FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	A Modular and Extensible Tool for Software Fault Localization
Student:	Petr Nevyhoštěný
Supervisor:	Ing. Petr Máj
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Familiarize with the existing research and methods in the field of software fault localization.
Design and implement a new fault localization tool capable of implementing the existing techniques that will be extensible with respect to new localization techniques.
Demonstrate the functionality by implementing frontends for the C and Python programming languages.
Perform an evaluation of the implemented tool on existing datasets.

References

Will be provided by the supervisor.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 11, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

A Modular and Extensible Tool for Software Fault Localization

Bc. Petr Nevyhoštěný

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

June 7, 2020

Acknowledgements

I would like to express my sincere gratitude to my supervisor Ing. Petr Máj, whose valuable feedback and important advice helped to form the path while writing the thesis and developing the tool.

Another huge acknowledgement belongs to people at the university where I discovered and learned plenty of beauties of the computer science field.

Many thanks go also to my friends for helping me to have a life I mostly enjoy to live. And last but not least, I would like to let my parents know that I am grateful for what they allowed me to achieve thanks to their great support.

Declaration

Citation of this thesis

Nevyhoštěný, Petr. *A Modular and Extensible Tool for Software Fault Localization.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Lokalizace chyb je považována za jeden z nejvíce únavných a časově náročných úkolů při vývoji software. Přesto je stále často prováděna manuálně. Lokalizace softwarových chyb (anglická zkratka SFL) je oblast výzkumu zabývající se vývojem automatizovaných technik pro usnadnění této aktivity. Navzdory množství práce, která byla tomuto výzkumu věnována, nesplňují aplikace navržených metod kompletně potřeby praxe.

Tato magisterská práce proto popisuje nový nástroj a framework pro lokalizaci softwarových chyb s názvem *Aardwolf*, jehož hlavním cílem je usnadnit použití SFL technik v praxi. Toho je dosaženo velmi modulárním designem, umožňujícím snadnou rozšiřitelnost, a aplikováním doporučení, která lze nalezout v publikovaných uživatelských studiích.

Pro demonstraci možností nástroje Aardwolf byla implementována trojice různých SFL technik a frontendy pro programovací jazyky C a Python. Integrace do dvou větších softwarových projektů byla popsána jako ukázka aplikovatelnosti. Předběžné výsledky ukazují, že je efektivita nástroje srovnatelná s literaturou. Zaměření na uživatelskou přívětivost a rozšiřitelnost řešení je ovšem významné zlepšení oproti současnému stavu v tomto odvětví.

Klíčová slova lokalizace chyb v software, debugování, testování, spektrum programu, pravděpodobnostní grafický model, programové invarianty, nástroj, framework

Abstract

Localization of faults has been recognized as one of the most tedious and time-consuming tasks in software development. Yet it is still performed mostly manually. Software fault localization (SFL) is a research field that studies the development of automated techniques helping developers with this activity. Despite the amount of effort put into the research, applications of proposed methods do not fully meet the practical needs.

To that end, this thesis describes *Aardwolf*, a new fault localization tool and framework whose main goal is to ease the usage of SFL techniques in real-world projects. This is achieved by highly modular design allowing convenient extensibility, and by applying recommendations that can be found in published user studies.

To demonstrate Aardwolf's capabilities, three different SFL techniques and frontends for C and Python programming languages were implemented. Integration into two larger software projects was described to present its applicability. Preliminary results show that the effectiveness is comparable with the literature. The focus on user experience and tool's extensibility is however a major improvement over the current state in the field.

Keywords software fault localization, debugging, testing, program spectrum, probabilistic graphical model, program invariants, tool, framework

Contents

List of Figures

List of Tables

Introduction

Computer software is ubiquitous in today's world, present not only on our personal computers and smartphones, but driving a wide range of things, from smart devices, such as televisions or watches, to industrial machines to safety-critical systems in medicine or aeronautics, for example. We interact with software on a daily basis and we are more or less dependent and influenced by it to a large extent. Its usage and adoption are growing at a fast pace thanks to developments of physical hardware as well as recent advancements in artificial intelligence and machine learning that enable automation of a variety of tasks.

Unsurprisingly, this trend has been accompanied by a large increase in the scale and complexity of the software. This fact raises the challenges for maintaining software quality. In particular, the growth of software complexity results in more software faults, in the field referred to as bugs. A fault is an incorrect behavior of the program, introduced by a programmer's mistake or misunderstanding, which causes it to produce unexpected output or behave in unintended ways.

The process of determining the exact nature and location of the fault and then correcting the program is called debugging [1]. Some studies show that, in a typical programming project, programmers spend approximately 50 % of their time doing debugging and testing [2, 1].

The first step in debugging – understanding the error and locating its cause – takes roughly 95 % of the time in the process [1]. If we connect this number to the ratio of time spent in debugging mentioned earlier, we can conclude that this activity is considerably time-consuming. When carried out manually, it has been recognized as to be greatly tedious [3] and mentally demanding [1]. The reason is that the fault may be present virtually anywhere in the program code that affects the erroneous behavior and requires the ability to comprehend the entire codebase or its significant part.

Software fault localization (SFL) is a research field that aims to help programmers with this effort by developing techniques that partially or fully automate the localization of faults in the source code. In recent years, a broad spectrum of approaches have been proposed and the number of publications on this topic per year is growing [3]. We refer to surveys [3, 4] for a comprehensive overview. The motivation for fault localization tools is obvious: provide the developer with a helpful report of suspicious program elements, events, or states to help them to reduce the amount of time and effort in this tedious and costly task. A showcase where such a tool can aid the debugging process is given in Chapter 1.

There are two general categories of techniques in automated fault localization [5]: *dynamic* (or test-driven) and *static* (or formal). Techniques of the first type execute the subject program using a set of tests, while techniques of the second type perform their reasoning solely using static analysis. In this thesis, we focus on the first category.

In general, these techniques assign a score to program elements (e.g., statements or methods[1]) which corresponds to the degree of suspiciousness of causing the failure as determined by the algorithm. The developer is then provided with the list of program elements sorted in descending order of their suspiciousness score. The more the actual faulty element is closer to the top of the list, the faster it is discovered by the developer, hence the more effective the fault localization technique is.

Despite the amount of work that has been put into the research of fault localization and high demand for it (more than 97 % of respondents in the survey [6] consider it essential or worthwhile), the applications of the techniques do not completely fulfill the requirements and needs of the developers. In particular, they make assumptions that do not hold in practice [7] and lack the integration with popular IDEs [6].

There are two main assumptions that are false. First, fault localization techniques usually assume perfect bug understanding, that is, the developer can detect and understand the fault solely by examining the program element in isolation. However, this is not really true. Without knowing why the element is marked as suspicious and any context, the programmer usually cannot decide if the element is actually faulty. Second, the evaluation of the techniques is often performed such that the rank of the faulty element is compared relative to the size of the program. Although the result like 2 % of lines needed to be inspected to find the fault might seem successful, in case of a program with 5,000 lines of code, which can be considered as a medium-sized program [8], it means to inspect roughly 100 statements. Such a number is not acceptable by developers [7] as they would switch to the traditional manual debugging process instead.

The findings and conclusions of performed user-focused studies [7, 6, 9] can be summarized in the following points:

- *Large demand.* Current fault localization tools are mostly research prototypes, therefore they are not used in practice. However, there is a

[1]We use the terms method and function interchangeably in this thesis.

significant willingness to adopt a fault localization technique that satisfies certain criteria. These criteria are discussed in the subsequent points.

- *High adoption barriers.* Most users consider a technique helpful if they find the actual fault in the top 5 or so positions in the suspiciousness list, require it to be successful at least 75 % of the time, and expect to be able to run the tool on programs of size hundreds of thousands of lines of code in less than a minute. For research, this particularly means to use absolute ranking metrics and discussing scalability in the experimental evaluation.

- *Trustworthiness/reliability.* It is frustrating for the user to waste their effort unsuccessfully examining the elements produced by the tool. An obvious remedy for this problem is to develop more effective methods, however, techniques can also get better at filtering irrelevant parts of code or estimating their confidence about the results.

- *Rationale and context.* As already mentioned, developers are usually not able to detect and understand the fault just by seeing a program element. An explicit rationale, which explains the reason for high suspiciousness, and useful context can help the user to succeed in this task. Moreover, this additional information allows the user to quickly identify the faulty element based on their intuition and experience, ignoring the particular order of elements given by the tool if preferred.

- *Integration.* There is a demanding need for developing practical tools that are integrated into the user's development workflow and toolchain (including various programming languages). An interesting fact is that the research would also benefit from such tools because they may serve as an ecosystem for experimentation and comparative studies. The aid lies mainly in the elimination of the need for considerable engineering effort of implementing an infrastructure that is common for various techniques regardless of the approach.

Motivated by the enthusiasm of developers and considering all the concerns and requirements, we develop a new tool for software fault localization focused on extensibility and user experience. We call it *Aardwolf*, named after an insectivorous (that is, eating insects, or bugs so to say) mammal, for obvious analogy.

Our explicit and one of the two most important goals is to have a highly modular design that enables simple extensibility. There are multiple points where one can extend Aardwolf, in particular, by implementing one of the following components:

- Programs that act as an interface between programming language, which user's software is written in, and the shared infrastructure provided by Aardwolf. We refer to these programs as frontends. Implementation of a new frontend brings support of the entire Aardwolf ecosystem for the programming language immediately without further work.

- Plugins that do the actual fault localization or an accompanied task (such as data pruning or results enhancement). It is possible to realize a new approach or algorithm inside the Aardwolf ecosystem utilizing provided infrastructure. There are two advantages to this approach. First, researchers can experiment with their ideas supported by the engineering efforts already made by others. Second, users may instantly benefit from state-of-the-art methods because they are developed in a practical tool they can use.

- Integrations of localization results with the programmer's toolchain. The spectrum of possibilities is broad. It can be a simple command-line output or integration with a popular editor or IDE, as well as with other tools such as continuous integration and development platforms like GitHub[2] or GitLab[3]. It enables bringing the support of fault localization closer to the user without the need of actually implementing a localization technique.

We strive for the design where all these individual components can be implemented as simply and conveniently as possible.

The second most important goal is to focus on recommendations given by authors of user studies, which are unfortunately not addressed in the majority of publications, although they are slowly gaining attention recently [9]. We provide mechanisms that are intended for dealing with the problem of trustworthiness and specifying the rationale and context to the user.

It is important to limit the scope of the tool to make its development feasible. We deal solely with logical software faults that cannot be recognized by static analysis. There are several tools that provide advanced program analysis to warn about potential errors in the source code without actually running the program. However, we target errors that can be observed only by verifying the program's results after running it on input data, and runtime information is used on top of static information to determine the likely cause. For input data and expected results, a test suite for the program is required to be available.

For the purposes of the thesis, we reviewed the literature and implemented three already existing techniques, varying quite greatly in how they approach the problem as well as needs from the infrastructure. The reason is mainly

[2]`https://github.com/`

[3]`https://about.gitlab.com/`

that the goal of this thesis is not to develop a new fault localization technique, but rather to develop an ecosystem. The use of existing methods, in which a lot of research effort has already been put, also allows us to compare our results with the published literature, that said, verifying our implementation.

We provide an implementation for two programming languages. C language is supported through our frontend for LLVM[4], a framework that is used by several widely used programming languages like C, C++, Rust, Swift, or Julia. An important reason for this choice was the existence of standard fault localization datasets for C. Python language features dynamic types, high-level syntactic constructs, and object-oriented paradigm. By support for Python we intend to demonstrate Aardwolf's flexibility.

Contributions

This thesis makes the following contributions.

- Summarizes various fault localization techniques and related methods and discusses their requirements and implications from an implementation perspective.
- Proposes a design of a new software fault localization tool. The focus has been placed on extensibility, language independence, and user-friendliness. These goals, especially the first two, are a significant improvement compared to existing solutions.
- Implements the proposed framework and offers three different fault localization methods, support for two programming languages, and informative terminal output.
- Experimentally evaluates the tool on a standard dataset in means of effectiveness, and on two larger software projects in means of ease of integration and scalability. It demonstrates that the accuracy is comparable with literature while being convenient for incorporation into a real-world project.

Organization

This thesis is organized into the following chapters: Chapter 1 gives an overview of three fault localization techniques of our choice and short description of tasks that are related to fault localization, accompanied by the necessary theoretical background. Chapter 2 proposes our architecture design of a new tool for SFL while discussing its components in detail. It also evaluates the design applicability on methods that we did not consider in Chapter 1. The

[4] `https://llvm.org/`

implementation of the tool is described in Chapter 3. Chapter 4 gives the experimental evaluation, where the localization effectiveness and scalability is measured as well as the integration of the tool with an existing real-world software project is discussed. Chapter 5 concludes the thesis and related work and also presents our plans for future development of Aardwolf.

Chapter **1**

Fault Localization

This chapter discusses the fault localization task, particularly focusing on selected techniques. In Section 1.1 we give an introductory overview of the problem and proposed solutions. Precise definitions of general terms and concepts along with their basic explanations are presented in Section 1.2. More detailed descriptions of selected fault localization techniques are given in Sections 1.3 to 1.5. We demonstrate their intuition and operation on a motivating example. Subsequent sections discuss problems related to the fault localization, such as heuristics for reducing the computational costs and the amount of false positives (Sections 1.6 and 1.7), combination of results into a stronger prediction (Section 1.8) and providing the user with some rich metadata (Section 1.9).

1.1 Overview

The essential task of a fault localization technique is to reason about the program given a failing execution or executions, usually by differentiating it with some successful ones. These executions are obtained from running a set of tests written by the developers. Test cases assert that the output values of the subject program are equal to those that are expected. In this way, failing and successful runs are distinguished from each other.

Algorithms for fault localization combine usually two sources of information: knowledge of program structure and the runtime behavior based on the test suite execution. Therefore it has the potential to reveal the causes of bugs of logic nature, that is, not discoverable by any static analysis[5].

Spectrum-based fault localization (SBFL) is the most studied, understood and evaluated FL technique in the literature [4, 10]. This and its implementation simplicity were the reason why we chose this approach as one we imple-

[5]We do not consider topics like model checking which require a formal specification of the program behavior.

mented for this thesis. It utilizes coverage statistics of certain program entities (e.g., statements) and its difference in failing and passing tests. Intuitively, an element covered more in failing executions than in passing ones is suspicious. The main degree of freedom in this family is what entities are inspected and how to combine given statistics into a single suspiciousness score.

The next method we chose adopts a probabilistic model with a graph structure, where the graph represents certain dependencies among the statements in the program. Roughly speaking, it builds a model where each node can enter into several possible states during the execution and it then estimates the probability of nodes being in a specific state given the states of its dependencies. If a statement happens to be in a very unusual state, it might be an explanation of the failure. We chose this method because it involves a graph structure and advanced properties like control and data dependencies.

The last fault localization approach of our choice is based on program invariants. Invariant is a property which holds for a given program point. For instance, it might hold that a variable is always in a specific range or two variables are in a certain relationship. If a failing execution violates an invariant, its location is considered as a strong predictor for the failure's root cause. We do not expect users to provide us with real, proper invariants, and thus these are inferred from the successful runs of the program.

Not only suspiciousness estimating algorithms on their own are useful for a fault localization tool. The process can be improved in several ways.

Test cases prioritization aims at reducing computational costs and runtime data noise by prioritizing those which are considered to have a higher value for the fault localization process. Such techniques are mostly based on test cases characteristics, either on their own or by comparing them with the rest.

Program elements can be prioritized as well for the very similar reasons. The approaches include considering the statement structure, statistics of its runtime behavior in failing and successful executions or their relationships and dependencies inside the program.

When we have predictions from multiple sources of different characteristics, we can combine them into stronger and more robust results by amplifying strengths and reducing weaknesses of individual methods. This is one big advantage of unification and generalization of the fault localization process into a single tool.

To help the user in the debugging task even more, on top of the suspicious elements estimation results, we can present some supplementary information. Such details can help to understand our output or guide the user when using the predictions.

1.1.1 Motivating Example

We now introduce a motivating example on which we demonstrate how later-described fault localization techniques work. It is somewhat artificial in order

```
1   def get_range(values):
2       min = max = values[0]
3
4       for x in values:
5           if x < min:
6               min = x
7           if x > max:
8               max = x
9
10      return min, max
11
12
13  def safe_div(nom, denom):
14      if denom == 0:
15          return float('inf')
16      else:
17          return nom / denom
18
19
20  def scale_minmax(values):
21      min, max = get_range(values)
22
23      if min == max:
24          min = 0
25
26      scaled = []
27      for x in values:
28          y = safe_div(x - min, max - min)
29          scaled.append(y)
30
31      return scaled
```

Figure 1.1: A correct implementation of motivating example.

to be simple enough but realistic and sufficient to illustrate all three methods at the same time.

Suppose that we develop a small utility function which normalizes values in a given array to be in the range from 0 to 1 using *minmax* scaling. An example of correct implementation is depicted in Figure 1.1. The program is written in Python language.

Our code is composed from three simple functions. The first one (lines 1–10) determines minimum and maximum values in the array. This is done simply by traversing the array and updating corresponding local variables.

Then we have a helper function (lines 13–17) which performs safe division of numbers. In Python, division by zero throws the `ZeroDivisionError`. Suppose that we do not want to throw exceptions from our code, therefore we check the denominator and return ∞ if it is zero. Note that IEEE-754 standard for floating point numbers contains a bit more complicated rules for

Table 1.1: The test suite for the motivating example.

Test	Values	Commentary
t_{pos}	$\langle 1, 3, 2\rangle$	Positive values only
t_{neg}	$\langle -1, -3, -2\rangle$	Negative values only
t_{mix}	$\langle -1, 3, 1\rangle$	Mixed sign
t_{eq}	$\langle 2, 2, 2\rangle$	Identical values

what the result of this ill operation should be, but that would unnecessarily complicate our example.

Finally, there is a function `scale_minmax` which performs the actual normalization. It firsts gets the minimum and maximum from the array. If they are equal, it means that the array contains only identical values. In that case, we set the value for minimum to be zero (line 24) with the effect that the result array will contain only ones. The actual normalization routine maps the values from the array using the standard formula (lines 28 and 29).

The test suite used for this motivating example is specified in Table 1.1.

1.2 Preliminaries

In this section, we present the basic terms and concepts used throughout the thesis with their precise definitions.

Definition 1.1. *Software error* is the deviance of an observed value or condition from the expected one. *Software fault* is an incorrect step, process or data definition causing a software error during a specific application. *Software failure* is an incorrect result caused by a software fault when it is activated and not corrected or "neutralized" by subsequent steps. [11, 12]

A showcase of these terms can be found in Figure 1.2. According to specification, `is_positive` function should return *true* if given value is included in the set of positive integers $\{1, 2, \ldots\}$. However, the predicate on line 3 is `value >= 0` instead of `value > 0` or `value >= 1`. Therefore, this predicate is faulty and it causes the failure in the test case on line 8 where the error is that it returns *true* instead of expected *false*.

A test suite can identify a failure but the searching for the fault is the task for the programmer or an automated tool. Synonyms for the term fault are "bug", "defect" and – with the context of a failure – "root cause".

To limit the scope of the thesis, we consider only deterministic failures. The determinism lies in the existence of an oracle which decides whether an output of a program execution is correct or not, and such decision is guaranteed to be consistent across multiple runs. For instance, this assumption discards programs with uncontrolled randomness or concurrency bugs.

```
1  # Program
2  def is_positive_int(value):
3      return isinstance(value, int) and value >= 0
4
5  # Test case
6  def test_is_positive_int_zero():
7      # Zero is not considered positive number
8      assert is_positive_int(0) == False
```

Figure 1.2: A showcase for the explanation of terms error, fault and failure. The fault is highlighted. It causes the error in the assertion on line 8 inducing the program failure.

