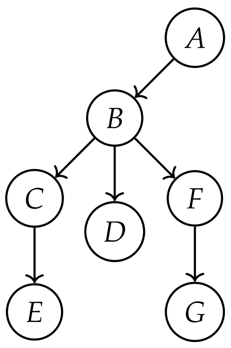


Figure 2.2: Dominator tree of the ICFG from Figure 2.1

2.4 TAINT ANALYSIS

The idea behind a taint analysis is to track the influence of a variable throughout a program by tainting or marking every instruction that uses an already tainted variable.

In order to get the process started, we define taint sources in our code, which are the seeds for the taint analysis. Taint sources are instructions we mark by hand to start the taint propagation from there. Next up, an algorithm analyses the code instruction by instruction and instructions that use a tainted variable. This step is called taint propagation, and it results in all instructions that depend on a taint source being tainted as well.

We now run through a small example to demonstrate how taint analysis works. In Listing 2.2, you can see some LLVM instructions. The taint source for our analysis is the variable `%feature_variable` marked with `A`. Marked with `B` - `K` are instructions that are tainted by taint propagation because our taint source `%feature_variable` influences them. For example, the instruction at `C` is tainted because the variable `%2` was previously tainted in `B`. `C` is an excellent example of taints propagating to other variables through variable assignment. Since `%2` is dependent on `%feature_variable` from this point on, instructions dependent on `%2` are also dependent on `%feature_variable` and are therefore tainted. `E` and `F` are tainted because the branch instruction at `D` is tainted. With that, every instruction that is part of the `if.then:` or the `if.else:` sections is influenced by a tainted variable. `G` - `K` are tainted again, similarly to `B` and `C`. Multiple variables like `%3`, `%4`, and `%5` are directly assigned to a tainted variable, and the other instructions make use of one of these variables, so they are tainted as well.

2.5 LLVM PROJECT

The LLVM project is a collection of libraries, compilers, and toolchain technologies. In general, there are three categories of tools in the LLVM architecture, frontends, optimizers, and backends [10].

Frontend tools can convert code of various languages into a universal representation used by the other two categories. The CLANG compiler, for example, can compile C or C++ code into this universal representation called LLVM intermediate representation (LLVM IR). The optimizer is a tool that runs on the LLVM IR and performs a specific task like replacing instructions to optimize code or performing analysis like detecting feature regions.

Listing 2.2: Example: Showcasing taint propagation on example instructions

```
entry:
  %feature_variable = alloca i8, align 1            ; A: Taint Source
  %sum = alloca i32, align 4
  store i32 4, i32* %sum, align 4
  %2 = load i8, i8* %feature_variable, align 1      ; B:
  %tobool = trunc i8 %2 to i1                       ; C:
  br i1 %tobool, label %if.then, label %if.else     ; D:

if.then:
  store i32 0, i32* %sum, align 4, !tbaa !2         ; E:
  br label %if.end                                  ; E:

if.else:
  store i32 2, i32* %sum, align 4, !tbaa !2         ; F:
  br label %if.end                                  ; F:

if.end:
  %3 = load i32, i32* %sum, align 4                 ; G:
  store i32 %3, i32* %retval, align 4               ; H:
  %4 = bitcast i32* %sum to i8*                     ; I:
  call void llvm.lifetime.end.p0i8(i64 4, i8* call void
      llvm.lifetime.end.p0i8(i64 1, i8* %feature_variable) #2
  %5 = load i32, i32* %retval, align 4              ; J:
  ret i32 %5                                        ; K:
```

The tasks the optimizer can run are represented as LLVM passes or short modules describing what to do with the LLVM IR. Lastly, backend tools convert LLVM IR into machine code that can be executed on a CPU. Then there are library implementations like libc++ or libclc, which implement standard libraries.

2.5.1 *LLVM Intermediate Representation*

LLVM IR is a low-level universal code representation. In LLVM IR, variables are assigned precisely once and are always defined before they are used. This property is called Static Single Assignment and is part of most intermediate representations. Thus, when a variable is reassigned, it usually gets redefined with a subscript describing its current version. Any programming language can theoretically be translated into LLVM IR as it provides basic building blocks like global variables, linking, aliases, functions, structure types, and visibility styles.[4] Tools that translate code of a specific language into LLVM IR are part of the frontend tools. However, since LLVM IR is a universal representation, it cannot handle most language-specific optimizations, and some information is lost here. However, having a low-level intermediate representation comes with two advantages.

4 https://llvm.org/docs/LangRef.html#introduction

First, a low-level representation allows optimization tools to have a high impact before the intermediate representation is changed into executable bytecode. Secondly, creating optimization tools for an intermediate representation comes with the significant advantage of reusability. Optimizations or tools utilizing LLVM IR are language agnostic and can be used on LLVM IR created from any language [11].

2.5.2 *LLVM Metadata*

LLVM metadata represents extra information in LLVM IR that can be attached to instructions and global objects. LLVM metadata provides more information about the code to tools that work on LLVM IR.[5]

Listing 2.3: Example: Metadata annotations

```
1    store i8 0, i8* %feature_variable, align 1, !1
2    !llvm.module.flags = !{!2}
3
4    !1 = !{!"metadata", !"further metadata"}
5    !2 = !{i32 4, !"Metadata added to module object 32−bit integer and a string"}
```

In Listing 2.3, we can see some example annotations being added to an instruction in Line 1 and to the module using the `llvm.module.flags` object in Line 2. The metadata annotated in each case is defined in Lines 4 and 5, respectively. The main driver for introducing metadata was creating a channel for debugging information or additional optimization information that would not influence the code structure. Not influencing the code structure is vital to prevent debugging outputs or additional information added to instructions from obscuring a bug that occurred without the debug output being enabled previously or introducing bugs when the program works fine without the added information. With LLVM Metadata, we can annotate instructions with strings or integers or reference any LLVM IR values and other metadata, which makes it very flexible.[6] To access the metadata information, we can use the NamedMDNode class in an LLVM Module. [7]

2.5.3 *LLVM pass pipeline*

The LLVM pass pipeline is flexible and consists of different analysis- and transformation passes. Each pass performs a task on a unit of LLVM IR, so on modules, functions, basic blocks, or instructions. A pass can be used to manipulate the LLVM IR, analyze the LLVM IR without changing it, produce some output, or perform any other task. Common passes are optimization passes that run over the instructions and look for patterns that can be reordered or replaced by semantically equivalent but faster/fewer instructions. [8]

5 https://llvm.org/docs/LangRef.html#metadata
6 https://llvm.org/docs/LangRef.html#metadata-nodes-and-metadata-strings
7 http://blog.llvm.org/2010/04/extensible-metadata-in-llvm-ir.html
8 https://www.llvm.org/devmtg/2014-04/PDFs/Talks/Passes.pdf

Passes can be run one after the other in any order in the pipeline. It is important to note that a pass later in the pipeline can use of the changes to the LLVM IR made by previous passes. This allows for complex tasks to be performed in a modular way by highly specialized passes that are ordered in the right way.