Debugging is a two-step process beginning with the determination of the exact nature and location of the fault and concluding with fixing it [1]. The first step can be further divided into the *localization* and *comprehension* of the fault [7]. Automated tools can help in both phases, although the second one has not received as much attention in the literature as the first one [13].

Definition 1.2. *Test case* is a set of inputs, an executable piece of code exhibiting certain aspects of program behavior, and expected results developed for verifying compliance with the requirements. *Test suite* is a set of one or more test cases. [11] Executed test case has its *status*, which can be either *passing* (observed results conform to the expected) or *failing* (the opposite). Each test suite T can be thus partitioned as $T = T_p \cup T_f$, where T_p and T_f are the sets of passing and failing test cases, respectively, with $|T_p| = m_p$ and $|T_f| = m_f$.

An example of a test case can be seen in Figure 1.2 spanning on lines from 6 to 8. Test suite is a collection of such short testing routines.

It is a subject of research how types and characteristics of test suites affect the automated fault localization process [5, 14]. For instance, using unit tests with very narrow scope of what part of the program they exercise can already lead to successful localization performed manually very quickly. On the other hand, if system tests, which examine the global behavior of the program as a whole, using help of an automated tool might be more meaningful.

A program is composed from *statements* which can be clustered to groups of a certain level of granularity. The most related examples of such groups are *basic blocks* (sequence of statements with no branches except the last one) and *methods* (group of statements with an input performing a particular task). There are various analyses that can be done about programs, both static and dynamic. In the following text we describe some essential types of analysis and terms used in fault localization.

Speaking about individual statements, the following two terms are crucial for the data-flow analysis.

Definition 1.3. *Definition* of a variable v (usually abbreviated as *def*) is a statement which assigns a value to v. *Use* of a variable v is a reading of the value of v. [15]

In order to do something useful, the flow of control in programs can usually go into multiple branches at certain program points. To reason about this behavior, we need the following representation.

Definition 1.4. *Control flow graph* (CFG) depicts all possible control flow paths that may be performed during an execution. Nodes in the graph represent program statements and the directed edges represent control transfers between the statements. [11, 15]

Control flow graphs are usually constructed for standalone functions, and optionally connected with each other in a *call graph*.

Very useful information comes from program dependence analysis whose goal is to determine how statements influence each other, either in terms of data flow or control flow. Informal definitions for the program dependencies taken from [13] are presented here, while their rigorous versions can be found in [16].

Definition 1.5. In a control flow graph G, node x_1 is *control dependent* on node x_2 if x_2 has two outgoing edges e_1 and e_2 such that following conditions hold:

(1) every path in G, which starts with e_1 and ends in an exit node, contains x_1, and

(2) there exists a path starting with e_2 and ending in an exit node that does not contain x_1.

Definition 1.6. In a control flow graph G, node x_1 is *data dependent* on node x_2 if

(1) x_2 defines a variable v,

(2) there exists a path from x_2 to x_1 that does not redefine v, and

(3) x_1 uses v.

Together they form the following representation.

Definition 1.7. A *program dependence graph* (PDG) is a directed graph, whose nodes represent program statements and whose edges represent data- and control dependencies. Control dependence edges are labeled with the condition outcome of the predicate node. Data dependence edges are labeled with the name of the variables.

The accuracy of a PDG depends on the precision of the underlying analyses. For example, the precision of the pointer analysis affects the correctness of the data dependencies.

As the focus of this thesis is on dynamic fault localization techniques, the most fundamental data obtained from program execution are the following.

Definition 1.8. *Execution trace* is a sequence of statements encountered during an execution of the program. *Variable trace* is a record of the name and values of variables accessed and modified during an execution of the program. [11]

All these terms are general and commonly used in vast majority of methods. In the next sections, we describe the techniques we selected and present the terms that are mostly specific for them.

1.3 Spectrum Based FL

1.3.1 Introduction

A *program spectrum* characterizes a behavior of the program by tracking the execution of certain program entities for a specific test suite [17]. Examples of such entities are statements in general, branching statements, control flow paths or data dependencies. The run-time information that is gathered could be binary coverage status (whether the execution encountered the entity), the execution frequency (how many times it did), or similar [18]. The most commonly used program spectrum is binary coverage status of the statements [8, 18]. However, the study by [19] evaluated three types of spectra with the following observations:

- Different types of coverage spectra are better for different types of faults. No individual technique is superior to others.
- Overall, def-use pairs are more effective and stable than branches outcome, which are then more effective and stable than statements.
- Generally, although statement coverage is widely used, there seem to exist types of spectra which outperform it, however, usually with more demanding computational cost.

Let us use the following definition:

Definition 1.9. *Program (coverage) spectrum* is a quadruple of coefficients $\langle a^i_{ep}, a^i_{np}, a^i_{ef}, a^i_{nf} \rangle$ for every program entity e_i of a certain type (e.g., statements or def-use pairs). The coefficients represent, following a common notation, the number of passing (p) or failing (f) test cases that did (e) or did not (n) executed the entity e_i.

Let $c_t(e)$ be a function which returns 1 if test t covers e and 0 otherwise. Then the coefficients are computed as follows:

$$a^i_{ep} = \sum_{t \in T_p} c_t(e_i)\,, \qquad a^i_{np} = \sum_{t \in T_p} 1 - c_t(e_i)\,,$$
$$a^i_{ef} = \sum_{t \in T_f} c_t(e_i)\,, \qquad a^i_{nf} = \sum_{t \in T_f} 1 - c_t(e_i)\,.$$

Although several strategies of spectrum-based techniques exist [8], we will focus solely on *metric-based* techniques. These use a *ranking metric* that computes a suspiciousness score from the program coverage spectrum. Intuitively, if an element is executed more in the failing test cases and less in the passing ones, it is likely that this element is the faulty one. The metrics are constructed centrally based on this intuition.

Definition 1.10. *Ranking metric* is a function from a program spectrum $\langle a^i_{ep}, a^i_{np}, a^i_{ef}, a^i_{nf}\rangle$ to a real number representing the degree of suspiciousness of e_i being faulty.

Take for example the D* metric $\frac{a^2_{ef}}{a_{nf}+a_{ep}}$ [20]. Suppose that an entity e_i is executed in 5 out of 10 test cases, while three of these five are the only failing ones and the rest is passing. Therefore, its program spectrum is $a^i_{ep} = 2, a^i_{np} = 5, a^i_{ef} = 3, a^i_{nf} = 0$ and the suspiciousness score as computed by D* happens to be $\frac{3^2}{0+2} = 4.5$.

Plenty of ranking metrics have been introduced in the past research. There have been many experimental evaluations [21, 10] as well as theoretical studies [22, 23] of proposed metrics.

A selection of previously proposed formulas is presented in Table 1.2. Our choice is based on the compilations of surveys [3, 8], experimental evaluation results of [21] (their effectiveness on real faults) and theoretical findings of [24] (equivalence classes in the means of relative position in the ranked lists).

Notice that majority of formulas contain fractions where the denominator can be possibly zero. Such situation is not exceptional: take again D* metric as an example. The denominator is zero when all failing and none of passing test cases executed the statement, which is a completely legitimate case. A solution is to add a suitably small constant $\varepsilon > 0$ to the denominator.

1.3.2 Assumptions and Limitations

It is important to realize that, as described so far, spectrum-based techniques compute the correlation (or association) between program elements and test case results [8]. However, when an element correlates with program failure, it does not mean that it *causes* it [5]. This property of SBFL leads to a high

Table 1.2: Selected spectrum-based ranking metrics. The expressions use a coverage spectrum coefficients and compute the suspiciousness score. The star symbol in D* is actually a parameter to be set and can be any positive number (but is usually set to 2).

Name	Formula
D*	$\frac{a_{ef}^*}{a_{nf}+a_{ep}}$
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
O^p	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$
Overlap	$\frac{a_{ef}}{\min\{a_{ef},a_{nf},a_{ep}\}}$
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{nf}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$
Wong1	a_{ef}
Zoltar	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+\frac{10000\times a_{nf}\times a_{ep}}{a_{ef}}}$

rate of false positives (elements that are blamed to be faulty, but are actually correct). The main challenge is that fault localization is a causal problem [25] and coverage spectrum on its own does not involve any notion of causality.

SBFL techniques make several assumptions we need to be aware of [26, 27, 5]:

(1) If multiple elements are covered only by failing test cases and are not executed by any of the passing ones, no metric is able to differentiate them using the spectrum to identify the real causes. All elements including faulty ones thus need to be covered by both failing and passing test cases. This raises considerable requirements on the size and characteristics of the test suite. It is not rare that considerable number of program elements are not covered by passing tests.

(2) Program elements are assumed to be independent of each other. Let e_i be an element that causes the failure. If another element e_j is always executed with e_i, then e_j has the same suspiciousness score but is a false positive. This happens very often, mainly for statements from one basic block, or two distinct basic blocks whose execution depends on the same variables and predicates.

(3) Every failing test case should execute at least one faulty element whose execution causes the failure. From their nature, spectrum-based techniques assign a suspiciousness score to executed elements, therefore they cannot be successful if the bug involves a non-executable code (like a declaration) or missing code.

(4) A faulty element gets high suspiciousness only when it often leads to a failure. This assumption is based on the intuition under which the ranking metrics are developed. However, this is not necessarily the case. There might be faulty elements which are executed only under certain conditions. This leads to false negatives. Suppose that line 2 in the motivating example in Figure 1.1 is faulty such that it assigns wrong initial values to the variables. Although the statement is executed always, its faultiness would be revealed only if the value is not overwritten by subsequent assignments.

Item (2) implies that the minimal reasonable granularity is basic blocks because SBFL techniques cannot distinguish between statements in one basic block. Moreover, they assess the suspiciousness of individual program elements, but ignore the structural relationships in the program.

There are some other limitations of SBFL (although many of them apply on the other techniques as well, as described later). One is the concept of *coincidental correctness*, brought up in Item (4). There are three conditions for failure to be observed: the faulty element is reached, the program transitions into an infectious state, and the infection propagates to the output [5]. If the last one or the last two conditions are not met for a test case execution, then its assertions do not reveal the faulty behavior in the program and so it results to a passing test. As can be seen in Table 1.2, the suspiciousness of an element in the majority of ranking metrics (sensibly) decreases with growing number of passing test cases that execute that element. However, if there is a faulty element that is often executed in coincidentally correct tests, its suspiciousness is also reduced.

Obviously, SBFL techniques cannot directly locate faults that are of non-executable nature, either missing code or declarations (such as custom data types). Code omission is generally challenge for majority of fault localization techniques [3], since they usually process the execution information but the faulty code is not even in the program in this case. Although this type of fault cannot be directly located by these techniques, such bugs may cause some anomalous effects in the program state which – if discovered by the technique – can hint the user and help them to identify the real cause [3].

1.3.3 Motivating Example

Before explaining the technique on a concrete example, let us first introduce a bug into our program in Figure 1.1. The fault will be wrong target variable

```
 def get_range(values):
     min = max = values[0]

     for x in values:
         if x < min:
             min = x
         if x > max:
-            max = x
+            min = x

     return min, max
```

Figure 1.3: Diff between the correct implementation and the first faulty version.

	Coverage				Spectrum				Suspiciousness		
Line	t_{pos}	t_{neg}	t_{mix}	t_{eq}	a_{ep}	a_{np}	a_{ef}	a_{nf}	D^2	Ochiai	Tarantula
2	•	•	•	•	2	0	2	0	1.6	0.69	0.38
4	•	•	•	•	2	0	2	0	1.6	0.69	0.38
5	•	•	•	•	2	0	2	0	1.6	0.69	0.38
6	•	•	•		1	1	2	0	2.67	0.78	0.47
7	•	•	•	•	2	0	2	0	1.6	0.69	0.38
▷ 8	•		•		0	2	2	0	**8**	**0.94**	**0.62**
10	•	•	•	•	2	0	2	0	1.6	0.69	0.38
	F	P	F	P							

Figure 1.4: Partial results for a subset of ranking metrics. The triangle denotes the faulty line. The safe constant ε in all denominators was set to 0.5.

of the assignment on line 8. This particular bug could be introduced by copy&pasting the line 6 and forgetting to rename the variable. Its nature belongs into one of the five most common bug-fix patterns as studied by [28] identified as *change of assignment expression*. The change to our code is illustrated in Figure 1.3.

Recall the test suite presented in Table 1.1. With the introduced fault, two tests are now failing, namely t_{pos} and t_{mix}. In the other test cases, the value of maximum is never to be updated. The faulty line is encountered by both failing executions and not encountered in the passing ones. Therefore it gets high suspiciousness values from ranking metrics and is indeed determined as the most suspicious statement by spectrum-based method. The situation is illustrated in Figure 1.4.

1.4 Probabilistic Graphical Model Based FL

1.4.1 Introduction

Probabilistic graphical model based fault localization is a family of techniques that adopt a probabilistic graphical model and its application on program dependence graph [29]. The difference between each technique is mainly the choice of the model, what program relationships are represented by the edges (e.g., control or data dependencies), and the computation of the suspiciousness score. Examples of such works are [13, 29, 30].

In this thesis, we describe, to some extent generalized, approach introduced in [13]. We see other similar approaches found in the literature as extensions [29], simplifications [30] or modifications of this method. We will use the term *probabilistic dependence* for referring to our implementation of this approach in this thesis.

The basis of the method is the program dependence graph defined in Definition 1.7, which models control and data dependencies between statements in the program – a very important information about the program structure. The technique first transforms PDG in a specific way, then associates a set of abstract, theoretically possible states with each node, and estimates conditional probabilities between dependent nodes with associated concrete states encountered during program execution on the test suite.

The underlying probabilistic graphical model is dependency network – a possibly cyclic directed graph modelling dependencies among random variables. It was chosen because it permits directed cycles which are common in PDGs of typical programs because of loops. In our case, the random variables are the states of the program statements, between which the control and data dependencies are modelled. The motivation for this approach is that it can differentiate between failing and passing runs not only statistically, but also structurally.

This technique first takes the program dependence graph and transforms it structurally to comply with the dependence network formalism. We refer to the original work [13] for the detailed description of this transformation. Then, each node X_j is assigned a set of discrete abstract states $\{x_{j_1}, \ldots, x_{j_k}\}$. Generally speaking, a state represents what its predicate outcome is (if any), or which parents assigned the current value of variables used by the node. The states must be mutually exclusive (i.e., a node cannot be in two different states at the same time).

1.4.2 State Specification

The technique distinguishes between two types of nodes: *predicate* and *non-predicate*. Note that when a statement is conditional and at the same time uses some variable (which is very common), it must be split into two distinct nodes during the PDG transformation, while one becomes of predicate type and the other of non-predicate type.

The states of predicate nodes are modelled using the predicate outcome. In their work, [13] transform all predicates into simple ones (i.e., a relation between two values only). Moreover, they distinguish between primitive variables and pointers (or references), and regardless of the actual relational operator in the predicate, they track every possible relation between the compared values during execution. For instance, even if the operator in a predicate is `<=`, they record whether the outcome is $<$, $=$, or $>$.

We consider these modifications to be a considerable work for frontend developers, thus we omit them in our explanation and deem as unnecessary. The influence of this aspect is unknown, even though we suppose that more detailed information could possibly lead to more accurate results. Note that despite that we do not expect these modifications explicitly in the technique itself, it can be simulated by the frontend by performing relevant transformations.

Existing works involving predicates like [13, 30] implicitly assume only branching statements whose conditional expression evaluates either to true, or false. However, we also include more complex predicates (like `switch` syntax construct), again, for not complicating the development of a frontend. This is a legitimate generalization since we only need to distinguish between possible outcome states of the predicate nodes.

To characterize abstract states of non-predicate nodes, we use several concepts. A *data environment* of a statement e_i is the set of all statements which e_i is data-dependent on. To illustrate, consider Figure 1.1 and let us $d_e(v)$ denote a definition of variable v by statement e (here identified by its line in the source code). Then the data environment of the condition on line 7 is $\{d_2(max), d_8(max), d_4(x)\}$. As can be seen, the value of variable `max` can be defined by two different statements.

An *elementary data context* of a statement e_i is the set of statements that assigned last to variables used by e_i at a certain execution point. Elementary data context is generally a subset of data environment. Consider the same example and suppose that, while running the program, we are in the first iteration of the loop on lines 4–8. That is, the assignment on line 8 was not executed, so the value of `max` comes from line 2. Then the elementary data context of the condition on line 7 is $\{d_2(max), d_4(x)\}$.

The set of all elementary data contexts of a statement encountered during the entire execution is called *data context*. An elementary data context therefore represents a possible state associated with a non-predicate node, while the data context is the set of all concrete states the node happened to be during the execution.

1.4.3 Learning

Model proposed in [13] is *probabilistic program dependence graph* (PPDG). It uses the dependency network created by transforming the program dependence graph, and estimates the conditional probabilities from dynamic data collected from the execution of the program on the test suite.

A special kind of trace, called *node-state trace*, is used for parameters estimation. It is a sequence of executed nodes from the PPDG along with their states active at that particular point in the execution. For example, when line 7 in Figure 1.1 is encountered during an execution, it is represented by two nodes in the trace. One must be associated with either the state $\{d_2(max), d_4(x)\}$ or $\{d_8(max), d_4(x)\}$ (non-predicate node) and the other with either the state *true* or *false* (predicate node). A node can appear multiple times in such trace, possibly with different states. The algorithm for translating ordinary execution trace into the node-state trace is rather simple. It uses a mapping from variables to their last definition statements to determine states of non-predicate nodes, and checks the successor of each conditional statement to determine its state as predicate node.

Learning the parameters consists of estimating conditional probability distributions of node being in a specific state given the states of its parents in PPDG. Let us denote the set of parents (immediate predecessors) of a node X as $Pa(X)$, and define it as

$$Pa(X) = \{X' \mid (X' \to X) \in E(G)\},$$

where $(X' \to X)$ stands for a directed edge from X' to X. It is important to remember that $Pa(X)$ represents the set of the predecessors in the (transformed) program dependence graph, and not in the control flow graph. The state associated with node X_j at point i is denoted as $x_j^{(i)}$ and the state configuration of its parents as $pa_j^{(i)}$.

For a node with no parents, the technique estimates the probability of the node being in a given state as

$$p(X_j = x_j^{(i)}) = \frac{n(X_j = x_j^{(i)})}{n(X_j)}, \tag{1.1}$$

where $n(X_j = x_j^{(i)})$ is the number of times node X_j is in the state $x_j^{(i)}$, and $n(X_j)$ the number of times node X_j occurs in given node-state traces. It essentially computes a probability of the node being in a specific state.

For a node that does have parents, the technique estimates the conditional probability of the node being in a state given the states of its parents as

$$p(X_j = x_j^{(i)} \mid Pa(X_j) = pa_j^{(i)}) = \frac{n(X_j = x_j^{(i)}, Pa(X_j) = pa_j^{(i)})}{n(Pa(X_j) = pa_j^{(i)})}, \tag{1.2}$$

where $n(X_j = x_j^{(i)}, Pa(X_j) = pa_j^{(i)})$ is the number of times node X_j and its parents are in this specific state configuration in given node-state traces.

We use the same algorithm for estimating the parameters of PPDG as presented in [13]. It traverses each node-state trace from the input set from beginning to end, and updates the state counters along the way. The values of counters are then used for computing the probabilities using Equations (1.1) and (1.2). Finally, learned PPDG is returned.