2.6 VARA

VaRA, or Variability-aware Region Analyzer, is an analysis framework sitting on top of the LLVM project [6]. It helps with the creation of high-level static and dynamic analyses.

2.6.1 *Feature Region*

A feature region is a code section dependent on one or more feature variables. "Dependent on feature variables" means whether or not the code section is executed depends on the value of the feature variables it is associated with. Looking at the example Listing 2.4, we can see the `%feature_variable` marked with ; Feature Variable and the associated Feature Regions marked with ; Feature Region. We want to detect these feature regions with feature-region detection algorithms and compare them with the results of a different detection approach.

Listing 2.4: Example: Feature regions as

```
entry:
  %feature_variable = alloca i8, align 1            ; Feature Variable
  %sum = alloca i32, align 4
  store i32 4, i32* %sum, align 4
  %2 = load i8, i8* %feature_variable, align 1
  %tobool = trunc i8 %2 to i1
  br i1 %tobool, label %if.then, label %if.else

if.then:
  store i32 0, i32* %sum, align 4, !tbaa !2         ; Feature Region
  br label %if.end                                  ; Feature Region

if.else:
  store i32 2, i32* %sum, align 4, !tbaa !2         ; Feature Region
  br label %if.end                                  ; Feature Region

if.end:
  %3 = load i32, i32* %sum, align 4
  store i32 %3, i32* %retval, align 4
  %4 = bitcast i32* %sum to i8*
  call void llvm.lifetime.end.p0i8(i64 4, i8* call void
      llvm.lifetime.end.p0i8(i64 1, i8* %feature_variable)
  %5 = load i32, i32* %retval, align 4
  ret i32 %5
```

Listing 2.5: Example detection results of the If approach

```
void main() {
  int fvar = 0;          // Marked Feature Variable
  int sum = 8;
  if (fvar > 0) {        // Feature Region: fvar
    sum += 5;            // Feature Region: fvar
  } else {
    sum += 2;            // Feature Region: fvar
  }
  return sum;
}
```

2.6.2 Clang *adaption*

VaRA utilizes a modified version of Clang [6]. The modifications allow Clang to read a feature model in XML format and add additional information to feature variable declarations outlined with the `<sourceRange>` tags of the feature model as taint sources.

2.6.3 *Detection Approaches*

Here we give a quick outline of the detection approaches we evaluate using our comparison framework. Both detection approaches use taint propagation to find instructions that depend on one or more feature variables. For that purpose, the detection approaches both require taint sources as input. Furthermore, these taint sources must be the declarations of feature variables for which the algorithms are supposed to find feature regions.

2.6.3.1 *If Approach*

The first approach we call $if\ approach$. It identifies code as feature-dependent if it is wrapped in the if case or else case of an if statement that uses feature variables (Section 2.1.3) in its condition statement. Listing 2.5 shows the feature regions the approach finds on a small example. We can see that the two statements inside the if/else cases are identified as feature dependent just as the if statement's condition.

2.6.3.2 *Dominator Approach*

The dominator approach uses domination relationships to identify feature regions. It looks out for the occurrence of feature variables in terminator instructions with the help of taint analysis. From there, it checks which basic block dominates other basic blocks with the dependent terminator instruction. Suppose a basic block $BB1$, for example, is dominated by $BB2$, and a feature variable is involved in determining whether or not $BB1$ is executed. In that case, all the instructions in $BB1$ depend on that feature variable. The code is never executed should the feature variable prevent a jump to the beginning of $BB1$.

2.6.4 *Taint Analysis vs Feature Region detection*

It is important to notice that taints generated by taint analysis do not correspond one-to-one to feature regions returned by the detection algorithms. A feature region is a code region whose execution depends on a feature variable's value. Taint analysis follows the usage of a variable through the code and shows us where the variable was used. However, it does not aim to identify which code regions can only be executed when the feature variable has a specific value. Detection algorithms can use taint analysis and additional logic and algorithms to determine what should be considered a feature region.

3 COMPARING DETECTION APPROACHES

Feature-region detection algorithms are a vital component in determining feature-specific code. Furthermore, they provide the information base for a deeper analysis of a program's behavior. Therefore, choosing a detection algorithm that provides accurate information about a program's feature regions is essential. This chapter presents a framework to compare feature-region detection algorithms to find weaknesses and strengths in their approaches and improve upon them.

3.1 OVERVIEW

We use the LLVM architecture to create our comparison framework. As our frontend, we use a modified version of CLANG to provide the required information for the detection approaches. A pipeline of LLVM passes handles the comparison process. The detection approaches are part of this pipeline, and each one is implemented as a separate LLVM pass. After the two detection approaches analyzed the instructions, our comparison pass compares their results and presents them to us.

In summary, our tasks consist of preparing the program for the detection approaches by marking feature variables as taint sources, annotating instructions with ground-truth values to compare to, later on, running the detection approaches on the program and then comparing the results to each other and ground-truth information added in the preparation phase. Lastly, we want to return valuable information for a qualitative and a quantitative evaluation of the detection approaches.

3.2 IMPLEMENTATION OF THE VERIFIER PIPELINE

This section is comprised of two parts. First, we describe how to preprocess C/C++ programs to prepare them for the comparison process. Next, we go through our implementation of the individual parts of our verifier pipeline. Finally, we explain the utilized LLVM passes, introduce the detection approaches, and show how we set up our verifier pipeline to run the detection approaches and compare their results.

3.2.1 *Preparation*

The first step for any new program, before we run it through the verification pipeline, is to prepare it for the detection approaches. Since the detection approaches require taint sources from which we can propagate the taints to other instructions, we need to define these taint sources before running the algorithms. Another manual job we perform in the preparation stage is adding ground-truth information to the instructions. We use this ground truth for the comparisons later in the comparison pass.

Listing 3.1: Our running example for this chapter

```c
include <stdbool.h>
int main() {
  bool feature_variable = true;
  if (feature_variable) {
    return 1;
  } else {
    return 0;
  }
}
```

MARKING FEATURE VARIABLES To be able to run the detection approaches on a program, we first have to define the seeds for their taint analysis, i.e., provide a mapping from features to the feature variable to specify which features should be tracked by the approach. In our case, we provide these taint sources by creating a *feature_model.xml* file and reading it with a modified CLANG version when compiling the C/C++ code. CLANG was modified to annotate the appropriate instructions when transforming the C/C++ code into LLVM IR after reading the *feature_model.xml* file. The basic structure of a feature-model file and its meaning is explained in Section 2.1.4. In addition, it provides a feature variable location for Listing 3.1, which is our running example for this chapter.

In each feature-model file, we describe all the relevant variable definitions whose influence we want to trace through the program and relate them to the software feature they represent. When compiling the source code with CLANG, we pass the feature-model file as an additional input, which enables CLANG to mark the instructions tied to the declaration of these variables as taint sources for the detection approaches. CLANG does this by using LLVM metadata, which allows us to define key-value pairs and add them as additional information to an object in the LLVM IR. Using this information, we can use CLANG to add our LLVM metadata to the instruction that represents this variable declaration in the LLVM IR. An example of what this looks like can be seen in Listing 2.3.

GROUND TRUTH To verify the results of both detection approaches, we need to compare them to ground-truth values. Otherwise, we would only be able to find differences in the approaches, but we would not be able to verify correctness. Therefore, for every instruction, we need to be able to tell the comparison pass which features affect the instruction. The comparison pass can then compare the ground truth against the results of the detection approaches.

We provide the ground-truth information via LLVM metadata attached to every instruction that should be part of a feature region. We construct the ground-truth values as an LLVM metadata key-value object and use `!GT` as the key. As the value, we use a list of all the names of feature variables that should affect the instruction. For example, in Listing 3.2, we see an example instruction marked with the ground-truth value `taint_source_name`.

It is important to note that we use `llvm.module.flags` to add a flag to the LLVM IR that lets us identify whether or not ground-truth values are available for a program.

Listing 3.2: Example: instruction marked with GT metadata

```
%feature_variable = alloca i8, align 1, !FVar !1
store i8 0, i8* %feature_variable, align 1, !GT !1

!llvm.module.flags = !{!2}

!1 = !{!"taint_source_name", !"another_taint_source_name"}
!2 = !{i32 4, !"withGT", i8 1}
```

Having added both taint sources and ground-truth information to a program, it is ready for the detection approaches. The approaches can now run on the LLVM IR of the program and access taint sources to identify feature regions. Additionally, we have another point of comparison with optionally added ground-truth values we can use to verify the results of the detection approaches later on.

3.2.2 *Verifier Pipeline*

After a program is prepared, i.e., feature variables are marked, and optionally ground-truth values have been added, we can feed the prepared .ll file into our verifier pipeline for analysis. Figure 3.1 shows the general setup of the verifier pipeline, which includes the detection approaches we look at in our evaluation later on. Apart from the exchangeable detection approaches, there is the comparison pass, which compares the feature regions generated by the detection approaches with each other and ground-truth values.

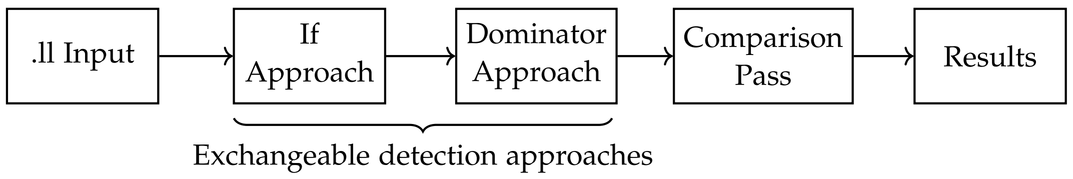


Figure 3.1: The LLVM pass pipeline we setup for our evaluation

3.2.2.1 *The detection approaches*

The first step in our pipeline is running the detection approaches. In our case, each detection approach is implemented as an LLVM pass so we can concatenate them in an LLVM pass pipeline. They make up the beginning of the pipeline, and their order is irrelevant as long as they do not change the program's instructions.

Each detection approach analyzes the program based on the feature variables marked during the preparation phase. Each detection approach maps each instruction in the LLVM IR to the feature variables detected to influence the instruction. Every detection approach creates its separate mapping, allowing us to compare these mappings later in the pipeline. The mappings are stored in an LLVM context, which acts closely to a global scope and allows us to access this mapping anywhere in our pipeline.

In the following, we explain the two approaches we compare in our evaluation.

IF APPROACH The first detection approach is called if approach. It utilizes taint analysis with feature variables as taint sources to identify tainted branching instructions created by if and else statements. Should it detect such a tainted branching instruction, the approach will return the taints on the branching instruction as the feature regions of the instructions that are part of the if and else statement associated with the branching instruction. In Listing 3.1, both return calls in Lines 5 and 7 are part of an if-then-else statement. Thus, if `feature_variable` in Line 3 is a feature variable we marked as a taint source, the if approach would mark Lines 5 and 7 as dependent on `feature_variable` and return it as the feature region for these lines.

DOMINATOR APPROACH The second detection approach is called dominator approach. The idea of this approach is to combine taint analysis with domination relationships. The approach starts by finding a basic block with tainted branch instructions and then tries to determine if it dominates another basic block. Should the basic block containing the tainted branching instruction dominate another basic block, the dominator approach will return the taints of the branching instruction as feature regions for the instructions in the dominated basic block. For example, in Listing 3.1, if we marked `feature_variable` as a taint source, the dominator approach would find that the if statement in Line 4 branches the code into an if and else case. Since it is also dependent on `feature_variable` and thus tainted by it, the approach determines dominated basic blocks next. Instructions associated with Lines 5 and 7 constitute such dominated basic blocks since they can only be reached after going through Line 4. Hence, the dominator approach would return `feature_region` as a result for the instructions associated with Lines 5 and 7.

After all detection approaches ran on the LLVM IR of the program, we now have a mapping from every instruction to the detected feature regions for every detection approach. Next, we compare these mappings using the comparison pass. We discuss the comparison pass in detail in the next section.

3.2.2.2 *The comparison pass*

The last step in the verifier pipeline is the comparison pass. After both the detection approaches have run and created their results as a mapping from instruction to feature regions in the LLVM context, the comparison pass compares their results and prints the comparison out on the console.

The comparison pass works by iterating over every instruction in the LLVM module. In each iteration, the pass fetches the corresponding feature region information from the LLVM context and ground truth from the instructions LLVM metadata. It then compares both taint lists against each other and the ground truth. Next, it prints out the comparison results and the feature regions assigned to the instruction by each detection approach. Additionally, it adds some debugging information, consisting of the path to the file and the line number from where the instruction originated. With this information, we can evaluate the performance of the two detection approaches. For example, we can track down missed taints in the analysis and find out why these taints are not set. The debug information is only available if the `-g` flag is used when compiling the code using CLANG. In the end, the comparison pass also prints out a summary of the comparison. This summary includes the absolute and the relative number of instructions placed into the same feature regions by both detection algorithms and the absolute and relative number of instructions for which the detection results match the ground truth broken down for each detection approach. All three numbers allow us to quantify how similar both approaches are and how well they perform. Finally, in Listing 3.3, we can see some exemplary output from the comparison pass.