1.4.4 Fault Localization

For fault localization, the technique first learns a PPDG on node-state traces of all *passing* tests. This step estimates the expected probabilities when the result of the execution is correct. Then it analyses a *single failing* test case and ranks the nodes in particular state configurations in ascending order by the lowest conditional probability obtained from PPDG. The rationale is that when a probability of a node being in a state given the states of its parents is low, then the node entered some unusual state in the execution, and therefore is suspicious to be the cause of the failure. When multiple nodes have the same lowest probability, the tie is broken such that suspicious states that ocurred earlier in the execution are prioritized.

The state configuration associated with the suspicious point in the execution can be used to explain to the user why the technique blames that particular statement. This information is further extended by the algorithm for fault comprehension [13], where expected state configurations of each node are determined. An expected configuration is that with the highest likelihood as estimated by PPDG since it offers the best explanation of what the expected behavior of the node and its parents is.

1.4.5 Assumptions and Limitations

As many fault localization techniques, this approach heavily relies on the characteristics of the test suite. The probability estimates of PPDG are learned on the passing test cases and therefore reflect the probability distribution in these particular runs. Moreover, they also depend on the sample sizes (the number of states and their count of occurrence) used to compute them – if the sample size for a particular node is too small, the estimated probabilities cannot be accurate. This limitation could be addressed by learning PPDG from running the program in the field. In such case, the probability estimates would reflect actual program behavior. [13]

There are also several limitations that are specific for this technique:

(1) Because the program dependence graph is transformed in order to reduce the number of possible node states (by splitting predicate and non-predicate nodes), and to comply with the dependence network formalism (removing self-loops), computed statistical dependencies between nodes

may not exactly correspond to the dependencies in PDG. This might affect the application of the approach, especially when a more complex inference algorithm is used [13].

(2) Since the technique is based on the dependency network, it does not support reasoning across nonadjacent nodes [29]. In other words, it only discovers local anomalies. Similarly to traditional SBFL approach, presented algorithm finds an element that is most associated with the failure, but does not necessarily cause it [25]. However, taking program structure and program states into account gives this technique a considerable advantage over spectrum-based family.

(3) It is also difficult to scale the approach to larger software. The first reason is that the number of possible conditional probabilities of a node can be very large, depending on the number of predecessors in PDG. The second reason is that precise computation of data dependencies is expensive [30].

If the issue mentioned in Item (2) was addressed by incorporating causality, one would need to convert the dependency network to the causal graph – a representation lacking cycles. However, elimination of cycles may result in loss of dependence information, likely affecting the effectiveness of the localization [25]. Yet, such an approach is proposed in [29], where the Bayesian network is used as acyclic graphical model, and the suspiciousness is computed by taking erroneous output nodes as conditions in the conditional probability calculations for each non-output node.

Regarding the scalability mentioned in Item (3), our simplification of state specification of predicate nodes might result in a performance improvement. The computation overhead can be also controlled by the precision of underlying data dependence analysis. Obviously, less precise algorithms can be much faster, but presumably affect negatively the localization effectiveness. To avoid the issue with expensive data dependence analysis, authors of [30] use only control dependence information.

Handling of missing code is also problematic for this technique. However, indicating an unusual and more likely control- or data-flow path has potential to help the user to identify the fault of omission. To our knowledge, there is no study about coincidental correctness effects on probabilistic dependence family, but, intuitively, its model learning using passing test cases is affected by this aspect.

1.4.6 Motivating Example

Again, we introduce a bug into our motivating example program (Figure 1.1). It will be wrong initial value of `max` variable in the `get_range` function. Variable for minimum will be correctly initialized to $+\infty$, while variable for max-

```
1    def get_range(values):
-        min = max = values[0]
+        min, max = float('inf'), 0

         for x in values:
             if x < min:
                 min = x
             if x > max:
                 max = x

         return min, max
```

Figure 1.5: Diff between the correct implementation and the second faulty version.

imum to zero. This is wrong because no value from negative-only arrays exceeds this initial value and thus the maximum will not be properly determined. Thus the test case t_{neg} from Table 1.1 fails. The change to our code is illustrated in Figure 1.5.

Note that the faulty line is in this case always executed in all test cases. Therefore, using spectrum-based method would not help at all since it cannot differentiate the statement using the coverage spectrum. Moreover, in this particular example, 23 out of 25 statements get the same, highest suspiciousness score. This brings no information to the user.

Using probabilistic dependence method, Aardwolf pinpoints two statements which get significantly higher suspiciousness than the other statements. These are lines 5 and 7. Although they are not the actual faults, we get additional information from the analysis. The interesting case for us is the prediction of line 7. Here is the output of the algorithm:

- The predicate on line 7 was always false in the failing execution. Thus the control flow always continued without setting the maximum variable. This state is considered unlikely since the predicate was true at least once in the passing executions.
- By inspecting all possible states which the statement happened to be, the output mentions that it was expected to observe the predicate being satisfied.
- The list of statements which assigned last to variables used by the predicate at this unlikely state configuration is presented as well. In this case, the value of `max` variable comes from line 2, emphasizing that the assignment on line 8 is expected.

Despite the fact that it does not pinpoint the actual fault, from the rich information it provides with the prediction, we could possibly deduce what is the root cause. The technique was able to locate the approximate faulty segment and differentiate it from the other parts of the program. In this case, it was therefore much more successful than spectrum-based technique.

1.5 Likely Invariants Based FL

1.5.1 Introduction

Program invariants are properties in form of predicates at certain program points that are guaranteed to hold for all possible inputs. Predicates observed to hold for some, but perhaps not all program inputs are known as *likely invariants* [31], or potential invariants [32]. The first idea of using likely invariants for fault localization was presented in [33], followed by [32, 34] for example.

Likely invariants can be extracted by monitoring the execution of the program and summarize the properties that held during such execution. This distinguishes dynamic invariant detection from static one, where an inferred property is guaranteed to hold for all possible inputs. To satisfy this claim, static inference must be conservative and therefore has limited scope. On the other way, dynamic approach can detect properties that would be hard or impossible to prove statically. An inherent limitation is that inferred invariants might not be valid. This is highly constrained by the quality of the test suite [33].

In context of fault localization, a technique utilizing likely invariants can infer them during the executions of passing test cases. Since the program state at most[6] points in these executions can be assumed to be correct, detected program invariants has the potential to be real constraints. Every violation of these invariants when executed with failing test case indicates difference from the passing executions and possible faulty program state. Therefore, locations of such violations are candidate root causes of the failure.

The invariants are inferred from tracking the result values of selected expressions at various program points [33]. In *learning phase*, the detector maintains an invariant hypothesis that is satisfied by all the values that have occurred during the execution so far. At the beginning, this hypothesis is the most strict (e.g., variable is constantly equal to the value encountered first). When a new value does not conform to such hypothesis, it is relaxed to allow this new value or falsified completely. In *checking phase*, when a new value does not conform to the hypothesis proposed by the learning phase, it is considered as a violation and reported to the user. Learning phase is applied on the passing test cases, whereas checking phase is performed on the failing test cases.

[6]But not all, due to coincidental correctness.

```
1  def scale_minmax(values):
2      lo = min(values)
3      hi = max(values)
4
5      scaled = []
6      for x in values:
7          y = (x - lo) / (hi - lo)
8          scaled.append(y)
9
10     return scaled
```

(a)

Values	Invariants
$\langle 1, 3, 2 \rangle$	$lo = 1, hi = 3, 0 \leq y \leq 1$
$\langle -1, -3, -2 \rangle$	$-3 \leq lo \leq 1, -1 \leq hi \leq 3,$ $0 \leq y \leq 1$
$\langle -1, 3, 1 \rangle$	$-3 \leq lo \leq 1, -1 \leq hi \leq 3,$ $0 \leq y \leq 1$
$\langle 1, 1, 1 \rangle$	$-3 \leq lo \leq 1, -1 \leq hi \leq 3,$ $0 \leq y \leq 1 \quad (y = \infty)$

(b)

Figure 1.6: A sample program code (a) and an example of gradual relaxation of invariants during passing runs (1–3) and learned invariants violation (in red) in failing runs (4) (b).

Consider an example code depicted in Figure 1.6a which scales values in given array to be in $[0, 1]$ range. It is essentially the same function as in our motivating example, only simplified by removing `get_range` and `safe_div` functions, and not handling the situation when all values are equal. When this situation happens, `lo` and `hi` variables are also equal, and thus a division by zero occurs on line 7. Let us pretend that built-in division follows IEEE-754 standard by returning NaN or infinity instead of throwing `ZeroDivisionError` exception. Suppose that we track the invariant that a variable is in a range. The program point where it is checked is assignment into the variable.

Each line in the top segment of Figure 1.6b shows the set of invariants that still hold after running the test case and all its predecessors. Gradual relaxation of the invariant hypothesis is clearly observable on `lo` and `hi` variables.

After learning the set of invariants that hold in first three passing executions, the program is executed on the fourth, failing test. Since we suppose that the result of division by zero on line 7 is either positive infinity, it does violate the invariant that variable `y` is in range $[0, 1]$. Therefore, the statement is a candidate to be the root cause of the erroneous behavior.

1.5.2 Invariants

There is plenty of possibilities which invariants can be considered in a detection tool. In order to have the inference tractable, the technique must use a fixed set of invariant schemas. For example, [33] checks several bitwise-oriented properties such as being a constant, only positive/negative, having approximate upper bound or being null; [32] check for equality, sum and less-than relationships between pair of variables, and constant equality; and [34] use only range invariant.

We selected four invariant schemas based on our intuition about how they might help in fault localization task. Required implementation effort was also considered. Our solution checks the following invariants:

- *Constant.* Variable x is a constant value during the whole execution.
- *Range.* Variable x is in a range $a \leq x$, $x \leq b$ or $a \leq x \leq b$, where a, b are some constants.
- *Type stability.* Variable x is of the same data type throughout the execution. Violation of this invariant is possible in dynamically typed or polymorphic languages.
- *Non-exceptional value.* Variable x has never been assigned an exceptional value. We consider NaN, positive and negative infinity for floating point numbers, and null for reference types to be exceptional values.

The first two invariants were used in [33, 32] and [34], respectively. If a variable was a constant value in all passing executions but is suddenly a different value in a failing case, such discrepancy can point to a considerable difference between passing and failing tests. Similar applies to range invariant which can be seen as a relaxation of constant invariant.

The other two were partially used in [33] for reference types (that is, change of run-time type of a polymorphic object, and null value, respectively). Type stability constraint makes more sense in languages with dynamic typing or polymorphic capabilities, and since our aim is to support such languages, we incorporated this invariant into our tool. In case of non-exceptional value, we also track occurrences of NaN and both infinities because these often arise from invalid arithmetic operations, possibly indicating suspicious behavior.

Theoretically, program invariants can be inferred for any data type in a program. For implementation simplification and practical reasons, so far we support only primitive data types: booleans, integers of various width and both signs, single and double precision floating point numbers and references (identified by the name of their runtime type or whether they are null). Similar scope of supported data types is used in [33], excluding floating points numbers due to the invariant representation they used, but the presence of complications with this data type is also mentioned in [31]. Note that in [34] authors are able to work on arrays and even some pointer-based data structures containing these primitive types, thanks to the advanced dynamic invariants detection tool presented in [31].

Although invariants can be checked at function entry and exit nodes [32, 31], for more specific information, reads and writes from/into object-like variables (including arrays) [33, 34], reads and writes from/into static variables [33], results at function call sites [33], and function return statements [34] are more preferred choices.

In both works ([33, 34]), stack-allocated local variables are ignored for being time-consuming to track and less interesting than object-like and static variables and function call/return statements, which both better capture global state of program execution. Nevertheless, we decided to include stack-allocated variables to have more detailed information.

Note that function call sites and function return statements (i.e., results of callees) carry almost identical information. The main difference is when a non-instrumented component takes place in the execution. In case of instrumenting function return statements, the output of calling foreign interface, which is a valuable information, is lost. Therefore we believe that instrumenting function call sites is superior choice.

To summarize, we put invariants checks at these locations:

- Writes into any variable. We do not track reads from variables used in expressions because we think that, in context of fault localization, violated invariant on write to a variable indicates a faulty statement that computed such value, while violation on variable read just reveal that fault might have occurred in the past.
- Results of function calls. Such failed invariant can indicate two causes of a fault, either the fault is contained in the function itself, or the function was called with wrong arguments.
- Function arguments. Complementary to the previous point, checking function arguments can early reveal a wrong use of a function.

So far, invariants were applied on values integral to the program. In [35], the notion of program invariants is extended also for execution features such as basic block counts or number of function calls. The authors argue that use of these *extended invariants* can be useful for fault localization.

1.5.3 Assumptions and Limitations

The first concern for a variable values analysis is its scalability, since a program execution can generate huge amount of data to be processed. The computational and memory costs of the invariants detection grows with the number of instrumented program points and variables, number of invariants checked, and the size of runtime traces generated by executing the test suite [31]. There is a trade-off between richness of the recorded information and processing costs. For example, one could dump entire arrays to the output in order to be able to analyze its contents, however, such analysis might become practically unfeasible.

The tool presented in [33] is designed to monitor the invariants online while running the program, and uses a very compact and efficient representation. However, capabilities of the representation are very limited. We implement an “offline” approach, where we record all values in a trace file which is then processed, similarly as Daikon tool [31] does.

The fundamental assumption of invariant-based technique is that useful invariants can be inferred from the passing executions and will be violated in the failing executions. The two key observations of [34] are:

- Using training inputs which are close to the failing input is more effective than using invariants detected using a large number of unrelated inputs in a traditional test suite.

- Sophisticated filtering allows to start with a large set of initial candidate causes (reducing false negatives). Such filtering then narrows down the initial set to a small set of final predictions (reducing false positives).

The size and characteristics of the test suite highly influences how sensible and reliable invariants are obtained. An inadequate tests may result in false or uninteresting invariants or very few of them. Based on mentioned observations, utilization of automatic or semi-automatic methods for tests generation (such as property-based or grammar-based, with automatic memory-error checkers or programmer-inserted assertions) can lead to effective likely invariants inference. Grammar-based approach was evaluated in [31] for dynamic invariant detection and successfully used in [34] aiding their invariant-based fault localization technique.

As the majority of failed invariants do not correspond to actual root causes, advanced algorithms for discarding the unrelated ones is crucial in order to have a usable technique. This introduces a difficult challenge. Moreover, there is no direct way how to obtain suspiciousness scores for the statements that are involved in the failed invariants, as it is just a binary indication whether it was violated or not. One could use some measure of confidence, but that might not correlate with the likeliness of the statement being the root cause.

An interesting application of invariant-based localization tool presented in [33] is to apply it on long-running programs. Assuming that the program does not enter an infected state, caused by the faulty element, in the early part, the invariants might be inferred from running the program in the production. Later, it can be switched to the checking phase where invariants are not relaxed but checked for violations. The success on this type of programs suggests that closeness of the “good” inputs to the failure-inducing inputs is beneficial.

1.5.4 Motivating Example

We make a third faulty variant of our program by omitting the check for `min` and `max` equality on line 23 (Figure 1.1). This type of bug is also one of the most common bug-fix patterns [28] when *a precondition check is added.* The fault involves missing code which causes issues for traditional fault localization

```
20  def scale_minmax(values):
        min, max = get_range(values)

-       if min == max:
-           min = 0
-
        scaled = []
        for x in values:
            y = safe_div(x - min, max - min)
            scaled.append(y)

        return scaled
```

Figure 1.7: Diff between the correct implementation and the third faulty version.

techniques. When it is run on our test suite, the test t_{eq} becomes failing. The change to our code is illustrated in Figure 1.7.

For this case, both spectrum-based and probabilistic dependence techniques identify line 15, which returns ∞ when zero denominator occurs, as the root cause. Although this line indeed produces a value which we do not expect in the output, in this context, the implementation of function `safe_div` is correct. Thus blaming its content is a false track.

In this particular case and given the current implementation of Aardwolf, the result of likely invariants analysis is as follows:

- The argument `denom` on line 13 was expected to be in range $[2, 4]$, but was 0 in the failing execution.
- The result of the `safe_div` call on line 28 was expected to be in range $[0, 1]$, but was ∞ in the failing execution.
- The assignment value on the same line was expected to be in range $[0, 1]$ as well, but was ∞ in the failing execution.

From the output, we can clearly deduce that the root cause of the error is that the denominator happens to be 0, in other words, `min` and `max` variables are equal, which has the effect that the scaled variable gets infinity value. Such information leads to a correct fix (which might be either ours presented in Figure 1.1 or some other).

Again, despite that it did not provide the user with the true faulty line, which is in this case actually impossible due to ambiguity of the fix, it produced very valuable details which informed the user what is the root cause, so they can implement a fix of their choice.

1.6 Test Cases Prioritization

The characteristics and composition of the test suite considerably affect the performance of fault localization techniques. The test suite reduction (or analogously test cases prioritization), as one of the aspects of the test suite composition, was studied in several works [36, 37, 38, 39] (see [3] for many others).

The impact of the test suite reduction was also evaluated by [14] with the following key observations. First, reduction in the size of a test suite generally harms the effectiveness of the fault localization techniques, but thorough and less severe approach causes negligible impact on the effectiveness, while reducing the computational costs. Second, removing test case redundancy (e.g., in the means of code coverage resemblance) occasionally improves the fault localization.

Test cases prioritization has at least potential to lower the costs of computationally expensive localization techniques without sacrificing their effectiveness too much. Intuitively, not all test cases have the same positive impact on the localization performance. Therefore, while selecting a feasible subset of test cases, the technique should use those that contribute the most to differentiating the likeliness of being faulty for program elements.

In most cases, relevant test case is selected based on some notion of *proximity* to failing test cases. Being close to the failure-inducing execution, such passing test case has potential to narrow down the set of suspicious program elements.

In [36], a difference metric that involves the sequence of statements and their control dependencies, while differentiating branch instances with similar contexts but different outcomes, is used. Both [37, 39] apply some notion of dividing the domain into categories and measuring the degree of difference. Contrary to others, [39] assume that test cases with diverse coverage are of high value for the fault localization. In [38], an information-theoretic approach of closeness is proposed.

We consider the test cases prioritization task as finding a priority function $\pi(t)$ mapping a test case t to a priority $r_t \in \mathbb{R}$. Generally, the priorities are assigned to passing and failing test cases separately. For passing test cases, the priority should indicate the degree of potential to discriminate suspicious elements while contrasting them with the failing test cases. For failing test cases, the priority should estimate the difficulty of faulty elements localization and easier instances should be prioritized.

We do not implement any test cases prioritization technique in this thesis but we provide the API which can be used for it. In particular, all test cases have by default equal priority, but a technique can assign specific values to certain tests. If set to zero, it is considered as completely irrelevant. Localization plugins can then obtain the test cases sorted by this priority.

1.7 Program Elements Prioritization

Similarly to test cases, not every program element has the same impact on suspiciousness estimation. Authors of [40] argue that, in the computation of suspiciousness of an element, the contribution of the failing test cases which cover it should be greater than that of passing test cases. The rationale behind this intuition is that the execution of failing test cases carry the information that they certainly cover at least one faulty statement, whereas execution of passing test cases are not guaranteed to be absolutely free of any faulty statement [41]. This is the implication of coincidental correctness [3].

Similarly to the previous section, we consider the prioritization task as finding a priority function $\pi(e)$ which maps an entity e to a priority $r_e \in \mathbb{R}$. The higher the element has the priority, the closer to the top of the ranked list it should be. The goal of such mapping is to rank the elements such that if there is a set of elements, containing a faulty one e_f, that are assigned the same suspiciousness score by a fault localization technique, e_f is ranked closest to the top among them. If no prioritization is used, the elements are ranked arbitrarily.