We have seen how we generate comparison results for two detection approaches on any C/C++ program. From the preparation phase to the verifier pipeline. Adding ground-truth information, seeding the program with taint sources, running the detection approaches, and comparing their results. A big advantage of this pipeline setup is its modularity. The detection approaches can be swapped out for implementations of new or adapted algorithms, and the pipeline can easily evaluate their performance. Similarly, the comparison pass can be adapted to represent data differently or perform other or more evaluations in the future.

Listing 3.3: Example verifier-pipeline output for our running example in Listing 3.1

```
Instruction Index: 8 test.c:4
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 9 test.c:4
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 10 test.c:4
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 11 test.c:5
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 12 :0
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 13 test.c:5
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 14 test.c:7
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 15 :0
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
Instruction Index: 16 test.c:7
    Ground-Truth: ---
              IF: var       IF vs GT: Fail
       Dominator: var       Dom vs GT: Fail         Dom vs If: Ok
=====================================================================
ANALYSIS SUMMARY
Comparison between Dominator and If
100.000% (19/19) where classified the same way
0.000% (0/19) where classified differently

0.000% (0/19) where assigned a feature region by DOM but not by IF

0.000% (0/19) where assigned a feature region by IF but not by DOM

=====================================================================
```

4 EVALUATION

In our evaluation, we test the verifier pipeline by comparing the if, and dominator approaches outlined earlier on different code samples. Our evaluation is split into a qualitative and a quantitative part. The qualitative evaluation aims to test the verifier pipeline with a couple of code samples to see if it works as intended. In the quantitative evaluation, we run real-world projects through the verifier pipeline and compare the performance of both approaches. By isolating noteworthy findings from the comparison results into test cases, we showcase the strengths and weaknesses of the detection approaches that were highlighted by our framework.

4.1 QUALITATIVE EVALUATION

In this section, we test our verifier pipeline with small code samples and see if it works as expected. Thenm, we run code samples annotated with ground-truth values through the verifier pipeline and look at the generated results to determine if the pipeline works as expected and if the feature regions for each instruction are correctly compared between the detection approaches and the ground truth.

4.1.1 *Operationalization*

Each small code sample is designed to encode a specific way feature variables can be used in real-world projects and has been annotated with ground-truth information, as discussed in Section 3.2.1. Using our verifier pipeline described in Chapter 3, we analyze each code sample with the if and the dominator-based detection approaches and then compare their results with each other and the annotated ground-truth values. We then go through the results generated by the verifier pipeline and check whether or not agreements and disagreements between the detection approaches and the ground truth have been marked correctly. Additionally, we check if the detection approaches' displayed results match what we expect them to produce for the given instructions. That way, we can test if the verifier pipeline works as intended.

4.1.2 *Results*

INTRODUCATRY EXAMPLE We start with a very simple example to test the verifier pipeline. It consists of an if and an else statement. We expect both detection approaches to deliver the same results and match the ground truth for our verifier pipeline to work correctly.

The source code with information on what we marked as ground truth can be seen in Listing 4.1 under CODE and GT, respectively. Under IF (if approach) and DOM (dominator approach) to the right of the ground truth are the results of the approaches, with each row being dedicated to a different line of code. Taint sources given to the detection approaches are marked with _source above the results for every source given to the approach. In this case, the feature's name is fv, and the variable given as the taint source is also named fv. Lines marked with fv indicate that the approach marked this line of code as part of the feature region associated with the feature variable fv_source. Going through the results from the comparison, we can see that both approaches correctly identify every feature region and perfectly match the ground truth. So in a simple example, the verifier pipeline worked as intended, and both approaches deliver the same results and match the ground truth.

```
CODE                                 | GT          IF          DOM
#include <stdbool.h>                 |
                                     |
int main(int argc, char *argv[]) {   |
  bool fv = argc > 5;                | fv_source   fv_source   fv_source
  int sum = 4;                       |
  if (fv) {                          | fv          fv          fv
    sum = 0;                         | fv          fv          fv
  } else {                           | fv          fv          fv
    sum = 2;                         | fv          fv          fv
  }                                  |
  return sum;                        |
}                                    |
```

Listing 4.1: Comparison results of an If-Then-Else statement.

OTHER CONTROL-FLOW CHANGES Our next example uses a switch statement to avoid detection by the if approach. This allows us to test disagreements between the detection approaches. For the verifier pipeline to work, we expect the if and the dominator approach to disagree for the entire switch statement, where the code depends on our feature variable.

The source code of the second example, the taint sources, and the detection results can be seen in Listing 4.2. The taint source given to the detection approaches can be seen in Line 2, and we use its feature variable fv once in Line 5 to make the switch statement dependent on it. As we can see in the DOM and IF columns to the right of the code, the if approach does not identify any feature regions, while the dominator approach discovers all of them as expected. Since the if approach only considers code regions to be feature regions if they are inside an if or else block, alternative control-flow statements avoid detection altogether. This apparent weakness would make the if approach very situational and, in this case, allowed us to test our verifier pipeline for disagreements between detection approaches.

```
CODE                                GT          IF          DOM
int main() {
  int fv = 44;                      fv_source   fv_source   fv_source

  int inst;
  switch (fv) {                     fv          🟥          fv
  case 32:
    inst = 0;                       fv          🟥          fv
    break;                          fv          🟥          fv

  default:
    inst = 1;                       fv          🟥          fv
    break;                          fv          🟥          fv
  }
  return inst;
}
```

Listing 4.2: Comparison results of a switch statement.

Now we have seen both agreements and disagreements between the detection approaches. The verifier pipeline has been tested and works as expected, given the results of these tests. The next step is using the verifier pipeline to learn more about the detection approaches. The following section explores real-world projects using the verifier pipeline and outlines what we find in the results.

4.2 QUANTITATIVE EVALUATION

In this chapter, we take a closer look at the detection approaches. By running real-world projects through the verifier pipeline, we generate detection results for hundreds of lines of code we analyze and evaluate. We aim to get an idea of how close the results of both detection approaches are to each other and analyze cases in which they disagree to find possible weak points in the detection approaches. Later on, we isolate interesting findings into small code samples and annotate them with ground-truth information to find out what causes these weak points if they are, in fact, weak points.

4.2.1 *Operationalization*

For the quantitative evaluation, we marked feature variables for the projects: xz, bzip2, lz4 and gzip. Then we ran each project through our verifier pipeline (Figure 3.1). Because of the size of the projects, we have not added any ground-truth values to them. Thus, we focus our quantitative evaluation on the comparison between the results of the detection approaches.

Disagreements with results from one approach but without results from the other approach are of particular interest. These instructions may reveal a possible blind spot in the detection approach without results.

4.2.2 *Results*

This section presents the disagreements we found when running the projects xz, gzip, lz4 and bzip2 through the verifier pipeline. First, we briefly describe general findings. Then we analyze significant disagreements in isolated test cases.

Table 4.1: Aggregate Statistics

Project	Both Agree	Both Disagree	Only Dom	Only If	Total Number of instructions
xz	68.322%	31.678%	12.068%	8.526%	10872
gzip	58.564%	41.436%	28.211%	4.150%	18216
lz4	99.539%	0.461%	0.025%	0.437%	36412
bzip2	92.241%	7.759%	5.232%	0.403%	29797

In Table 4.1, we present some data about our results. In particular, we see the percentage of instructions where both approaches agree with their detected feature regions. The higher this percentage, the closer the detection approaches are to detecting the same feature regions for an entire project. Two more percentages are written in the Only Dom and Only If columns. They represent the percentage of instructions only one of the detection approaches assigned feature regions to, while the other one assigned none. Our first intuition when introducing these statistics was that the Only If percentage would be 0%, and the Only Dom percentage would be greater than 0%. The idea is that the dominator approach should also be able to detect everything the if approach detects. However, the dominator approach should also detect much more feature regions than the if approach since it can find feature regions in switch statements and loops, for example. Seeing that the If Only percentage is greater than 0%, we looked at most of these cases in detail to find out whether or not the if approach found something the dominator could not detect. However, in most cases, we find the if approach assigning feature regions without visible dependencies. For example, in Listing 4.3, we can see a code snipped from the bzip2 project where the if approach assigns the feature region *verbosity* to the instructions generated from Line 230 in decompressed.c. From Line 224 to 235, it is the only line either approach assigns a feature region. The only difference between Line 230 and other almost identical function calls is the previously defined constant value BZ_X_BLKHDR_4. It makes no sense why only instructions associated with this function call are assigned a feature region but not those in Lines 226, 228, 232, or 234. In general, we could not find any link between one of the supposedly detected feature regions and the instructions marked by the if approach as dependent on these features.

```
CODE
⋮
if (uc == 0x17) goto endhdr_2;
if (uc != 0x31) RETURN(
    BZ_DATA_ERROR);
GET_UCHAR(BZ_X_BLKHDR_2, uc);
if (uc != 0x41) RETURN(
    BZ_DATA_ERROR);
GET_UCHAR(BZ_X_BLKHDR_3, uc);
if (uc != 0x59) RETURN(
    BZ_DATA_ERROR);
GET_UCHAR(BZ_X_BLKHDR_4, uc);
if (uc != 0x26) RETURN(
    BZ_DATA_ERROR);
GET_UCHAR(BZ_X_BLKHDR_5, uc);
if (uc != 0x53) RETURN(
    BZ_DATA_ERROR);
GET_UCHAR(BZ_X_BLKHDR_6, uc);
if (uc != 0x59) RETURN(
    BZ_DATA_ERROR);
⋮
```

```
IF
⋮
verbosity
⋮
```

Listing 4.3: Example: The if approach falsely detects an isolated feature region

```
CODE
⋮
if (zfile != NO_FILE) {
  read_buf  = file_read;
}
⋮
```

```
IF
⋮
decompress, no_name, force
decompress, no_time, no_name, force
decompress, no_time, no_name, force
⋮
```

Listing 4.4: Example: The if approach detects `no_time` out of nowhere

Additionally, sometimes the if approach detected one or more feature regions than the dominator approach on the same instruction. While in these cases, it is mainly effortless to identify the instruction's dependency on the feature regions both approaches detect, we again fail to detect any dependency on the additional feature regions the if approach claims to detect.

In the example given in Listing 4.4, the dominator approach detects `decompress, no_name`, and `force` for all three lines of code, but the if approach also adds `no_time` halfway in. Thus we conclude that the if approach is subject to one or more bugs that cause feature regions to be associated with instructions that are not dependent on them.

As expected, the Dom Only percentage is greater than 0%, and a deeper analysis shows that switch cases and loops are being detected by the dominator approach and contribute to this percentage. Even more interesting is what we should have expected at first glance but did not. Contributing to this Dom Only percentage are code patterns we call *implicit else case*, and we break them and other interesting findings down in isolated test cases shown in Section 4.2.3.

4.2.3 *Observations*

This section presents interesting discoveries we made while analyzing the results generated from the real-world projects in Section 4.2. The discoveries were made while looking for disagreements between the two detection approaches, which leaves out a large chunk of data where both approaches agree on their results. For example, in theory, the two approaches can agree on the feature regions they assign to instructions but be wrong in their analysis. Unfortunately, we only combed through some of these cases and only if both approaches assigned any feature regions in the first place. Focusing on the disagreements So, instead, we focused on both approaches' disagreements when detecting feature regions. They have the advantage of being apparent sections of code in which at least one approach fails to detect what it should.

4.2.3.1 *Boolean Statements*

We start with a code pattern that makes a repeated appearance in the XZ project where we first discovered it. It shows a, at first, unexpected impact of boolean statements on the dominator approach and detection failure for the if approach.

In the example code shown in Listing 4.5, we introduce two feature variables, `fv1`, and `fv2`, and use them in the condition of an if statement coupled with a boolean or. Because the if statements condition depends on both feature variables, we expect both features to influence the code region inside the if statement, creating a feature region for both features. However, the if approach only returns the last feature in the boolean statement for Lines 4 and 5. On the other hand, the dominator approach only returns the first feature for some instructions associated with Line 4 but correctly assigns both to the rest of the instructions associated with Lines 4 and 5. We expressed this behavior using the braces `fv1` `(, fv2)` to denote that `fv2` was assigned additionally to fv1 on other instructions associated with this line of code. The dominator approach's peculiar behavior towards the if condition can be explained by a lazy evaluation method used for boolean statements called short-circuit evaluation. It only evaluates the second parameter of the boolean statement when the first parameter has been evaluated as false. As such, the LLVM IR contains one instruction only dependent on the first feature variable in the boolean statement.

For the if approach, not detecting the influence of feature variables when boolean operations are involved is a significant problem that could lead to many of wrong detections simply because it is common to use boolean operations in the conditions of if statements. Since if statements are the only control flow detectable by the if approach in the first place, this problem further lowers the usability and robustness of the if approach. For the dominator approach, on the other hand, we have found an example of a correct analysis, which seemed wrong to us as observers at first.

```
CODE                                          | GT           IF           DOM
int main(int argc, char *argv[]) {            |
  int fv1 = 1;                                | fv1_source   fv1_source   fv1_source
  int fv2 = 2;                                | fv2_source   fv2_source   fv2_source
  if (fv1 == 3 || fv2 == 3) {                 | fv1, fv2     fv2          fv1 (, fv2)
    return 0;                                 | fv1, fv2     fv2          fv1, fv2
  }                                           |
}                                             |
```

Listing 4.5: Comparison results of boolean statements used in an if condition.