Although a program elements prioritization technique can be usually performed before the actual fault localization (such as [41, 42]), there are strategies that can assign rank elements that happen to be in a suspiciousness tie [40], and thus need to be activated after the scores are computed. We therefore allow to prioritize program elements both before and after the localization. However, we do not actually implement any program elements prioritization technique for this thesis.

1.8 Combining the Results

The effectiveness of a fault localization technique is heavily dependent on characteristics of the test suite, program structure and semantics, nature of the bug, etc. [3] As shown by [43], even various ranking metrics from spectrum-based fault localization domain can give different results for different categories of bugs. Presumably, such diversity is even more substantial between techniques with very different bases, as those presented in this thesis. It makes therefore sense to combine the results from all these different sources to retain beneficial aspects of individual approaches while mitigating their drawbacks.

Several works [44, 45, 46, 47] indeed show superior performance over individual techniques. Both [46, 47] use learning-to-rank machine learning model to incorporate results from multiple sources. In [44], search-based optimization techniques, namely genetic algorithm and simulated annealing, are used to find optimal combinations of individual techniques.

All these approaches require a training phase on labeled data. However, such data are often unavailable or may not be representative enough for new buggy programs, which could affect the effectiveness. Contrary to mentioned techniques, [45] proposes an approach which requires no training data and is therefore highly flexible and extensible. In their work, authors applied this approach on multiple spectrum-based ranking metrics, but also declare that it is possible to use it with different fault localization techniques as well. The only requirement is that such technique assigns some numeric suspiciousness score to program elements.

In Aardwolf, this functionality can be implemented as a post-processing plugin, which is given a collection of results computed in the previous phase. These can be then used for building a new list of results.

1.9 Supplementary Information

The survey [6] demonstrates that information richer than just ordered list is very important for the potential users of a fault localization tool. As mentioned in the introduction, the most crucial is providing the rationale for the prediction. Some other types of supplementary information are discussed in [7]. Here are some examples:

- Based on the responses in [6], developers will likely distrust the tool if it gives too much false positives. This issue was addressed by [48] by building a machine learning classification model which attempts to decide whether the localization output should be trusted or not. If not, the developer can skip the automated localization not to waste their time by examining its output, and switch to conventional debugging instead. The model is based on features extracted from the program, execution traces and output results. The effectivity criterion for learning the classifier is whether the faulty element appears in the top 10 list. All features used in the technique can be conveniently obtained from Aardwolf's data structures.

- The results can be clustered by aggregating them by functions or files. Such summarization will not help to pinpoint the exact location of the fault, but can suggest promising starting places, especially to developers which are not very familiar with the codebase.

- Displaying the values of variables in failing executions is very helpful and used in conventional (manual) debugging as it helps the developer to identify suspicious values. Since Aardwolf is able to gather the variable trace, at least in limited fashion, it could include the variable values in the output.

- Even though the purpose of Aardwolf is fault localization, it can provide the user with details describing the quality of the test suite. The most essential quality metric is code coverage (for example, statement coverage or predicate outcome coverage) [1]. This can be easily computed from data available in Aardwolf.

Chapter 2

Design

This chapter describes the architectural design of Aardwolf. It thoroughly characterizes individual components and specifies how these components interact with each other.

The design goals are summarized in the following list:

- *Extensibility.* It should be possible and convenient to implement a new fault localization technique or alike into Aardwolf. We would like it to be able to serve also as a research ecosystem in which new approaches are easily developed and evaluated.

- *Language independence.* The goal is to offer all localization functionality implemented in Aardwolf to multiple programming languages. Building a frontend (that is, bringing support of Aardwolf for a language) should take a reasonably small engineering effort.

- *User experience.* We want to create a tool actually usable by developers. This does not include only accuracy, but also results presentation with context and explanations, scalability, and convenient integration with existing projects.

- *Generality.* This relates to the first point in this list. Although we implement just a small fraction of existing fault localization techniques (although they are fairly different from each other), the design should be general enough to be able to support majority of them. We discuss this topic at the end of this chapter.

The overall diagram of all components and related artifacts, from high level perspective, is depicted in Figure 2.1. The source files of user's program are first processed by a frontend with support for the programming language which the source code is written in. Frontend performs static analysis, which is saved for later use, and program instrumentation. An executable is built from the instrumented source and tests, and is run to produce runtime data. All

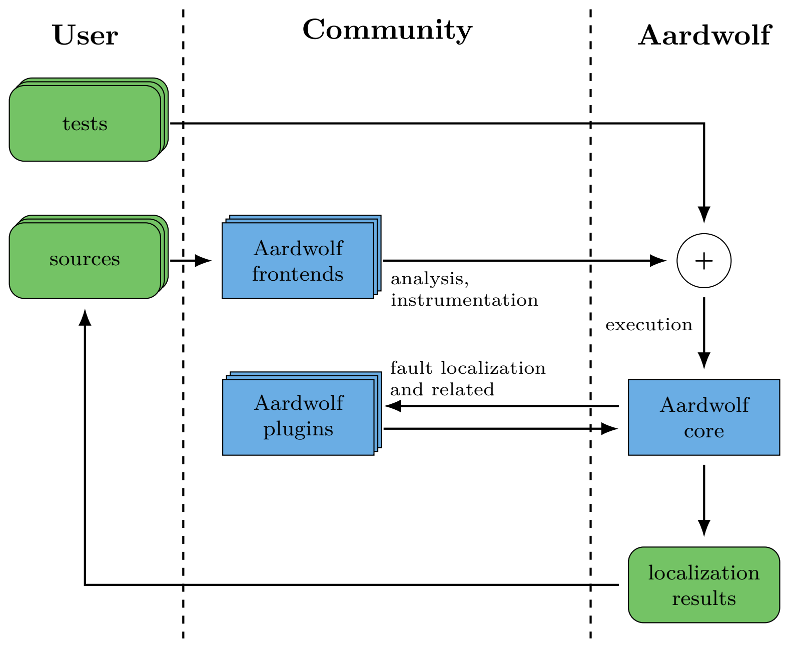


Figure 2.1: Architecture of Aardwolf from high level perspective. Blue rectangles correspond to the components of the system, whereas green rounded ones represent artifacts consumed or produced by the components. Components that generally have multiple instances are visually distinguished.

generated data is consumed by the core and, via high-level API, provided to plugins that actually perform the fault localization or a related task. Results then identify suspicious elements in the source code.

A more detailed diagram is presented in Figure 2.2. It enumerates all components and artifacts in the system and illustrates the flow of information and the relationships between them.

The tool is divided into these components:

- *Frontends*. This component is responsible for static data collection and instrumentation of the program. There are multiple frontends because a specific frontend must be developed for every source programming language[7]. Frontends act as a gateway from user's programming language to data that are consumed by Aardwolf. Our aim is to make required functionality of the frontend as simple as possible in order to simplify the development of new ones and maintenance of existing ones.

- *Test drivers*. Instead of implementing a custom test driver for each programming language, the goal is to allow integration with as many already existing test drivers and as smooth as possible. The integration

[7]There might be frontends which are implemented for a technology that is underlying for multiple programming languages, such as LLVM or JVM.

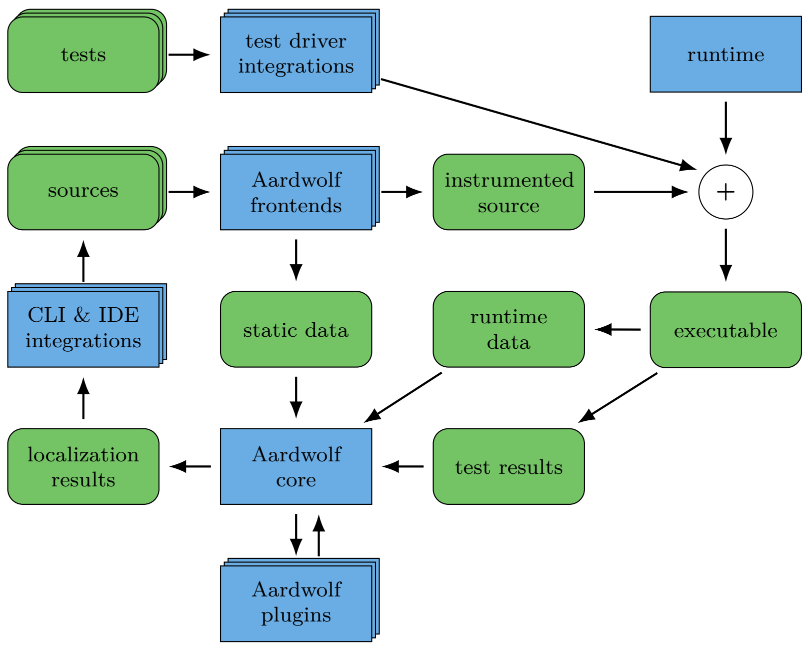


Figure 2.2: Detailed component diagram of the Aardwolf. Blue rectangles correspond to the components of the system, whereas green rounded ones represent artifacts consumed or produced by the components. Components that generally have multiple instances are visually distinguished.

can take form of an extension or a wrapper. However, since the requirements for the integration – as described later – are minimal, it is also feasible for the user to integrate with Aardwolf manually.

- *Runtime.* This component is used by frontends as API in program instrumentation and by test drivers for dividing continuous execution trace into individual test cases. It is theoretically possible to share one common implementation in multiple frontends, but since it is currently rather simple, it is perhaps more convenient to reimplement it in the language the frontend is written in. The custom implementation approach has the advantage that its API can be tailored for the target programming language specifics.

- *Core.* This component is responsible for parsing the artifacts produced by a frontend and the execution of the test suite, and provides the fundamental infrastructure for fault localization plugins and presentation of the results to the user.

- *Plugins.* On top of the core, there are plugins that implement specific fault localization techniques or associated tasks. They use the core's API to compute the necessary information for the feedback to the user

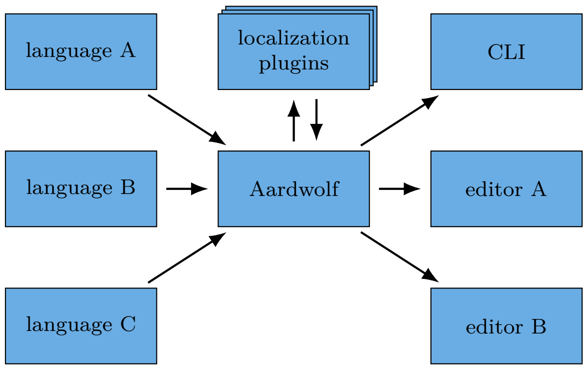


Figure 2.3: Illustration of potential for wide adoption by supporting multiple programming languages and development environments.

(like suspiciousness values or explanations) which is then sent back to the core.

- *Results presentation.* This component presents the computed results to the user and is equally important as the others. To be useful, the output must contain rich information that helps the user to successfully locate the real fault from the tool's output.

This architecture enables potential for wide adoption by supporting many users with their programming languages and development environments of choice. This aspect is illustrated in Figure 2.3.

In the following sections we provide detailed descriptions of individual components and their subcomponents.

2.1 Frontends

The design goal of Aardwolf is to support multiple programming languages on its input. From a program written in such a language, we need to extract relevant information and convert it to an intermediate representation. This representation is general and common for all source languages, therefore it needs to hide unnecessary implementation details, but on the other hand, must contain all the information required by later analyses.

The most-preferred granularity levels in fault localization are methods, statements and basic blocks [6], where the statement level is the most fine-grained and the other ones can be derived from it (with control flow graph and function names as supplementary information). For that reason, our intermediate representation uses statements as its elements.

Program statement represents an abstract action, whose complexity can vary greatly. Some types of statements are common for majority of programming languages but generally they can differ from language to language.

Furthermore, the notion of a statement is rather related to imperative programming paradigm, whereas there are languages which are either fully or partially expression-based (typical examples are functional programming languages). We therefore leave the decision about what exactly statement is to implementers of the frontend without giving some exact specification.

However, we give here some general notes as an attempt to have outputs from all frontends as semantically similar as possible. We consider these constructs as being a single statement:

- Entire expression that computes a value assigned to a variable. Even though such expression might be very complex in some cases, we suppose that identifying a faulty subexpression would be very difficult for any localization technique and unnecessarily cause higher computational costs. These complex expressions could be even advanced syntactical constructs like list comprehensions in Python.

- Conditional expression that branches the control flow to multiple paths. This includes typical constructs like `if` and `switch`, but also conditional statements in loops or advanced concepts such as pattern matching.

- Function call together with all expressions that compute its arguments. Note that every function call with an output value defines a new implicit variable that must be indicated by the frontend. This is necessary for the analysis.

- For each argument of a function, an artificial statement is created. Such statement does not use any variable and defines exactly one – the argument. The order of these statements must match with the order of the arguments. Treating arguments as statements simplifies the internal representation.

As the first step, a frontend must detect such statements in the program and uniquely identify them using a numeric identifier. Then, it generates static analysis data and instruments the program to generate runtime data during the execution. These two tasks are described in the following subsections.

2.1.1 Static Analysis Data

Static data are generated during the analysis of the program without the need of executing it. It provides the information about the program's structure and the location of elements in the original source code.

For every statement, all possible outgoing edges in the control flow graph are listed, in form of the identifier of the statement which is direct successor in that path. This is important for control flow analysis.

```
function: get_range

#1:1 -> #1:2  ::  defs: %1 / uses:  [@1 1:15-1:21]  { arg }
#1:2 -> #1:3  ::  defs: %2, %3 / uses: %1[] [@1 2:5-2:26]
#1:3 -> #1:4, #1:5  ::  defs: %4 / uses: %1 [@1 4:9-4:10]
#1:4 -> #1:6, #1:7  ::  defs:  / uses: %2, %4 [@1 5:12-5:19]
#1:6 -> #1:7  ::  defs: %2 / uses: %4 [@1 6:13-6:20]
#1:7 -> #1:3, #1:8  ::  defs:  / uses: %3, %4 [@1 7:12-7:19]
#1:8 -> #1:3  ::  defs: %3 / uses: %4 [@1 8:13-8:20]
#1:5 ->   ::  defs:  / uses: %2, %3 [@1 10:5-10:20]  { ret }

@1 = <absolute path to the source file>
```

Figure 2.4: An example of the static analysis output from a frontend in a human-readable form. It corresponds to function `get_range` from the motivating example from Figure 1.1.

Each statement contains also the set of variables it uses and defines, in the sense of Definition 1.3. This is important for data flow analysis. Variables are specified by their unique numeric identifiers.

There must be a way to locate the statement back in the source code in order to guide the user with the results. The location consists of the file and the line and column numbers where the statement syntactically begins and ends. Two line numbers are necessary, because complex statements can generally spread over multiple lines, and in such cases, Aardwolf needs to know the whole range in order to provide user with a meaningful information. If the range cannot be determined (e.g., because of limited support by the technology used for developing the frontend), both values must be equal. Aardwolf core might apply some heuristics in such cases.

An illustrative example of the output is presented in Figure 2.4. Note that it is given in an arbitrary human-readable form, but the actual implementation uses a binary format (see Chapter 3). One can clearly see the control flow inside the function and def-use information as well as source code location. It actually contains additional metadata and some special syntax for the variables. Both details are described in the following paragraphs.

The statements are divided into several categories depending on their type. This is useful in later analyses and even necessary for some localization techniques, such as likely invariants described in Section 1.5, where checks are placed at certain program points like return statements, or work by [49], where different statement categories are assigned a different weight. Aardwolf currently indicates the following statement categories (the list might grow in the future): function argument, return statement, and function call. If a statement does not fall into any category, it is treated as "other".

The statements are grouped under the functions to which they belong in the source code. This explicit partition helps in many ways, for example in control flow graph construction or for providing method-level output granularity. The names of the functions must be unique and should be human-readable, since they might be used in the Aardwolf's output. Object-oriented programming languages are supported such that the names of methods of a class are prefixed with the class name. Variants of polymorphic functions (in the sense of ad hoc polymorphism) must be somehow distinguished using an artificial identifier.

We differentiate between three categories of variable use and definition (we refer to them jointly as *access*):

- *Scalar.* This category represents access to symbolic variables. These might be simple values like integers or floating point numbers, but also assignments of whole structures or pointers to variables. In other words, an access is scalar if and only if its value is atomically set.

- *Structural.* Contrary to scalar type, structural access works on subcomponents of the base variable. Subcomponents (e.g., structure fields) of such variable access can be determined by some statically-known unique identifiers. Note that fixed-size arrays and tuples also fall into this category (if they are statically known to be such), because they can be both treated as some form of a structure with numeric identifiers of its fields.

- *Array-like.* This represents two scenarios. First, the source or target variable represents an array, or more generally a block of memory, and its subcomponents can be accessed only via a numeric index which is unbounded from static analysis perspective. Second, it is an access to a structure where the accessor is not known during static analysis (this is more typical for dynamically typed languages, but not exclusive). Note that the expression determining the index to the array might be more complex that just a single variable; it could be an arithmetic expression with multiple variables. We indicate such situations but, for simplicity, we do not specify the concrete expression.

More formally, a variable access takes the following tree form, depicted in Backus-Naur form as follows:

$$\begin{array}{rcl}
\langle\text{Access}\rangle & ::= & \langle\text{Var}\rangle \mid \langle\text{Access}\rangle\texttt{.}\langle\text{Access}\rangle \mid \langle\text{Access}\rangle\texttt{[}\langle\text{Index}\rangle\texttt{]} \\
\langle\text{Index}\rangle & ::= & \langle\text{IndexVars}\rangle \mid \epsilon \\
\langle\text{IndexVars}\rangle & ::= & \langle\text{Access}\rangle\texttt{,}\langle\text{IndexVars}\rangle \mid \langle\text{Access}\rangle \\
\langle\text{Var}\rangle & ::= & \textit{unique identifier of the variable or structure field}
\end{array}$$

The first alternative for ⟨Access⟩ rule (⟨Var⟩) represents the scalar access, the second one (⟨Access⟩`.`⟨Access⟩) is for structural access and the last one (⟨Access⟩`[`⟨Index⟩`]`) corresponds to array-like access. The provided grammar does not specify a concrete syntax in a programming language, it just expresses the concept.

For example, the access tree of C/Python expression `foo[bar.baz + i]` could be conceptually expressed as *foo*`[`*bar*`.`*baz*`,` *i*`]`. Note that the equivalence of two access trees does not imply the equivalence of their semantics in the original source code since we do not consider expression trees and constant values (for example, `foo[i * 2]` and `foo[i + 1]` both have the same access tree but clearly are semantically different).

Note that when analyzing data dependencies, the access trees should be treated in a sort of a subset approach. In simple words, when *foo*`.`*bar* is modified somewhere in the program, then all statements which use *foo*, *foo*`.`*bar* or *foo*`.`*bar*`.`*baz* are affected by this change. When an access tree is assigned, the variables captured by ⟨Index⟩ rule are not considered to be modified as they just specify a particular location inside the modified variable. For example, when *foo*`[`*i*`,` *j*`].`*baz* is assigned, all statements which use *foo*, *foo*`[*]` or *foo*`[*].`*baz* are affected, where the star symbol represents any set of variables, but statements which use *i* or *j* are not.

Talking about statically-known structure is a bit misleading for dynamically typed languages. In such cases, we approach this problem from syntactical perspective. For example, in Python language we consider `foo.bar` to be structural accesses, even though we cannot know the exact structure of `foo`, and `foo['bar']`, `foo[bar]` and `foo[0]` to be array-like accesses (even that the last one might be an access to a tuple).

Note that to support dynamically typed programming languages, we must not require any notion of data type of a variable in the static output. We do not support specifying data types even when the target language is statically typed, because this information is provided in dynamic data anyway, so omitting it in the static analysis data simplifies its representation and reasoning in later analyses.

At the moment, Aardwolf supports yielding data only for statement definitions – a function argument, a variable being assigned to, or a result of a function call. We consider it a reasonable compromise between usefulness and performance costs as we think that analyzing suspiciousness of a statement given the result it produces is more valuable that the values it uses. Even such a limited variable trace is however expensive for processing and storage, thus it should be a good practice for frontends to allow to disable its generation.

All numeric identifiers used in the static analysis data must be unique per entire program. This requirement is necessary for matching runtime data with static data.

2.1.2 Program Instrumentation

To observe the runtime behavior of a program, it must be first instrumented by inserting probes in form of writing statements that reveal the executed path and the data accompanied by it. Such probes work on the statement granularity discussed in the Section 2.1.

The simplest information that a probe can give is to log the statement identifier at every execution of the statement. The output of such instrumented program is called *execution trace*. Such trace is just a long sequence of encountered statements.

On the other hand, dumping the values of variables during the execution produces a *variable trace*. Both traces are actually interwound in a single sequence in our implementation. Instrumentation for variable trace is much more complex than for execution trace.

The approach used in widely-used tool Daikon [31] is to dump the values of variables in scope at certain program points (function entry and exit nodes by default). The data type of a considered variable must be integral (integers, booleans, characters), floating-point, sequence of any of these, or string. All trace values are converted into one of these primitive forms, that is, an array of structures is transformed into multiple arrays, each corresponding to a specific field of the structure. Entire contents of array variables are recorded and processed.

We see some drawbacks of this approach considering our design goals. First, it requires considerable engineering effort for frontend implementers to support transformation of arrays of structures and linearization of pointer-based collections (e.g., linked list). Second, the requirement of statically known data types of variables rules out our desire to support dynamically typed languages. Third, dumping all the "reachable" data at certain program points introduces extensive disk usage if we acknowledge that we want to check the invariants on every store of a variable. However, all these design decisions of the Daikon's author are completely justifiable considering its goals and use cases, which are slightly different from ours.

Instead, we dump the contents only of the memory location which is assigned. For non-scalar accesses, it means that only the value of the indexed location, in case of array access, or the value of the field, in case of structural access, are recorded. If the node is not of a primitive type, it is dumped in its deconstructed form field by field[8]. This structural information should be available at runtime even in case of dynamically typed languages using their value inspection capabilities.

Along with the value of primitive variable or field, its data type is recorded as well. This information is redundant in case of statically typed languages without polymorphism, however, the overhead is negligible in comparison with the entire output. The data type is necessary for proper parsing of the value from the file in binary format, but also enables detection of type changes.

The invariant detection can treat the access tree as an implicit variable without the need of reasoning about its form. Only the semantic meaning

[8]This is not implemented yet.

of the invariant can change. Take for example the invariant "variable v is a constant value". If v is substituted with *foo*.*bar* (structural access), then it means "field `bar` in structure `foo` is a constant value". On the other hand, if v is substituted with *foo*[*bar*] (array-like access), its meaning changes to "elements of array `foo` are, *as far as we can tell*, all of the same value", which is rather a different statement.

Another example of the difference for array-like access is the following: consider that "variable" *foo*[*bar*] did not change its data type, then "array `foo` is homogenous", whereas it's "heterogeneous" if it did.

To summarize, an item of execution trace consists of just the statement's identifier, while an item in variable trace must follow its defining statement and is composed from current data type and its data in raw binary form. We refer to the output that mixes execution and variable traces to just *trace*.

2.2 Integration with Test Drivers

Every software project can be divided into two parts: application software and support software. Application software is designed to fulfill specific needs of a user, whereas support software aids in the development or maintenance of the application software (for example, tests) [11].

A software project can be split code-wise into application code and testing code. Obviously, static and runtime data should be collected only for application code. Therefore, static analysis and program instrumentation is applied on the user's program and not on the testing code. There is one exception to this. We need to split the trace into blocks each belonging to a specific, individual test case. The reason is obvious, we need to apply the information about the status of a test case only on the runtime data belonging to this test.

At the very beginning of each test case, a function from Aardwolf runtime intended for this purpose must be called with the identification (usually the name) of the test case. This information then appears in the created trace. Splitting is then very simple – all trace items beginning with the test case identification item until the next one or the end of the trace belongs to that test.

This "instrumentation" might be provided by an extension of the used test driver or its wrapper. As already mentioned in the beginning of this chapter, this task is also feasible to perform by the user manually. It would usually involve just find&replace procedures and gluing them together with Aardwolf's runtime.

The tool also needs to know which test cases are passing and which are failing. This information is, however, very easy to get. A test driver can output it in a widely-used machine-readable format that is then loaded by Aardwolf. In the worst case, it can be parsed from the human-readable output which is always available.

2.3 Runtime

Runtime component is currently very simple. Its only goal is to dump data into an output file lazily created during the execution. These data are identifiers of executed statements (for execution trace), scalar values of definition statements (for variable trace) and name of a test case (for dividing the whole trace into traces per test case). The first two are used by an instrumented program, whereas the last one must be handled by the integration with test driver.

In the future, we might implement various space usage optimizations. For example, [31] uses an artificial status variable to indicate whether a variable has changed since last execution of the program point. However, this approach introduces another challenge for frontend implementers since all function signatures must change to pass status variables of the arguments. We might thus need a different technique which would implement the status checking in the runtime itself, completely hiding the details from the frontend.

2.4 Core

Core provides plugins with the underlying infrastructure. Its driver glues all the necessary components together, from parsing the data over running the plugins to present the results to the user. For plugins it offers several high-level data structures (such control flow graph or def-use sets) and allows to implement custom ones from the raw data.

2.5 Plugins

Plugins are extensions of Aardwolf that use core's API to perform the actual fault localization or associated tasks. They are supposed to be self-contained and non-interacting with other plugins (at least not directly). Plugins may implement their own data structures from those provided by the core if needed.

We divide plugins into three categories which form a kind of pipeline (which is depicted in Figure 2.5):

(1) *Data pruning.* These are run before any fault localization. Their task is usually to filter or prioritize the input space according to some criterion in order to improve the results and reduce the computational costs. Examples of such plugins are test case prioritization and program elements prioritization, described in Section 1.6 and Section 1.7, respectively.

(2) *Fault localization.* These are the plugins responsible for fault localization itself. They output a sequence of program elements ordered descending by their suspiciousness score.

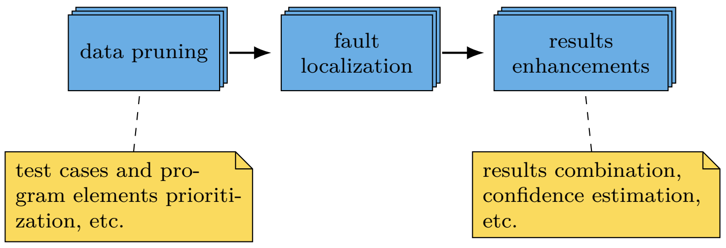

data pruning
fault localization
results enhancements
test cases and program elements prioritization, etc.
results combination, confidence estimation, etc.

Figure 2.5: Processing pipeline of the localization process.

(3) *Results enhancements.* These are run after the fault localization in order to further enhance or refine the results or provide the user with supplementary information. Examples include combination of results from multiple localization techniques (Section 1.8) or producing various metadata (Section 1.9).

We plan to extend this notion of a pipeline further by allowing the user to specify its own pipeline, however complicated, in the configuration file. This is quite a common practice for example in continuous integration setups in today's software projects.

The task of fault localization is to generate suspicious elements along with some context and rationale. Their output is a list of items consisting of the following information:

- *Location.* File's numeric identifier and line numbers where the item begins and ends. Note that this allows to express all reasonable levels of localization granularity (statements, blocks and methods). These excerpts can be then copied to the terminal output or highlighted in an editor. As a general rule, fault localization plugins should stick to the statement level (if it is not against their very nature) and let grouping and contextualization to be done by the results enhancements stage.

- *Suspiciousness.* The value of suspiciousness estimated by the technique. This is used mainly for sorting the items which is handled by the core. Furthermore, the core also normalizes the values to be in $[0, 1]$ range in order to present a unified scale across all fault localization techniques. The actual absolute values are not relevant for the user, but the unified scale may be important for results enhancement plugins.

- *Root statement.* An indication which the statement was originally the reason for blaming the code block. This is mainly for implementation purposes as it can help some localization refinement techniques to achieve their goal (consider for example combining the results from multiple sources).

- *Rationale.* Human-readable description of the reason why the technique blames the item to be suspicious. It should be brief but helpful. Moreover, it can contain anchors into the source code that are presented to the user in an appropriate way (imagine for example a rationale "Assignment to variable *foo* is unusual at [1], it usually happens at [2].").

Every fault localization plugin creates such a list. Results enhancement plugins then have access to these lists and can refine or combine them. The source of suspiciousness list can be identified by a name, and the behavior of results enhancement techniques can depend on this information.

Along with the list of suspicious items, some additional metadata might be provided. Since they are supposed for the user and not for some successor analysis, they can take any form. As an example, a plugin can estimate localization confidence as described in Section 1.9 and report this value to the user.

2.6 Configuration

The cornerstone of the integration of Aardwolf within an existing project is its configuration file. It has a textual, human readable form in the YAML format[9]. The only two required items are *script* and *plugins*.

Script item specifies a sequence of commands needed for getting all data that Aardwolf needs. At the very minimum, it should compile source files with Aardwolf analysis and instrumentation extensions (in case of compiled language) and execute the test driver while collecting the test results. This is however very specific for each project. The compilation and testing machinery already existing in the project should be reused as much as possible.

In plugins item, the user specifies the plugins which they want to use, optionally giving them custom names and specifying their options. Examples of possible options are name of the ranking metric for spectrum-based technique or disabling certain program points for checking likely invariants. Since the user can give a custom name to the plugin, there can be multiple instances of the very same technique, only with different options.

Other optional configuration settings include output directory for data generated for Aardwolf or the number of top-ranked elements which will be displayed to the user (by default, this number is 10). We expect the number of such settings to grow in the future depending on the user needs.

2.7 Design Applicability

In this section, we review a broad spectrum of methods proposed in the literature and speculate how they could be integrated into presented design of

[9] `https://yaml.org/`

Aardwolf or what would need to be added to Aardwolf in order to support them. Our review cannot be exhaustive, but we use published surveys [3, 4, 47] as our basis.

2.7.1 Slice-Based Techniques

Program slicing reduces the program domain which the programmer needs to examine by filtering statements that do not affect a program point of interest (e.g., return value where the error is revealed by a test case) in the means of control- and/or data-flow. Such techniques do not – at least in their base variant – assign any form of suspiciousness score to program elements, only filter the irrelevant ones. Therefore, in Aardwolf design they would rather serve as a helper technique in form of a data pruning plugin which would be used together with an actual fault localization technique of choice.

Since Aardwolf frontends emit control flow graphs along with variable def-use sets as well as execution trace, both static and dynamic slicing are possible in Aardwolf. However, as the representation used is rather simplified, precise data-flow analysis is either difficult or impossible. This is the case especially for pointer analysis which is difficult in general [15].

This implies that slicing support in Aardwolf is limited to imprecise approximations. Our arguments why Aardwolf should not aim for including more precise pointer analyses are following:

- Pointer analysis for dynamically typed languages is even more difficult. Although such analysis would benefit static languages, we think that future work should be rather invested into techniques that benefit all types of languages.

- Enhancing intermediate representation produced by frontends and consumed by the core with pointers-related information would complicate development of frontends, especially for languages where this information is inapplicable (e.g., dynamic languages, as already mentioned).

- Low-level pointer analysis might be unnecessarily hard and imprecise in cases where the semantics of the source language, such as type system, provide more specific and easier-to-obtain information (e.g., non-aliasing property) about the data-flow behavior. Aardwolf intermediate representation could be then augmented with such explicit, optional data-flow information to help its reasoning.

2.7.2 Program Spectrum-Based Techniques

This fault localization family is thoroughly described in Section 1.3. Its implementation is quite straightforward. However, our tool needs to be flexible enough to provide ways how to gather various types of spectra. We now assess which types of spectra taken from [3, 8, 17, 19] are currently possible in Aardwolf[10].

The simplest type of spectra is *coverage* of code entities of various granularity, mainly *statements*, *basic blocks* and *methods*. All these information can be easily extracted from the execution trace and control flow graphs. Similar situation is for *branch hit spectrum* that records how predicates are executed. This information to some extent characterizes the behavior of the program. Control flow information and execution trace in Aardwolf can be conveniently used for gathering such spectrum.

An extension of branch count spectrum, called *predicate count spectrum*, makes use of implicit predicates not actually present in the program. For instance, [50] uses several predicates on numerical values at function returns and variable assignments comparing them to zero or other constants. For this use case, the variable trace can be utilized.

For *method calls sequence hit spectrum*, execution trace can be again used, since each statement is associated with a function name which it is located in. More detailed information is provided by *path hit spectrum* which records the execution of inter-procedural, loop-free paths. It can be gathered from the execution trace, the only difficulty is discarding loops from the trace.

Def-use pair (abbreviated as du-pair) *spectrum* represents the coverage of a pair of statements where one statement uses a value of a particular variable defined (i.e., assigned) by the other statement. Since the Aardwolf intermediate representation of programs contain definition and use information along the statements, this type of spectra is also relatively easy to gather.

Time spectrum records the execution time spent in individual functions. Currently, there is no way how to measure time using Aardwolf, but – if proved valuable – runtime data could contain a timestamp of the logged item.

Although families like *statistics-based techniques* (e.g., [50, 51, 52, 53, 54, 55]) or *machine learning techniques* (e.g., [56, 57]) are justly distinguished in the literature from the program spectrum-based family for their fundamental difference in analysis, regarding their architecture and required data, they usually fall into this category. Thus the Aardwolf design does not prevent such techniques to be implemented. Some require additional information as for instance [58], where execution frequency vectors over continuous sections of the program are collected. These can be obtained from the Aardwolf intermediate representation and execution trace as well.

[10]Many of mentioned types spectra can be divided into *hit* and *count* types. Their distinction is certainly important for the evaluation results, but not so much for their gathering. This section therefore uses these types interchangeably.

2.7.3 Program State-Based Techniques

Authors of [59, 60] proposed a technique which isolates the root cause of a failure by contrasting *program states*, where program state is represented by values of the variables. Their method was later extended further by some other works. Although Aardwolf produces variable trace of values of some primitive types, this is very limited (and presumably inefficient) compared to the data processing of techniques in this family, which capture entire program state with a *memory graph* containing all values and all variables of the program.

Moreover, these techniques are usually accompanied by failure-inducing input generation and simplification, or replacing actual values of variables in failing executions. Both are impossible in Aardwolf architecture since it considers the test suite execution as given and immutable. However, we think that experimenting with input generation and property-based testing[11] would be valuable for Aardwolf effectiveness, complementing the test cases prioritization techniques presented in Section 1.6. The goals of such generation could be to find failing input as simple as possible and passing inputs close to such failing one in terms of execution similarity.

2.7.4 Model-based Techniques

What connects the techniques from this group is that they create some sort of model generated from the program and potentially the test suite execution information. However, the type of the model may vary a lot. One example of such technique is presented in this thesis in Section 1.4 and similar program dependencies-based are implementable in Aardwolf to certain extent.

Other methods are however heavily dependent on the programming language semantics [61] and require advanced static analysis techniques like abstract interpretation [62] or symbolic execution [63]. There are also methods (e.g., [64]) that make use of the counterexample found by a model checker that is based on a formal specification of the program. These technique could not be supported by our design, at least not naturally.

2.7.5 Mutation-Based Techniques

Techniques based on the information from mutation analysis modify program elements and analyze which ones affect failing test cases more frequently than others. This localization family does not fit into the design of Aardwolf at all since the approach mutates the program, which we consider immutable[12], and is heavily programming language-dependent, which interferes with our goal to have as uncomplicated frontends as possible. Therefore, this type of

[11] An automatic input generation with constraints specified by the tester and checking if the output satisfies a specific property. A prominent example of such technique is QuickCheck tool.

[12] Apart from the required instrumentation to gather runtime information.

techniques are very unlikely to be supported in the future. Moreover, current implementation does not support reacting on the execution results to adjust the next steps in the fault localization process. This issue is discussed in the following paragraphs.

2.7.6 Semi-Automated Techniques.

In order to improve localization effectiveness, there exist methods (e.g., [65, 55]) that incorporate the information given by the user who is interacting with the tool, usually by specifying whether a suspicious element is actually faulty or is healthy instead. Such techniques proceed in iterations, each step refining their results based on the user feedback.

There is no obstacle in the architecture of Aardwolf preventing the interactive mode to be part of it per se. As far as the data gathered from static analysis and test suite execution remain immutable, a technique is free to take user feedback and adjust its results appropriately. However, it is not supported by the current implementation.

2.7.7 Techniques with Learning Phase

Some approaches (e.g., [47, 48, 66]) require the learning phase, in which the technique is run on a benchmark data with labels, where labels identify program elements that are real root causes. The output of this phase is a trained model which is then used in the deployment phase to perform the localization or a related task for the user.

This split into two modes of operation is not conveniently available in Aardwolf yet. But considering recent advances and popularity of machine learning applications, we think that such addition with provided infrastructure and data would be very significant enhancement for Aardwolf in the future.

2.7.8 Information Retrieval Techniques

Information retrieval techniques estimate suspiciousness of program elements by analyzing the bug report, created by a user, and the source code. Contrary to all families discussed so far, this approach does not use any execution information. Aardwolf is currently designed around analysis of program structure and runtime information from test suite execution. However, we consider it to be trivial to employ an external source, such as bug report text, and plugin developers could implement it by their own. Source code is accessible by using the absolute path to the source file which is present in the intermediate representation.

2.8 Conclusion

We argue that Aardwolf can support a decent portion of techniques. Taking the statistics from [3, 4], it is the majority[13] of methods published in the literature. Therefore it fulfills our goal to be general and widely applicable.

Language independence – another goal of ours –, however, poses a major limitation for Aardwolf applicability because of the absence of way how to mutate the program and its execution, since it is highly dependent on the programming language and the runtime. There are other extensions that would increase the applicability of Aardwolf which are not prevented by its current design: mainly providing infrastructure and data for pre-training phase to ease the support for machine learning approaches and implementing an interactive mode to be able to gather user feedback for adjusting the results.

Aardwolf is by default highly extensible with multiple ways how to extend it. First, by implementing a new frontend, one brings the whole fault localization groundwork for another programming language right away, bringing the support to wider range of users. Second, new localization or related techniques can be implemented in the form of plugins, building on top of already implemented framework without the need to deal with general and common things like instrumentation or user interface. The results presentation is the third dimension where Aardwolf can be extended, providing support for different editors or IDEs, or even continuous integration services, etc.

Last but not least, Aardwolf addresses the recommendations of user studies with first-class support for giving prediction rationale, contextualization, and producing metadata, aiding the reasoning about the computed results. This and its focus on convenient integration with existing projects delivers good user experience and gives Aardwolf the potential for wide adoption.

[13] Spectrum-based techniques is the most popular category by large margin in the literature. Support for families like model-based is dependent on their specific requirements.

Chapter 3

Implementation

This chapter discusses details of the implementation of the Aardwolf fault localization tool. Individual sections describe corresponding components presented in the Chapter 2.

We implement most of our tool in Rust programming language. The main exception is frontends which we implement in technology that is most suitable for given task, in particular, C++ for LLVM frontend and Python for Python frontend. Rust performance is comparable to C/C++, which helps making Aardwolf scalable, but on the other hand, is designed to eliminate many classes of bugs at compile-time (mainly memory safety), hence it promotes reliability.