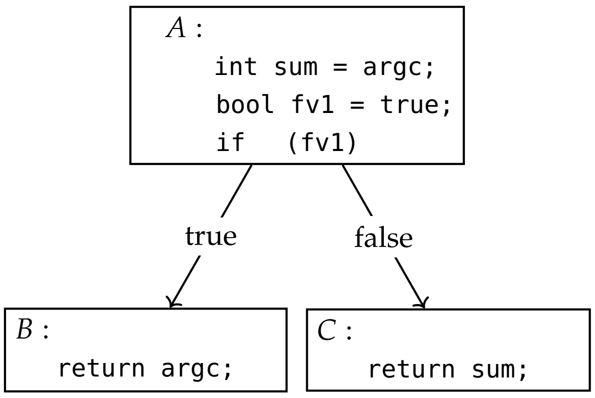
4.2.4 *Implicit Else Cases*

Implicit else cases are another huge problem for the if approach that greatly reduces its robustness. The dominator approach, on the other hand, handles implicit else cases very well. Moreover, it presents many unique ways of implicitly creating feature dependencies in code to us in the data from Section 4.2. We start by presenting the most obvious case of an implicit else case by using if statements and later describe a couple more exotic variations we have found in the data.

In the example shown in Listing 4.6, we introduce a feature variable `fv1` in Line 2, use it in the if condition in Line 6 to create a feature region, and return inside the if statement in Line 7. After the if statement, we return in Line 9 to create some code after the if statement. If we analyze the program now, we can see that the man function has two execution branches. The first branch is executed when the if condition in Line 6 is true, and we return the `sum` variable after the if statement. The second branch is executed when the if condition is false, and we return directly after the if statement. If we look at the control flow of this program, we have, in effect, created an else case for the if statement in Line 6. That is because we can only reach the code after the if statement if the if condition is false. So we have created an implicit else statement with our arrangement of return statements. However, implicit else statements are not detected by the if approach even though they fulfill a similar job since the if approach looks for constructs created in the LLVM IR by the *else* or if keywords to find possible feature regions. So all the code after the if statement is not seen as viable to be a feature region. Nevertheless, we achieved the same control flow as if we had used an else statement and created feature-dependent code.

```
CODE                                    | GT          IF          DOM
#include <stdbool.h>                    |
                                        |
int main(int argc, char *argv[]) {      |
  int sum = argc;                       |
  bool fv1 = true;                      | fv1_source  fv1_source  fv1_source
  if (fv1) {                            | fv1         fv1         fv1
    return argc;                        | fv1         fv1         fv1
  }                                     |
  return sum;                           | fv1         🟥          fv1
}                                       |
```

Listing 4.6: Example for an implicit else case



```
int sum = argc;
bool fv1 = true;
if   (fv1)
```

```
return argc;
```

```
return sum;
```

Figure 4.1: Control flow of the if statement in Listing 4.6

Another way of creating implicit feature dependencies we found in the data generated from real-world projects is using the `break` and `continue` keywords. Using `break`, as shown in Listing 4.7 on Line 7, for example, makes `break` dependent on the feature `fv1`. Additionally, we create an implicit dependency on `fv1` for the nodes *E*, *F*, and *A*, which correspond to Lines 8 and 6. Their execution depends on the value of `fv1` because if `fv1` is true, we break out of the while loop, never revisiting Lines 8 and 6.

Similarly, using `continue`, as shown in Listing 4.7 on Line 8, creates a dependency on `fv2` for Line 6, which is now implicitly dependent on `fv2` because of the `continue` statement and `fv1` because of the `break` statement.

To summarize, the if approach struggles with code structures that mimic else statements without invoking the `else` keyword. That is because it relies on structures generated in the LLVM IR when the `else` keyword is invoked to detect feature regions. Creating these implicit else cases can be achieved with statements that invoke jump instructions in LLVM IR, such as `return`, `break`, `continue`, and `goto`. The dominator does a good job of identifying these implicit else cases but has its limits. In our next example, we explore these limits using simple if and cleverly placed return statements.

```
CODE                                                  GT          DOM
1:  #include <stdbool.h>
2:
3:  int main(int argc, char *argv[]) {
4:    bool fv1 = true;                                fv1_source  fv1_source
5:    bool fv2 = true;                                fv2_source  fv2_source
6:    while (true) {                                  fv1, fv2    fv1, fv2
7:      if (fv1) break;                               fv1         fv1
8:      if (fv2) continue;                            fv1, fv2    fv1, fv2
9:    }
10:   return argc;
11: }
```

Listing 4.7: Example for implicit dependencies using break and continue

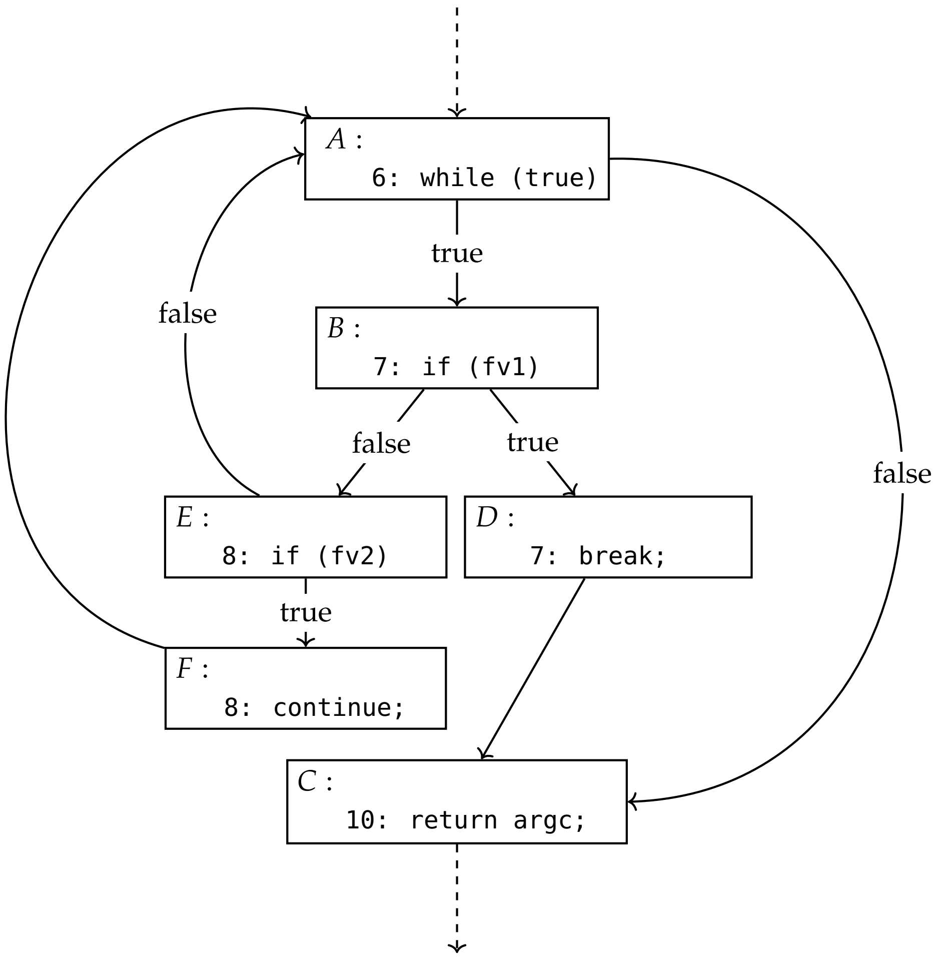


Figure 4.2: Control flow of the while loop in Listing 4.7

4.2.5 *Not dominated, but feature dependent*

We discovered the following problem in the last example utilizing the while loop and the break/continue statements. However, it is much simpler to understand what is happening using the following example instead. In essence, we create feature-dependent code using an implicit else case but in a way that avoids detection by the dominator approach. To accomplish this, we nest an if statement with a feature variable in its condition in another if statement. Then all we have to do is return in the nested if statement. The code and the corresponding CFG can be seen in Listing 4.8 and Figure 4.3. Because of how we constructed the program, the dominator approach does not identify the code after the if cases as feature dependent. However, its execution depends on whether or not we executed the return in Line 8, thanks to the feature variable utilized in Line 7. In the CFG, we can see that node D is not dominated by B since we now have a direct path from A to D. Something similar happened in the last example with the while loop. Although the while condition is still correctly identified as feature-dependent, regular code that runs inside the while loop before we create our implicit else case will not be identified as feature dependent because it is not being dominated.

CODE	GT	DOM
`1:  #include <stdbool.h>`		
`2:`		
`3:  int main(int argc, char *argv[]) {`		
`4:    bool fv1 = argv[0];`	fv1_source	fv1_source
`5:    bool no_feature = argv[2];`		
`6:    if (no_feature) {`		
`7:      if (fv1) {`	fv1	fv1
`8:       return 1;`	fv1	fv1
`9:      }`		
`10:   }`	fv1	fv1
`11:   return 0;`	fv1	🟥
`12: }`		

Listing 4.8: Example for a feature dependency without a domination relationship

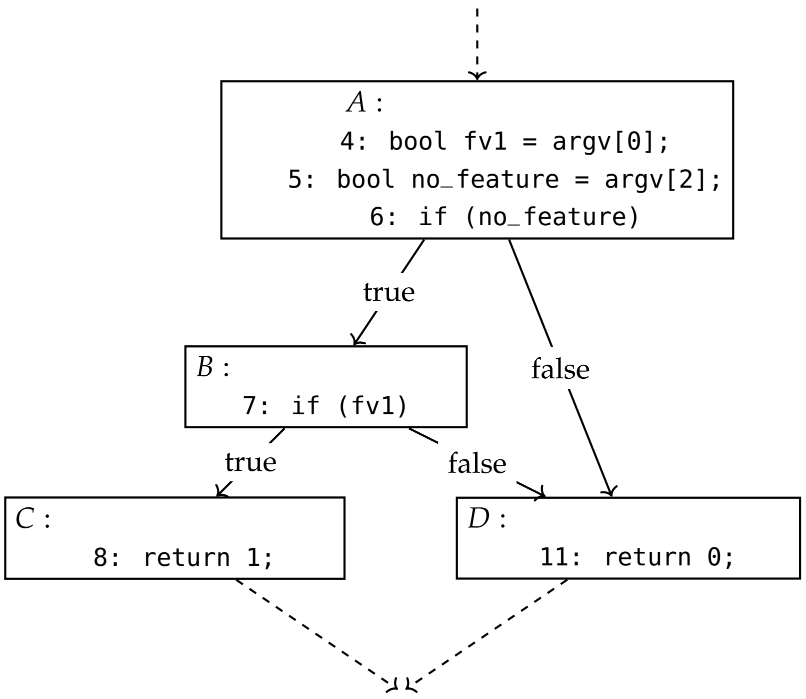


Figure 4.3: Control flow of the if statements from Listing 4.8

We conclude our evaluation of the comparison data generated from real-world projects. All in all, we learned quite a bit about the detection approaches. We identified and learned something about their weaknesses, such as implicit else cases and boolean statements, as well as their strengths, such as identifying feature dependencies independent of the control-flow statement used or handling most implicit feature dependencies correctly.

5 CONCLUSION

In the following, we conclude our thesis by summarizing our work, putting it into the current research context, and advising how future work could further improve on our work.

5.1 SUMMARY

This thesis proposed a framework for comparing feature-region detection approaches with each other and ground-truth information. In our evaluation, we demonstrated that our framework helps to identify strengths and weaknesses in detection approaches by highlighting disagreements in their results and thus drawing attention to code patterns. These code patterns exploit a weakness in a detection approach or expose a bug in its implementation. An evaluation of the framework on real-world projects with two detection approaches allowed us to identify several code patterns that caused at least one of the approaches to be unable to detect the correct feature regions. The framework can only detect false detection results when a ground truth has been established. Otherwise, the framework can only show the differences between the two detection results and cannot verify their correctness. Nevertheless, we have showcased that the ability to run and compare detection approaches on even large real-world projects opens up the possibility to test a detection approach on various configurable systems using different feature encodings.

IF APPROACH Starting with the if approach, we have shown in our evaluation that the detection approach lacks general robustness. Its only strength is simplicity. We cannot recommend this approach for general use, as it cannot detect the wide variety of ways feature variables can be encoded in real programs. Only finding feature regions when they are part of an if or else statement is insufficient. A feature-region detection approach needs to cover more implementation patterns to be considered helpful in a general sense. In addition, we have shown that the approach cannot detect implicit else cases (Section 4.2.4). This flaw makes it an unreliable pick, even if the only control flow structures used are if and else statements.

DOMINATOR APPROACH For the dominator approach, our evaluation shows much more promise. Since its ability to detect feature regions is not limited by a control-flow statement but is instead based on general control-flow data, the dominator approach is much more robust than the if approach. That makes the dominator approach a better pick for use on real-world projects where the encoding of features is unknown. However, the implementation of the detection approach has its challenges. During our evaluation, we discovered a way to bypass the dominator's analysis utilizing nested control-flow statements to prevent direct domination relationships from forming. This structure type is relatively easy to create and did occur a couple of times in the real-world projects we analyzed.

Looking at the summary of the evaluation results, we can confidently conclude that the dominator approach is the more robust detection approach of the two. Its strategy makes it a fundamentally better choice than the if approach on any project that is not known to encode every feature region with if and else statements. However, even if it does, we have shown that the if approach cannot handle implicit else statements created by returning inside an if statement (Section 4.2.4). Our comparison framework made it feasible and straight forward to sift through disagreements between the two detection approaches and directly reference the instructions and lines of code the results are generated for.