Aardwolf binaries can be currently built from source but we plan to provide more convenient ways of its distribution. All code is publicly available at `https://github.com/aardwolf-sfl/aardwolf`. To ease the building process, we provide an installation script written in Python that checks all required dependencies, compiles individual components, and copies the artifacts to default or specified directory.

3.1 Frontends

Both frontends we implement in this thesis have very similar structure from high level perspective. Essentially, there are three components: *program analysis*, *static data generation* and *program instrumentation*. The most complex one is for program analysis whose main goal is to identify the statements and variable access trees that the statements use and assign to. This is the most language-dependent part of the frontend. Static data component then just takes the result of program analysis and writes it into relevant files. Instrumentation phase instruments identified statements with the calls to an Aardwolf runtime.

In practice, the language frontends usually work on a single module (often represented by a file). A program is then composed from multiple modules. In order to guarantee the uniqueness of statement identifiers across the whole

```python
# AARD: function: __main__

# AARD: #1:1 -> #1:2  ::  defs: %1 / uses:  [@1 4:1-4:18]
condition = False

# AARD: #1:2 -> #1:3  ::  defs: %2 / uses:  [@1 7:1-7:6]
n = 3

# AARD: #1:3 -> #1:4, #1:5  ::  defs:  / uses: %1 [@1 10:7-10:16]
while condition:
    # AARD: #1:4 -> #1:3  ::  defs: %2 / uses: %2 [@1 12:5-12:11]
    n += 1
```

Figure 3.1: An excerpt of a test from the Python frontend.

program, they are composed from two numbers: statement unique number in the module and a file identifier of the source file. We use inode number of the file obtained from POSIX standard compatible `stat` function on Unix systems and file index obtained from `GetFileInformationByHandle` function on Windows system. We consider this to be reasonably sufficient platform compatibility.

To aid the development of a frontend, a Python package is created providing several useful tools such as a binary format parser which transforms Aardwolf raw data into a human-readable form, which is especially useful for debugging, or a framework for testing static analysis and tracing behavior of the frontend. The tests are written as short code snippets in the language for which the frontend is written, with human-readable Aardwolf data items in the comments. An excerpt of a test of static analysis from Python frontend is depicted in Figure 3.1. The framework also takes care of normalizing the actual raw data, for instance changing file system-based file identifiers to the sequential one-based index.[14]

Based on our experience while developing both frontends, we recommend to future developers to work closer to the source level (as Python framework does) rather than on a low-level representation (LLVM frontend case). The advantage is that high-level statements could be much more directly identified and exact and complete location information exists.

3.1.1 LLVM Frontend

LLVM frontend is implemented as multiple passes for LLVM compiler and operates over its bytecode (sometimes referred to as bitcode). This intermediate representation of programs is low-level – being closer to real machine code than high level languages –, but expressive and lightweight at the same

[14]This approach is inspired by `FileCheck` tool used for verifying outputs in LLVM project.

time. An LLVM "program" is composed from instructions like "store this virtual register to memory" or "jump conditionally to one of these basic blocks", and rich metadata such as data types or source level information. For proper description we refer to the official Language Reference Manual[15].

The program analysis is a pass running on the entire module as it needs to gather the information about the whole program (or the part of it represented by the module). It iterates over all instructions in all functions with bodies and, if the instruction represents a statement (e.g., `StoreInst`, `BranchInst`, etc.), the values it uses, its output variable (if any), and its location in the source code are obtained. The most interesting is finding which variables a statement uses. From the root instruction a graph-like traversal backwards over def-use edges is performed until all valid access trees are reached. For instance, reaching `AllocaInst` represents finding a scalar, local variable, whereas reaching `GetElementPtrInst` represents either structural or array-like access and the traversal must continue to find its base (instance or array) and accessors (field or variables used in index). Note that this process requires debug symbols enabled when emitting the LLVM bytecode.

Instrumentation is another module pass. It just iterates over previously identified statements and adds the tracing calls and (if applicable) the value of the statement output. Note that for statements like assignment and function calls, runtime call must be inserted after the statement, whereas for control flow statements, which are terminators of basic blocks, it must be inserted before them, because otherwise the basic block would be invalid, having an instruction after its terminator.

All Aardwolf passes are implemented in both legacy and upcoming pass manager[16]. The purpose of legacy implementation is to use it directly within Clang (the C language compiler based on LLVM) which still uses the legacy pass manager at the time of writing this thesis. The upcoming pass manager interface is future-proof and allows to conveniently implement custom command line application on top of the passes.

The preferred way of frontend usage with Clang is as follows (assuming Linux environment and omitting everything project-specific):

```
clang -Xclang -load -Xclang libAardwolfLLVM.so \
    -c -g -o <program>.o <sources>.c
clang -o <exec> <tests>.c <program>.o libaardwolf_runtime.{so,a}
```

where `libAardwolfLLVM.so` is the shared library composed from all Aardwolf's LLVM passes and `libaardwolf_runtime.{so,a}` is the Aardwolf C runtime library (either dynamically or statically linked). The arguments `-Xclang -load -Xclang <library>` instruct the Clang compiler to use given pass during the compilation.

[15]`https://llvm.org/docs/LangRef.html`

[16]At the time of writing this thesis, LLVM is in the middle of the phase of moving from the old pass manager to the new one.

One disadvantage of using LLVM is that its location information does not include the whole span (beginning and end) of the instruction in the source code, only a single position. For this reason, the location details of suspicious elements is not as precise and user-friendly as we would like. However, we consider it a good enough approximation.

Although LLVM is used by multiple real-world languages, we expect that the current implementation would not work with them out of the box and would require various tweaks and feature additions. Regarding C language, it correctly works on test inputs that we created and checked. However, support for the whole standard would require to explore what LLVM IR Clang compiler generates on every syntactic construct allowed by the standard and check if our passes handle it correctly, which is currently beyond our means.

3.1.2 Python Frontend

Python frontend heavily utilizes built-in module `ast` (abstract syntax tree). This guarantees full support for complete Python syntax from parsing perspective as well as maintenance of this module in the future. Note that we use Python in version 3.8 where several changes were introduced and therefore our frontend only supports this version of Python. However, adding support for previous minor versions should be fairly trivial[17]. We used to use also the built-in module for symbol tables (`symtable`), but there were certain limitations in the behavior and exposed API. Thus we had to create a custom implementation tailored to our needs.

The class responsible for program analysis implements `ast.NodeVisitor` – a visitor pattern-based interface for traversing the AST of a program. Reasoning about individual nodes locally is very convenient. Each visitor method essentially visits its children and, if the node represents a statement (for instance, assignment or conditional), it records the node along with its uses and definitions. For outputting the set of successors for each statement, we implement a very simple CFG construction used in the visitor. To appropriately distinguish between different variables of the same name, we use symbol tables generated from the tree.

The instrumentation class implements `ast.NodeTransformer` which has the same interface as `NodeVisitor`. The only difference is that it modifies the tree depending on the return value of the visitor methods (e.g., by returning a modified node). All visitor methods just visit their children and instrument the node using one of the two instrumentation techniques. *Statement* instrumentation is applied on nodes inside a list of statements (e.g., inside a function). In such case, instead of returning the original node alone, a runtime call is returned along with it and is appended to the list. Note that we need to distinguish between non-terminating and terminating statements and use the

[17]The main change in 3.8 is deprecation of several AST node classes and replacing them with a new one covering all the cases.

valid order. *Expression* instrumentation is applied on expressions (for which we cannot just add a new statement to a list). Expressions are logged using a call to `write_expr(expr, id)`, which logs the id and returns the expression. This way, the visitor method can still return a single node – a requirement of the transformer class – but tracing the statement in the execution as well. One interesting case is function calls, which need to be traced before calling. Hence the expression passed to `write_expr` is the function itself and it is called after it is returned from this runtime call.

The Python frontend uses its own runtime implementation. Although wrapping the C library is theoretically possible, the current simplicity of runtime outweighs the costs of such wrapping. If the official runtime implements some complicated optimizations in the future, this choice might be reconsidered.

The reference and most-widely used Python implementation is CPython. Although it compiles the sources when executing a program under the hood, and for optimized package distribution, from the user's perspective it acts as an interpreter. Therefore, there is not a distinct compilation step as in C case. In order to analyze and instrument user's source code, we adopt so-called *meta import hooks*[18] feature provided by Python. This technique is based on implementing a custom module *finder* which can specify a custom module *loader* for modules being imported during runtime. Aardwolf provides an `install` function which accepts a name or module specification of the user's package to test. This function should be called at the very beginning of the test suite execution as it injects a custom finder which, if encounters an import from the specified package, loads the file while analyzing and instrumenting it first. The approach intentionally works only for source code-based packages (that is, not somehow-compiled ones), since we need the source level information.

A very common practice in testing Python programs is to use `assert` statements that raise an exception when the condition is not met. Such exception is caught by a test driver and reported to the user as failed test. We thus provide a very convenient way how to integrate Aardwolf with such test suites using `wrap` decorator which is applied on test case functions in the suite. Before running the test case function, it logs its name using the corresponding runtime call. Then, if the test raises an exception, the test status is considered failing and as such is logged, while reraising the original exception to be consumed by the test driver. If the function finishes successfully, the test status is considered to be passing.

To avoid annotating every test case with the `wrap` decorator, we provide `wrap_module` function as well, which automatically wraps all functions in the module. Because not every function in the module is necessarily a test, `wrap_module` allows to optionally filter them in several ways (e.g., using regular expressions or specifying a set of names to be ignored).

[18] `https://docs.python.org/3/reference/import.html#import-hooks`

However, this is not the only way how to use Aardwolf in Python. For example, a popular test framework pytest[19] offers many hooks which can influence the test suite execution. Utilizing these hooks, one can trace the test names as well as their results.

3.1.3 Data Formats

There are three types of files that Aardwolf uses for its working: data produced by static analysis performed by the frontends, runtime information produced by running the test suite, and test results taken from the test driver.

For static and runtime data, a custom binary format is used. The reason is twofold: memory efficiency and easy and fast parsing, in comparison with a textual form. The files contain all information described in Section 2.1 in a compact structure. At the moment, we do not consider it stable and will likely be making breaking changes. However, we expect to stabilize it in the future and use a proper versioning schema with backward compatibility.

We are not aware of any test results output format which would be commonly used across different ecosystems. It is always dependent on the particular test driver. The output is usually in a human-readable form, sometimes accompanied by rich metadata such as value difference or similar. In IDE-integrated test drivers, it is presumably in a machine readable format. The Aardwolf core is able to parse a simple format we chose, which consists of each test case on an separate line, prefixed with either `PASS: ` or `FAIL: `. It is a compromise between human readability and simple parsing. As a side note, this exact format is used by ManyBugs benchmark dataset [67]. There are two ways how to integrate test drivers not directly supported by Aardwolf core (as for example exception-based runners are by the Python frontend). Either a custom results reporter is developed and plugged into, if the driver supports that (preferable), or a translation post-processing tool is implemented and used in the script in project-specific configuration file.

3.2 Runtime

The role of runtime is to record execution information to the trace file and thus is used by instrumented program for statements and values and test driver for dividing the entire traces into individual test cases by logging their name. The whole trace is written to a single file which is created on the first call to any of the functions provided by the runtime. That is, every such function first checks if the file handle for the trace is already available to be written to, and if not, it is initialized appropriately. A usual practice is to initialize all data structures of the logging mechanism with an initialization call inserted at the beginning of the program. However, the target program that we instrument

[19] `https://docs.pytest.org/`

might represent a library without the execution entry point, so we decided to use this lazy approach.

3.3 Core

Core is implemented as a command line application written in Rust programming language. The driver first finds the project-specific configuration file named `.aardwolf.yml`[20] by default, but this name can be overridden using the command line argument. The directory where the file is located is considered as the project root. It then proceeds with the following steps:

(1) *Output directory preparation.* The output directory can be specified in the configuration file and is relative to the project root. By default, its value is `.aardwolf`.

(2) *Script execution.* The script for generating data required by Aardwolf is retrieved from the configuration file and a temporary file with the script contents is created. The file is then executed with bash shell[21]. The script command is populated with several environment variables to properly generate the data or access Aardwolf libraries, mostly absolute paths to directories (e.g., output directory) or files (e.g., expected path for file with test execution results).

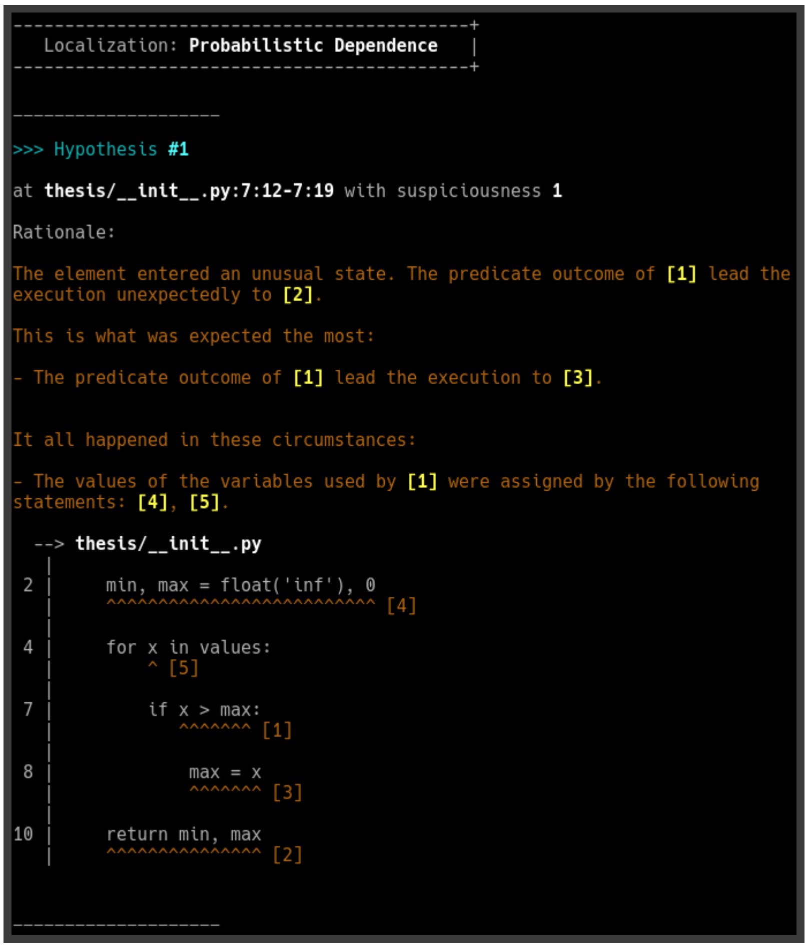
(3) *Data loading.* After script is executed, we expect that all required data files are generated. All of them are loaded and parsed into internal structures which greatly resemble the raw format of the data files. Note that there might be multiple files containing static analysis data, which might be even nested in directories. This is because the frontends usually work on per module basis and create corresponding file for each module separately.

(4) *Plugin initialization.* Plugins, as specified in the configuration file, are initialized along with their options. Each plugin item is identified by its name (if omitted, the name of the plugin itself is used).

(5) *Fault localization.* Initialized plugins are then used to perform the fault localization. As described in Section 2.5, there are three phases. In the real implementation, plugins are not categorized into these phases, instead, all plugins implement all three phases, possibly doing nothing in those which they are not interested in. This simplifies the management and allows to use all capabilities of Aardwolf if needed.

[20]Prefixing configuration files with dot and putting them inside source code repositories is a common practice nowadays.

[21]Bash is common shell for Linux system, and is officially available in Windows Subsystem for Linux since Windows 10.



```
--------------------------------------------+
   Localization: Probabilistic Dependence    |
--------------------------------------------+


--------------------

>>> Hypothesis #1

at thesis/__init__.py:7:12-7:19 with suspiciousness 1

Rationale:

The element entered an unusual state. The predicate outcome of [1] lead the
execution unexpectedly to [2].

This is what was expected the most:

- The predicate outcome of [1] lead the execution to [3].


It all happened in these circumstances:

- The values of the variables used by [1] were assigned by the following
statements: [4], [5].

  --> thesis/__init__.py
    |
 2  |     min, max = float('inf'), 0
    |     ^^^^^^^^^^^^^^^^^^^^^^^^^^ [4]
    |
 4  |     for x in values:
    |         ^ [5]
    |
 7  |         if x > max:
    |            ^^^^^^^ [1]
    |
 8  |             max = x
    |             ^^^^^^^ [3]
    |
10  |     return min, max
    |     ^^^^^^^^^^^^^^^ [2]


--------------------
```

Figure 3.2: A sample of terminal output from Aardwolf.

(6) *Results presentation.* Calculated results are in the end presented to the user. At the moment, Aardwolf implements two interfaces – a rich terminal output (a sample is depicted in Figure 3.2) and machine readable results in JSON format which can be consumed by external tools.

The memory management of raw data heavily utilizes an *arena*-based allocation, where arena is a contiguous block of memory to which new items are appended sequentially and identified by a space-efficient numeric identifier.

As a preliminary optimization driven by very high memory usage of Aardwolf during our experiments, we improved two things in our otherwise naive implementation. We limit the size of all arena indexes to 4 bytes. This is not only optimization compared to our previous implementation (three times smaller execution trace), but is also more efficient than using references, which would be of double size on today's most common 64-bit platforms. The second optimization is compression of data obtained from variable trace when loading them into memory. Consider for example a 64-bit variable which is often zero. Storing it always as 8 bytes is significant waste of space. If the actual value of a variable allows it, we therefore store it in a more compact form and properly decompress it when the actual value is needed. This comes at slight computational cost, but reduces the memory usage, although actual savings are very dependent on the actual behavior of the program.

Plugins do not use the raw data parsed from files directly, instead, we use a concept we call "queries". A *query* is a high-level data structure (e.g., control flow graph or statement coverage spectrum) which is created from raw data, possibly using other queries as its dependencies. We encourage plugin creators to implement their own queries if necessary. All these high-level data structures are computed lazily and are memoized. In the future, we might implement a cache flushing mechanism in case of memory usage stress. The queries are stored using a key created from type system level information.

During the whole computation, a process log is maintained by the driver. It contains info and debug messages, warnings and errors, and time measurements of various steps. Such information is useful for both users and developers of Aardwolf ecosystem.

3.4 Plugins

In the current implementation, all plugins are part of the repository and compiled into the Aardwolf executable. Nonetheless, all of them are initialized and used through a common interface. Once the interface is stable enough, we will implement the infrastructure for loading the plugins dynamically as shared libraries in order to achieve high level of flexibility by allowing users to use new, independently developed plugins without the need of recompiling Aardwolf.

The API provided to the plugins is intended to be capable of doing all the tasks described in Chapter 1. We also strive to make it as convenient as possible and to reduce the implementation efforts for the plugin developers.

CHAPTER 4

Experimental Evaluation

The purpose of this chapter is to demonstrate quantitative analysis about Aardwolf's effectiveness and performance and to illustrate how much effort is needed for integration to an existing project. As the objective of this thesis is an implementation of an extensible and user-friendly tool, and not to propose a new localization method, we refer to other literature for more thorough evaluation specific to fault localization methods. To show that our tool is functional, we present the results for a widely-used dataset.

We publish openly our infrastructure for all experiments performed in this chapter to provide reproducibility. Everything is wrapped in a Docker[22] image and thus is executed in an isolated and already set-up container, independent on the host environment. It is available at `https://github.com/aardwolf-sfl/evaluation`.

4.1 Localization Effectiveness

In this section, we evaluate our tool on a standard dataset with common evaluation metrics and present the results. Its main purpose is to compare our implementation with results published in the original literature where the techniques were proposed.