5.2 RELATED WORK

Feature-region detection approaches are utilized in many previously proposed tools and techniques for analyzing configurable systems [7, 8, 12, 15–20]. Our work can help find weaknesses in their detection approaches, which in turn can help improve the underlying feature region data for their analyses. For example, white-box performance modeling tools utilize static [17] and dynamic [18] feature-region detection approaches. Performance modeling of configurable systems aims to create performance-influence models explaining the impact of configuration options and their interactions on the performance of a system. To build these performance-influence models, they track configuration options in a system through feature-region detection and analyze the impact of different configurations on the program's performance given the same workload and environment. A static taint-analysis tool by Lillack, Kästner, and Bodden [12] automatically tracks load-time configuration options in Android applications. Their approach stands out because their tool detects feature regions and the configurations necessary to execute a feature region. In essence, they have developed a detection approach similar to the if approach or dominator approach used in this thesis. Garvin and Cohen [7] propose white-box criteria for an interaction fault and explore how often feature interactions are responsible for reported real-world faults. They seek to understand what constitutes an interaction fault between configuration options and develop a working definition based on a known mapping from feature to code. Their detection accuracy of interaction faults is directly dependent on the quality of the accuracy of the mapping from feature to code the utilized. Toman and Grossman [16] introduced a tool to detect errors introduced by dynamically updating configuration options. They utilized dynamic analysis to detect stale data computed from older configurations or inconsistent data to detect errors from reconfigurations. Finally, Jin et al. [9] found that bigger applications consist of multiple languages, which implies that static analysis must work regardless of the programming language barrier that might exist. Their findings relate to Bodden [3], who argues that the performance of static-analysis tools are held back by the general-purpose languages they are implemented in, in that both push future static analysis away from general language implementations and towards more specialized solutions that are ideally implemented in an intermediate representation supported by many languages. Our framework uses these findings by running on LLVM IR.

5.3 FUTURE WORK

This thesis uses ground-truth data added to small code examples by hand. However, that is very tedious and unrealistic to do on a real-world project with multiple hundreds of feature variables. That means we are limited to comparing detection approaches to each other on these larger projects. An interesting idea for future work is automatically generating ground-truth values based on code-coverage reports. The difference between a coverage report generated without a specific feature enabled and a report with that feature enabled should arguably result in a rough outline of the associated code region. Developing a way to create these code-coverage-based ground-truth values, evaluating their viability, and comparing the differences between this approach and handcrafted ground truth might provide valuable insights to improve feature region detection further.

Secondly, using the comparison approach presented in this thesis, future work that focuses on tuning existing algorithms to match ground-truth data better could provide insights into how specific tuning parameters affect the general results of a detection approach. On the one hand, a tuning parameter can enable a user to tune the algorithm to his or her needs. On the other hand, it can improve the general detection approach of an algorithm. A tuning parameter could, for example, be a boolean value deciding how the approach handles lazily evaluated statements, either including the lazy evaluated part in the feature mapping or excluding it. Secondly, tuning parameters can also describe how the approach handles feature-region detection in general. For example, tuning the if approach might involve making the algorithm aware of implicit else statements that are currently not detected by the algorithm.

BIBLIOGRAPHY

[1] Alfred Aho, Ravi Sethi, and J Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley 2nd edition, 2007.

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

[3] Eric Bodden. "Self-adaptive static analysis." In: *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 2018, pp. 45–48. URL: https://arxiv.org/pdf/1710.07430.pdf.

[4] Martin D Carroll and Barbara G Ryder. "Incremental data flow analysis via dominator and attribute update." In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 274–284. URL: https://dl.acm.org/doi/pdf/10.1145/73560.73584.

[5] Saumya K Debray and Todd A Proebsting. "Interprocedural control flow analysis of first-order programs with tail-call optimization." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.4 (1997), pp. 568–585. URL: https://dl.acm.org/doi/pdf/10.1145/262004.262006.

[6] Florian Sattler, Sebastian Böhm, Philipp Schubert, Norbert Siegmund, and Sven Apel. "SEAL: Integrating Program Analysis and Repository Mining." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2023, To appear).

[7] Brady J Garvin and Myra B Cohen. "Feature interaction faults revisited: An exploratory study." In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE. 2011, pp. 90–99.

[8] Sten Grüner, Andreas Burger, Hadil Abukwaik, Sascha El-Sharkawy, Klaus Schmid, Tewfik Ziadi, Anton Paule, Felix Suda, and Alexander Viehl. "Demonstration of a toolchain for feature extraction, analysis and visualization on an industrial case study." In: *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*. Vol. 1. IEEE. 2019, pp. 459–465.

[9] Dongpu Jin, Xiao Qu, Myra B Cohen, and Brian Robinson. "Configurations everywhere: Implications for testing and debugging in practice." In: *Companion Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 215–224. URL: https://dl.acm.org/doi/abs/10.1145/2591062.2591191.

[10] Chris Lattner. "Introduction to the llvm compiler infrastructure." In: *Itanium conference and expo*. 2006.

[11] Chris Lattner and Vikram Adve. "The llvm compiler framework and infrastructure tutorial." In: *Languages and Compilers for High Performance Computing: 17th International Workshop, LCPC 2004, West Lafayette, IN, USA, September 22-24, 2004, Revised Selected Papers 17*. Springer. 2005, pp. 15–16. URL: http://nozdr.ru/data/media/biblio/kolxoz/Cs/CsLn/Languages%20and%20Compilers%20for%20High%20Performance%

20Computing,%2017%20conf.,%20LCPC%202004(LNCS3602,%20Springer,%202005)(ISBN%20354028009X)(494s).pdf#page=23.

[12] Max Lillack, Christian Kästner, and Eric Bodden. "Tracking load-time configuration options." In: *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 2014, pp. 445–456. URL: https://www.bodden.de/pubs/lkb14tracking.pdf.

[13] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. "On essential configuration complexity: Measuring interactions in highly-configurable systems." In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 483–494.

[14] Sandra Rapps and Elaine J. Weyuker. "Data Flow Analysis Techniques for Test Data Selection." In: *Proceedings, 6th International Conference on Software Engineering, Tokyo, Japan, September 13-16, 1982*. IEEE Computer Society, 1982, pp. 272–278. URL: http://dl.acm.org/citation.cfm?id=807769.

[15] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. "A classification and survey of analysis strategies for software product lines." In: *ACM Computing Surveys (CSUR)* 47.1 (2014), pp. 1–45.

[16] John Toman and Dan Grossman. "Staccato: A bug finder for dynamic configuration updates." In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.

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

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

[19] Junyong Wang, Thar Baker, Yingnan Zhou, Ali Ismail Awad, Bin Wang, and Yongsheng Zhu. "Automatic mapping of configuration options in software using static analysis." In: *Journal of King Saud University-Computer and Information Sciences* 34.10 (2022), pp. 10044–10055.

[20] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. "Early Detection of Configuration Errors to Reduce Failure Damage." In: *OSDI*. Vol. 10. 2016, pp. 3026877–3026925.