4.1.1 Evaluation Metrics

Given a fault localization technique and a program consisting of n *executed* elements (statements, methods, etc.) with a single known faulty element e_f, and a test suite for the program, a numerical measure of the effectiveness of the fault localization technique can be computed as follows:

(1) Run the technique on the program with the test suite to produce the list of elements sorted by their suspiciousness.

[22]`https://www.docker.com/`

(2) Let n_f be the rank of e_f in the list of elements.

(3) Compute a selected metric using n_f and n to evaluate the effectiveness of the technique.

There are multiple scoring metrics that have been proposed in the literature. The most commonly used one is *EXAM score* [8, 10, 3], which is the percentage of program elements that have to be examined until the faulty element is reached. The formula is

$$EXAM = \frac{n_f}{n} \times 100\,\% \,. \tag{4.1}$$

This metric represents the developer's effort to find a fault using the list produced by the technique. The score value is relative to the program size, the lower the value of EXAM score is, the better.

To address the findings and recommendations of published user studies [7, 9], an absolute scoring metric called *Hit@k* [45] was introduced. It indicates the number of bugs that are discovered when inspecting top k elements in the ranked list. Mathematically speaking, the formula is as follows:

$$Hit@k = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}_{k,n_f^i} \times 100\,\% \,, \quad \mathbb{1}_{k,n_f^i} = \begin{cases} 1 & \text{if } n_f^i \leq k, \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

where m is the number of evaluated program versions and n_f^i is the rank of faulty element in the list when evaluating i-th program version.

According to [6], by far the most preferred maximum number of examined statements by practitioners is 5. In the literature [45, 47], number 10 is commonly used as well. Therefore we use Hit@5 and Hit@10 in our evaluations.

4.1.2 Subject Programs

We use Siemens dataset [68] for the evaluation. It is the most frequently used benchmark for this task in previous studies [3, 4]. Presenting our results on this benchmark thus allows to directly compare our work with the literature.

The suite was downloaded from `https://sir.csc.ncsu.edu`. It consists of seven small programs whose characteristics are summarized in Table 4.1. These programs are simple, yet complex enough to be considered realistic. The artificial faults were seeded for experimental purposes by the dataset creators. Each one contains single fault per "version" and is accompanied by a large test suite. For more thorough description we refer to the original article [68].

In works which we compared our results with, eight versions were omitted for one of these reasons:

- There were no syntactic differences in C file between the faulty and corrected versions of the program. That means that the fault was in a

Table 4.1: Characteristics of Siemens dataset.

Program	SLOC	Faulty versions	Tests	Description
replace	512	32	5542	pattern replace
tcas	141	41	1608	altitude separation
print_tokens	472	7	4130	lexical analyzer
print_tokens2	399	10	4115	lexical analyzer
schedule	292	9	2710	priority scheduler
schedule2	301	10	2650	priority scheduler
tot_info	440	23	1052	information measure

header file. Although a fault localization tool should be able to detect these non-executable parts of a program, this is the limitation of majority of techniques.

- Faulty versions caused segmentation fault when executed on the test suite.
- None of the test cases failed when executed on the faulty version of the program.

In particular, it was versions 8, 14, and 32 of *replace*, versions 4 and 6 of *print_tokens*, version 9 of *schedule* and *schedule2*, and version 38 of *tcas*. For reasonable comparison, we omit these versions as well.

4.1.3 Faulty Elements Determination

In order to evaluate a fault localization technique, we need to determine which elements in the program are faulty. This is usually done by comparing the changes between the buggy version of a program and its corrected version. We follow principles described in [10] – a large replication study evaluating fault localization techniques on real-world faults.

The standard technique for evaluating fault localization is to blame the elements which were modified or deleted in the developer patch. In cases when the patch consists of code additions only (that is, faults of omission), a fault localization technique should report the statement that immediately follows inserted statement.

There are multiple complicated cases which cannot be determined in such a straightforward way. We give their overview and commentary in Appendix A.

Real-world bugs usually span multiple statements [10]. There are various ways how to approach such cases. We follow the approach in existing work [10, 47] and consider a fault localization technique successful if it detects any one of the faulty elements. This assumes that by pointing out one statement, the developer understands the bug and can infer the other faulty statements.

4.1.4 Elements with the Same Suspiciousness Score

It is common that a fault localization technique assigns the same suspiciousness score to multiple elements. For our evaluation, we assume that a fault localization technique outputs elements with the same suspiciousness score in an arbitrary order (although it does not need to be the case, as for example in [13]). In the presence of such ties, the total number of elements that the developer needs to examine varies randomly.

To address this issue, two levels of effectiveness are computed [3]. Consider a portion of the ranked list consisting of elements with equal suspiciousness, containing the faulty element, which starts at rank n_{best} and ends at n_{worst} inclusively. When using these two ranks, two levels of presented metrics, EXAM and Hit@k, are produced. Since ties make the actual effectiveness of fault localization more uncertain, by comparing both levels we can assess such uncertainty for a given fault localization technique.

4.1.5 Results

Figures 4.1 and 4.2 show the cumulative results for different localization families and different ranking metrics of SBFL, respectively. Both EXAM and absolute rank (Hit@k, where k is on the horizontal axis) metrics are plotted for the best and worst case. The charts illustrate how large part of the program would a developer need to examine before finding a fault. The “step levels” in absolute rank plots are caused by different sizes of the programs because, for an unsuccessful prediction, we set the rank to be equal to the number of executed statements.

The very first observation in Figure 4.1 is that using likely invariants is not very successful. This is consistent with negative results of [32] which experimented with invariants on Siemens dataset as well. However, successes of [34, 33] promise that, when more sophisticated inference and filtering are implemented, this type of analysis might become useful. The results might be also skewed such that even when it was not effective in the sense of quantitative measurements, its detailed information could be valuable for the developer and lead to the successful end.

Comparing spectrum-based family represented by D* with probabilistic dependence technique gives us that the former is slightly better than the latter. This is especially interesting when their computational costs are contrasted (see Section 4.3 for details). However, the cause for this result might be, at least partially, our imperfect implementation. This is especially the case for data-flow analysis which is very primitive in our current state.

Figure 4.2 visualizes the effectiveness of different ranking metrics in SBFL family. Extreme cases are Wong1 and Overlap which follow almost identical line (therefore Overlap is not visible). Both metrics have enormous deviation between best and worst cases. The explanation is that they generate large

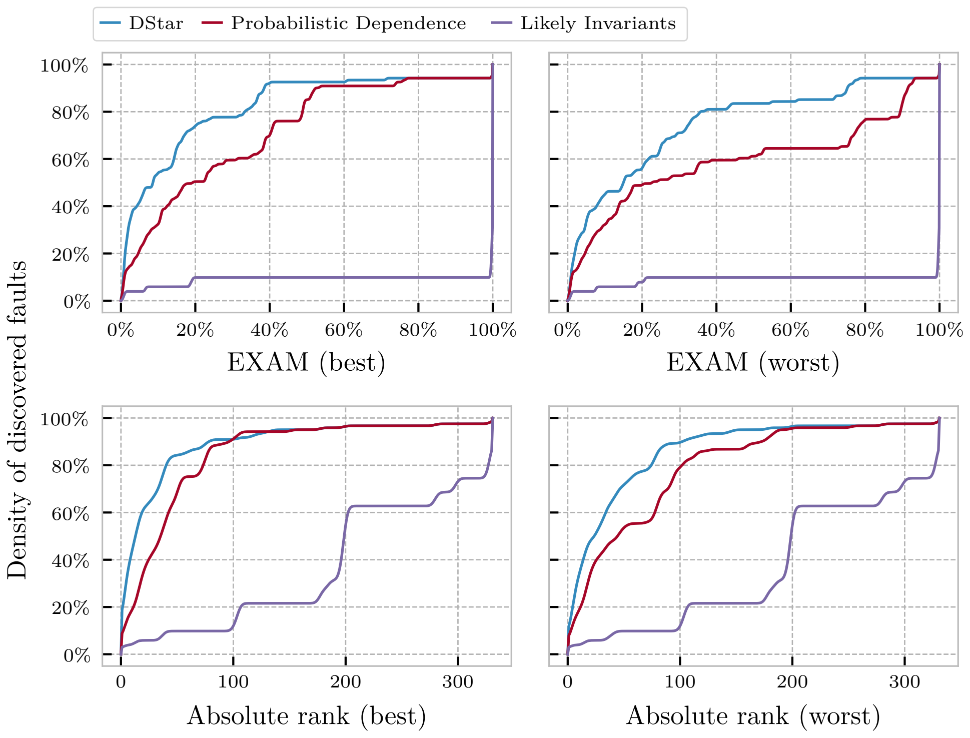


Figure 4.1: Localization effectiveness of different families. The lines represent how many bugs are found when inspecting given part of the program cumulatively.

clusters of statements which share the same suspiciousness score. The case of Overlap is consistent with findings in [10] where real faults are used. For this reason we consider them both rather ineffective.

Another pair of similarly behaving metrics is Zoltar and O^p. They are both more effective than the others and their best-worst range is reasonable. In [10] however, Zoltar is slightly worse than Ochiai and Tarantula which, together with D^*, form another group of similar metrics. This might indicate that Zoltar and O^p success is biased by evaluation on the Siemens dataset.

The numeric results are summarized in Table 4.2. When using O^p and Zoltar, one would be able to successfully locate more than half of the bugs after inspecting at most 10 statements in the best case, and almost 40% in the worst case.

From Hit@k results for probabilistic dependence we can conclude that when the technique is successful in discovering the fault, it is very confident about its prediction. On the other hand, if it is not the case, it is rather unsure as the deviation in its EXAM score suggests.

There is however no superior technique which would be always better than others. This fact is illustrated in Figure 4.3 visualization, where EXAM scores

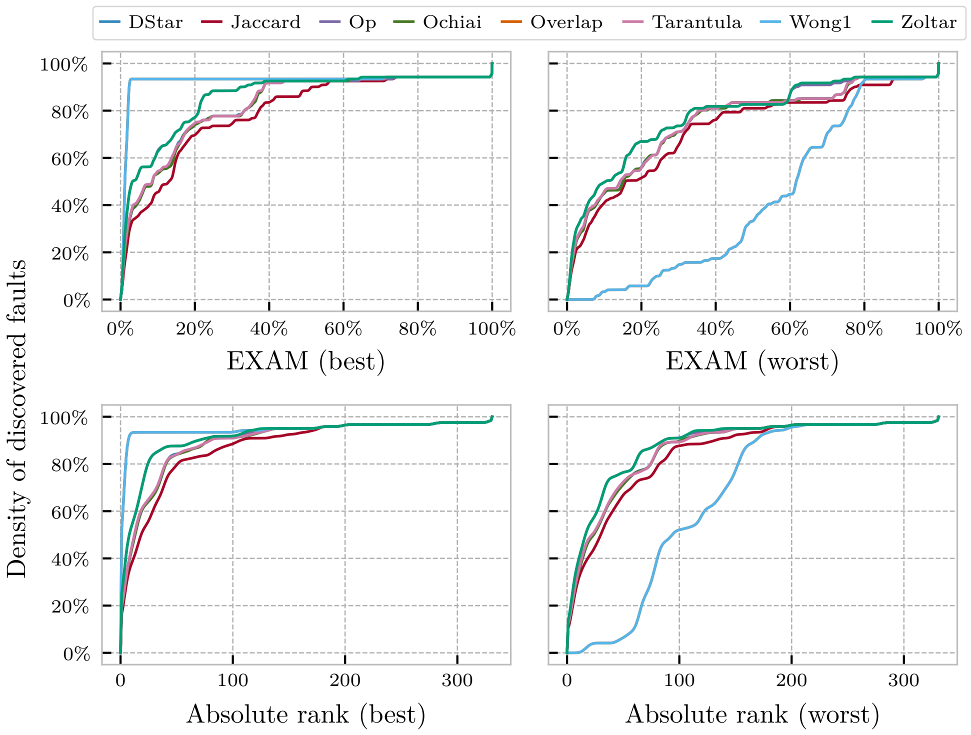


Figure 4.2: Localization effectiveness of different ranking metrics. The lines represent how many bugs are found when inspecting given part of the program cumulatively.

Table 4.2: Overall results. In case of EXAM score, the lower is better, while in case of Hit@k, the opposite is true. We highlighted techniques which were most effective in our evaluation.

Technique	EXAM		Hit@5		Hit@10	
D*	17.65	24.76	35.54	23.14	43.80	33.88
Jaccard	20.63	27.74	33.06	19.83	38.02	31.40
O^p	**14.40**	**22.10**	**48.76**	**28.93**	**55.37**	**38.84**
Ochiai	17.71	24.83	35.54	23.14	42.98	33.88
Overlap	7.48	57.86	93.39	0.00	93.39	0.00
Tarantula	17.52	24.64	35.54	23.97	44.63	34.71
Wong1	7.49	57.86	93.39	0.00	93.39	0.00
Zoltar	**14.29**	**21.98**	**48.76**	**28.93**	**55.37**	**38.84**
Prob. Dep.	27.87	40.39	14.05	13.22	18.18	18.18
Likely Invariants	91.09	91.14	3.92	3.92	3.92	3.92

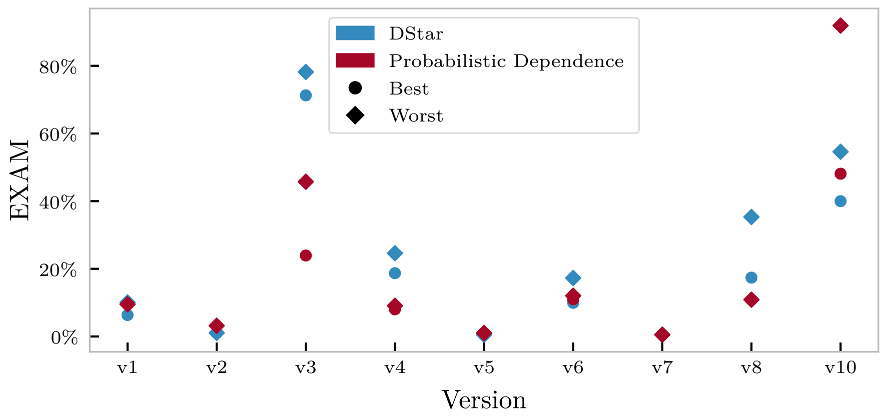


Figure 4.3: An example effectiveness “profile” of different families on *schedule2* program.

of spectrum-based and probabilistic dependence techniques are presented for different faulty versions of the same program. As can be seen, one is better than the other for some bugs while it is the opposite in other cases. There is therefore potential for results combination techniques, whose success is demonstrated in [47, 45].

Tabular comparison of our results and the numbers published in the original literature is given in Table 4.3. We compare Tarantula [69] with our implementation of SBFL with this metric and RankCP [13] which is the basis for our probabilistic dependence plugin. Two numbers for RankCP are given because the method is designed to work on a single failing test case originally and thus the results are shown for the most and the least successful test case. Our implementation utilizes all failing test cases by joining their scores and choosing the most suspicious prediction for each statement. Authors of [69] specifically mention that they use the worst rank in case of ties and hence we present the equivalent. This issue is not discussed in [13].

As can be seen, our implementation of Tarantula is comparable to the original one and the differences are caused by different faults determination methodology and statement coverage construction. On the other hand, compared to the results presented in the original publication, our implementation of probabilistic dependence method is inferior and suggests that improvements should be made.

We emphasize that the numbers are only of tentative nature because there are few aspects that make this direct comparison questionable. While the denominator in EXAM score formula (Equation (4.1)) is the number of executed statements for Tarantula and our results, in case of RankCP, it is the number of nodes in corresponding PPDG.

Although not extremely, the implementation details might cause differ-

Table 4.3: Comparison of the methods with their original literature. The table shows percentage of located faults for given EXAM range. Our results are marked with † symbol. RankCP is the name of fault localization algorithm used in [13] which we utilize in probabilistic dependence plugin. Original results for Tarantula are taken from [69], and for RankCP from [13].

EXAM	Tarantula	RankCP$_{best}$	RankCP$_{worst}$	Tar.†	Prob. Dep.†
0-1 %	13.93	41.94	17.74	14.05	9.09
1-10 %	41.80	31.45	27.42	31.40	23.14
10-20 %	5.74	13.71	25.81	10.74	16.53
20-30 %	9.84	2.42	4.84	14.88	4.13
30-40 %	8.20	2.42	4.84	9.92	6.61
40-50 %	7.38	5.65	8.06	2.48	1.65
50-60 %	0.82	1.61	2.42	0.83	3.31
60-70 %	0.82	0.00	5.65	0.83	0.00
70-80 %	4.10	0.80	2.42	9.09	12.40
80-90 %	7.38	0.00	0.81	0.00	7.44
90-100 %	0.00	0.00	0.00	5.79	15.70

ences in the statements count and form. For example, authors of Tarantula use Gcov coverage tool which monitors execution of source lines regardless of the syntactic structure.

Both [69] and (presumably) [13] use simply differing lines as the predictor for faulty statements. On the other hand, we follow more complex and precise methodology introduced by [10] (see Subsection 4.1.3 for details). We publish our labels in our public repository for reproducible evaluation.

All these imperfections accent the need for a general framework which would enable more fair and meaningful comparisons between individual techniques. To be such a framework is one of the Aardwolf's goals as it encourages extensibility while being the common denominator for program analysis and instrumentation and evaluation methodology.

4.2 Integration with Existing Projects

In this section, we describe how we integrated Aardwolf into two non-trivial, real-world software projects. Convenient integration is one of the goals of ours. The experience gained from this effort was valuable for improving this aspect of user-friendliness and the tool in general, and convinced us that it really has the potential to be incorporated into a project with a reasonable effort.

4.2.1 LibTIFF

The first software project in which we attempted to use Aardwolf is LibTIFF – a library providing support for the Tag Image File Format (TIFF), a widely used format for storing image data. We downloaded the sources from the official repository at `https://gitlab.com/libtiff/libtiff` in its most recent version at the time of writing the thesis. The project consists of more than 61,000 source lines of C code, and has around 140 tests. The test suite might be best characterized as system tests where each test calls whole programs in a sequence (LibTIFF consists of many small utility programs) [67].

The project uses multiple build systems, mainly CMake[23] and Autoconf[24]. We further describe configuration for Autoconf, but we expect that modifying the CMake build would be very similar.

First, a flag `--with-aardwolf` was introduced into `configure.ac` file, which serves as a macro-based template from which a real `configure` script is generated. If the flag is enabled, then the script checks if the selected compiler is Clang (as we use LLVM frontend) and if a dummy program can be compiled and linked with required compiler flags and Aardwolf libraries. The libraries need to be available either system-wide or inside the directory given by user as the argument. If any of these check fails, the whole configuration is terminated with error.

The biggest obstacle turned out to be libtool program which is used by LibTIFF's build process for compiling and linking the sources. Its purpose is to help with linking of shared libraries and one of its features is reorganizing the command line arguments to be sent to the compiler. In particular, it groups library linking flags together and puts them after the compilation flags. All arguments prefixed with `-l` are considered to be for linking libraries. However, this applies also to `-load` used in the pair with `-Xclang`. Obviously, when they are torn apart, it stops working. We solved this issue by patching the libtool script where we considered `-load` flag to be a special case.

To dump the name of a test to the trace file, a simple utility program distributed with Aardwolf is used. If it is given an argument, it adds it as the test name into the trace file. For simplicity, we added this call into the test driver script that is automatically generated by Autoconf. A much more clean and robust solution would be to use its custom test driver API, where a simple wrapper would log the test case name and then pass on the test execution to the default driver.

The last step is writing the `.aardwolf.yml` file. In the script, the project is configured to be built while enabling the Aardwolf using new flag we introduced, and setting the compiler to be Clang. Path to Aardwolf libraries is set using the variable passed into the script by the tool. Compilation of source files is triggered using `make` with compiler flags overridden with those for anal-

[23]`https://cmake.org/`

[24]`https://www.gnu.org/software/autoconf/`

ysis and instrumentation. Trace file is initialized by mentioned utility and the test suite is executed using `make check`. Finally, test results are obtained by extracting them from `.log` files using a one-line script. The whole process closely follows the standard test execution procedure as described in the project documentation. The only extensions are patching the libtool (might be fixed in a future version) and test driver (unnecessary if a custom driver is used), initialization of trace file using a single command, and extracting the test results using a simple one-line script.

As everything is activated conditionally only when `--with-aardwolf` is set, it does not interfere with the ordinary test suite execution. Therefore it does not affect the project development routines and conventions and only brings support for an automated fault localization if requested.

The process of integration fully utilizes the build systems used ordinarily in the project and only modifies certain bits of it. The biggest struggle happened to be issues with `-Xclang -load` in libtool. In the end, when the setup is done, to use Aardwolf one can just run the executable installed on their system and everything works out of the box. The performance implications for the test suite execution are discussed in Section 4.3.

4.2.2 Matplotlib

We chose matplotlib as the second project. It is a widely-used, comprehensive library for creating visualizations in Python[25]. The sources were downloaded from `https://github.com/matplotlib/matplotlib`. The library contains almost 70,000 source lines in Python. Note that there is also a couple of thousands of C++ code. One of the advantages of Aardwolf is that it can potentially support multi-language projects. For matplotlib in particular, a C++ frontend would need to be implemented. The test suite in the project is composed of more than 7,000 unit tests, each examining a very limited scope of functionality of the library.

The project has a Python test script which does few checks for assets and prepares dependencies for the tests execution, and then invokes pytest, a very popular testing framework for Python. We hook into the process of dependencies setup with installation of Aardwolf import hook for `matplotlib` and `mpl_toolkits` packages. Since the tests reside inside the sources directory tree and not in a separate one, we need to specify testing submodules to be ignored by the hook.

Test names and results logging is achieved through pytest hooks in which Aardwolf runtime functions are called. Matplotlib uses such hooks as well and there is a file which collects their definitions. We add our hooks to this file and import them with the others at appropriate initialization files (one for `matplotlib` and one for `mpl_toolkits`). In `.aardwolf.yml` script, we can

[25]In fact, we use matplotlib for graphs in this thesis.

then just execute matplotlib's test driver and the rest is done in initialization files as we described. Analysis and instrumentation is enabled only if specific environment variable is defined, which is set automatically when the script is run by the tool. If not defined, tests are executed as usual.

The project does not specify its development dependencies in any setup script, it only mentions them in the documentation. In a similar way, Aardwolf would need to be mentioned there too. As in the case of LibTIFF, when setup we just described is finished, using Aardwolf involves simply running its executable. Again, we discuss the performance in Section 4.3.

4.2.3 Conclusion

We were able to seamlessly integrate our tool into two non-trivial software projects, one written in C and one in Python. In both cases, Aardwolf was hooked into build and testing procedures used by the projects by default, and the process was simple. For LibTIFF, it meant writing seven-line script in `.aardwolf.yml` (from which three lines account to standard compilation and test suite execution process) plus list of plugins of choice, 40 lines for `--with-aardwolf` flag configuration including proper checks, and three-line patch of the test driver, excluding the patch for libtool which should not have been necessary. In case of matplotlib, it was one-line script running the test suite and all the machinery implementation in 30 lines of Python code inside the test configuration files.

After it is set up, developers can use our tool straightforwardly by running Aardwolf program installed on their computer.

4.3 Scalability

We now evaluate the scalability of Aardwolf on LibTIFF and matplotlib libraries from the previous section. Several metrics are measured in order to illustrate the overhead brought by the analysis and instrumentation on the runtime. Performance of individual Aardwolf stages is also benchmarked.

As mentioned in the introduction, scalability to larger software is an important requirement for any fault localization tool. Although we sacrifice some performance possibilities in our design for simpler implementation of Aardwolf extensions, the tool's computational costs should remain reasonable.

4.3.1 Runtime Overhead

First important aspect of using a fault localization tool is what overhead it brings to running the test suite. We are interested in three metrics here: execution time, peak memory consumption and the size of compiled binaries.

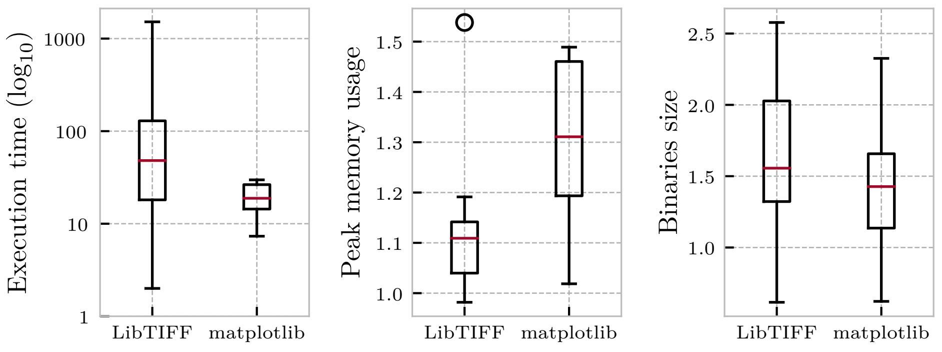


Figure 4.4: Time (in log scale), memory consumption and binary size increase factors when using Aardwolf for test suite execution of LibTIFF and matplotlib.

An overhead factor for every metric is calculated simply as

$$X = \frac{x_{aardwolf}}{x_{baseline}}.$$

Meaning of this factor is then for instance "when using Aardwolf, the peak memory consumption is X times higher".

Regarding the execution time, for LibTIFF we measure the runtime overhead for the test suite omitting the time spent in compilation[26]. "Compilation" and execution are tightly coupled in Python and cannot be conveniently split, hence this metric includes both of these factors in matplotlib case. Regarding the size of compiled binaries, we measure the disk size of `.o` files for LibTIFF and `.pyc` files for matplotlib. Note that in case of Python frontend, due to our usage of import hooks, no bytecode files are actually generated, and we must create them manually using Python internal API for analysis purposes. Running the entire test suite of matplotlib was beyond the capabilities of our hardware memory-wise as Aardwolf currently loads entire data into the operational memory. For this reason, we limited the test execution to roughly one third of the original test suite.

All three metrics for both projects are depicted in Figure 4.4 using box plots. They illustrate what is the average increase factor as well as the standard deviation and statistical outliers.

The slowdown is significant for both projects, more severely in LibTIFF case. Besides that, our benchmark compares the executions with disabled optimizations, as this is the requirement for our LLVM frontend at the moment. In ordinary execution, the LibTIFF test suite is however compiled with `-O2` optimizations, so the effective slowdown when using Aardwolf is even larger. The reason for the overhead is mainly performed I/O operations.

[26]From our measurements, the compilation is slowed down roughly by 10 %, which we consider negligible.

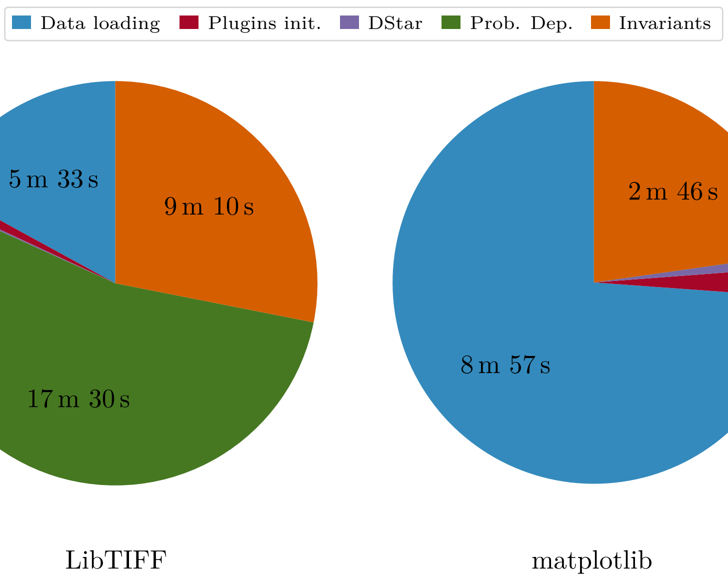


Figure 4.5: Approximate times spent in various stages of the localization process. The absolute values are averages of multiple runs.

The memory overhead for LibTIFF is reasonable while a little bit more notable in case of matplotlib. But we consider it as still acceptable. There is no significant source of memory overhead as all data are immediately dumped into the trace file. We might trade-off some of the memory usage for execution speedup, for example by storing the data temporarily in runtime's data structures and using file writes less frequently.

The overhead of binary size is not so important as there is no real cost in storing the files on the disk. It serves only as an interesting factor. On average, the compiled files are 1.5 times bigger when using Aardwolf for both projects. There are some cases when the file size was actually smaller, but we do not have a reasonable explanation for this phenomenon.

4.3.2 Aardwolf Scalability

The performance of Aardwolf analysis itself is discussed in this section. It should be noted that it is highly dependent on the test suite size and program behavior (which affects the number of encountered states in probabilistic dependence technique or invariants inference for likely invariants technique).

The execution times of various stages are presented in Figure 4.5. Due to incompleteness of the Python frontend, probabilistic dependence plugin was crashing on matplotlib data, and thus it is not depicted in this case. Recall that the times for matplotlib correspond to just a fraction of its entire test suite.

It can be seen that significant amount of time is spent in data loading stage. Probabilistic dependence analysis is the most costly but that is partly due to lack of optimizations. On the other hand, spectrum-based analysis is extremely lightweight and fast, taking only very small fraction of the entire run time. Invariants plugin is somewhere between.

We consider the absolute times as encouraging since the measurement was performed on two rather large software projects. Regarding this, it is important to repeat that no serious profiling and optimization effort was put into the implementation yet.

Chapter 5

Conclusion

5.1 Summary

This thesis presents a new automated tool *Aardwolf* for software fault localization. Its main goals are high extensibility and focus on user experience. Fault localization techniques which we implemented, or want to be able to implement, are briefly described, the proposed design of the tool is discussed and our current implementation is explained. In the end, the evaluation was performed, where we focused not only on the localization effectiveness, but on the convenience of integration with a real-world project and the scalability when using Aardwolf on these projects as well.

The architecture of our framework encourages researchers and engineers to integrate new methods into Aardwolf. Both scientists and users can benefit from this opportunity. Users will get the most advanced localization methods into their functional toolset, whereas researchers are provided with an environment where common and tedious implementation work is available so they can focus solely on what is unique in their approach. Moreover, a unified framework allows to quantitatively compare different techniques in a manner which is more fair and objective, improving reproducibility and trustworthiness of results in this field.

The tool is designed to be extensible on multiple axes, not only by new fault localization techniques. Its architecture allows to bring support for a new programming language by developing the corresponding frontend. By building a plugin into an editor or IDE, one can also deliver the fault localization features directly to the user's development environment. These aspects have the potential for wide adoption and represent a novel approach in fault localization tools development.

Currently, Aardwolf's results are presented in a terminal output with a source code context and suspiciousness explanation. The advantage of terminal output is that it is usable by all developers regardless of their development environment. Supported programming languages are C and Python. At the moment, we have implementations for three different fault localization techniques with plans for many others.

The tool is openly available at

`https://github.com/aardwolf-sfl/aardwolf`

5.2 Related Work

There are many studies about software fault localization. However, our objective was to design and develop a practical tool rather than a new SFL technique. We compile a list of tools available for software fault localization in Table 5.1. It is based on the summaries in [3, 70] as well as on our own research. Only solutions utilizing dynamic analysis for fault localization are considered, as this was the scope of this thesis. For instance, we do not include pure static analyzers or manual debugging helpers.

The final selection contains 13 tools. Seven of them are openly accessible on URLs given below the table, while there is no information on how to obtain the software in other cases, thus it is assumed that they are accessible solely via contacting its author.

The majority is based on program spectrum analysis, sometimes accompanied by an extension as in Apollo and Barinel. Prominent exceptions are delta debugging algorithm in DD and replay debugging for execution traces in DrDebug.

Best supported languages are Java (7 tools) and C (3 tools). Although there are explicit mentions about plans for extending language support in some works (e.g., [78, 70]), it can be hardly seen as a priority. There is quite a trend of implementing the tools as plugins into Eclipse IDE[34] with first-class support for the JUnit[35] testing framework for Java. These tight integrations however prevent expansion into other development environments and programming languages.

We did not notice efforts to make an extensible tool such that different methods can be implemented in one application. Instead, each solution focuses on the technique that their authors propose or on the spectrum-based family. In Flavs, there is a possibility to specify custom ranking metric as a textual expression which is parsed and used for the localization, but it still limits the extensibility to SBFL method.

Unfortunately, from openly accessible software only GZoltar seems to be still actively developed. Links to other works do not indicate any recent activity and in some cases the download links do not even work at the time of writing this thesis. This demonstrates the deficiency of the software fault localization ecosystem. The implication is that users have very few ways how they can benefit from the research in this area.

[34]`https://www.eclipse.org/eclipseide/`

[35]`https://junit.org/`

Table 5.1: Available tools for software fault localization.

Name	Description	Languages	Interface	Availability
Ample [71]	Call sequence spectrum	Java	Eclipse plugin	open[27]
Apollo [72]	Combination of Tarantula and symbolic execution	PHP, HTML	CLI	via author
Barinel [73]	Combination of SBFL and Bayesian reasoning	C	unknown	via author
DD [60]	Delta debugging toolset	Java	Eclipse plugin	open[28]
DrDebug [74]	Replay-based debugging with dynamic slicing	x86 executables	CLI, GUI	open[29]
Falcon [75]	Fault localization for concurrent bugs	Java	CLI	via author
Flavs[70]	SBFL tool	C#	Visual Studio plugin	via author
GZoltar [76]	Testing and debugging framework	Java	CLI, Eclipse plugin	open[30]
Holmes [77]	Statistical debugging tool	C	GUI	via author
HSFal [78]	Hybrid spectrum of slices	Java	unknown	via author
Jaguar [79]	SBFL tool	Java	GUI	open[31]
Tarantula [80]	SBFL tool	Java	CLI	open[32]
Zoltar [81]	SBFL tool	C	CLI	open[33]
Aardwolf	Fault localization framework	C, Python	CLI	open

[27] `https://www.st.cs.uni-saarland.de/ample/`
[28] `https://www.st.cs.uni-saarland.de/eclipse/`
[29] `https://software.intel.com/content/www/us/en/develop/articles/pintool-drdebug.html`
[30] `https://gzoltar.com/`
[31] `https://github.com/saeg/jaguar`
[32] `https://github.com/spideruci/Tarantula`
[33] `https://github.com/ncfxy/zoltar`

5.3 Further Development

We plan to extend Aardwolf mainly in the following aspects.

Auxiliary techniques described in Chapter 1 like test cases and program elements prioritization, results combination and supplementary information generation would fully demonstrate the capabilities of Aardwolf design and implementation. Moreover, additional fault localization techniques would further expand the variability of the tool. Currently available plugins could be augmented and improved, for example by introducing new types of spectra in SBFL, using different graphical models in probabilistic dependence plugin or adding more program invariants in likely invariants technique.

Regarding user adoption, more programming languages should be supported. One attractive option is Java, mainly because it is widely used, but also for the reason that advanced fault localization datasets exist for this language (e.g., [82]). An editor plugin implementation is desirable, partly for exploring the possibilities and difficulties of this area for the Aardwolf architecture. The results presentation can be improved on its own, by providing more explanations and more context.

More enhancements for user-friendliness can be brought to the configuration and platform compatibility. To be more developer-friendly, we need to polish and expand our APIs as well as extend the documentation such that the ways how to extend Aardwolf are convenient and understandable.

To improve user experience, more serious effort must be put into the optimizations of test suite execution overhead and memory usage of analysis. Test suite execution times is the domain for language frontends where instrumentation and runtime optimizations must be found and implemented. For Aardwolf itself, the major bottleneck is significant operational memory usage. We want to explore data structures serialization and deserialization to/from disk to avoid memory overloading with complete data. Another dimension for performance improvement is parallel execution, both for the test suite execution (which is currently limited by single trace file) and the analysis performed by Aardwolf (plugins in the same phase are completely independent).

Regarding architecture changes, the most feasible augmentations are providing support for semi-automated methods and algorithms that require a pre-training phase. The former requires the implementation of a client-server architecture model as is for example used in *Language Server Protocol*[36] where the server offers features like autocomplete for editors and IDEs. Support for model pre-training involves providing API for distinguishing between learning and deployment phases as well as model saving, and preferably a ready-made infrastructure with labeled datasets.

[36] `https://microsoft.github.io/language-server-protocol/`

Bibliography

[1] Myers, G. J.; Sandler, C.; et al. *The Art of Software Testing*. John Wiley & Sons, Inc., 2012, ISBN 978-1-118-03196-4.

[2] Britton, T.; Jeng, L.; et al. Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers. 2013.

[3] Wong, W. E.; Gao, R.; et al. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering*, 2016: pp. 707–740, doi: 10.1109/TSE.2016.2521368.

[4] Zakari, A.; Lee, S. P.; et al. Software fault localisation: a systematic mapping study. *IET Software*, 2019: pp. 60–74, doi:10.1049/iet-sen.2018.5137.

[5] Masri, W. Chapter Three - Automated Fault Localization: Advances and Challenges. Elsevier, 2015, pp. 103–156, doi:https://doi.org/10.1016/bs.adcom.2015.05.001. Available from: `http://www.sciencedirect.com/science/article/pii/S0065245815000339`

[6] Kochhar, P. S.; Xia, X.; et al. Practitioners' Expectations on Automated Fault Localization. *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016: pp. 165–176, doi: 10.1145/2931037.2931051.

[7] Parnin, C.; Orso, A. Are Automated Debugging Techniques Actually Helping Programmers? *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011.

[8] de Souza, H. A.; Chaim, M. L.; et al. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR*, volume abs/1607.04347, 2016.

[9] Ang, A.; Perez, A.; et al. Revisiting the Practical Use of Automated Software Fault Localization Techniques. *IEEE 28th International Symposium on Software Reliability Engineering Workshops*, 2017, doi:10.1109/ISSREW.2017.68.

[10] Pearson, S.; Campos, J.; et al. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, ISSN 1558-1225, pp. 609–620, doi:10.1109/ICSE.2017.62.

[11] Radatz, J. IEEE Standard Glossary of Software Engineering Terminology. Technical report, The Institute of Electrical and Electronics Engineers, 1990.

[12] Galin, D. *Software Quality Assurance: From Theory to Implementation.* Pearson Education Limited, 2004, ISBN 0201 70945 7.

[13] Baah, G. K.; Podgurski, A.; et al. The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis. *IEEE Transactions on Software Engineering*, 2010: pp. 528–545, doi:10.1109/TSE.2009.87.

[14] Yu, Y.; Jones, J. A.; et al. An Empirical Study of the Effects of Test-Suite Reduction on Fault Localization. In *Proceedings of the 30th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2008, ISBN 9781605580791, pp. 201–210, doi:10.1145/1368088.1368116. Available from: `https://doi.org/10.1145/1368088.1368116`

[15] Khedker, U.; Sanyal, A.; et al. *Data Flow Analysis: Theory and Practice.* CRC Press (Taylor and Francis Group), 2009, ISBN 9780849328800.

[16] Ferrante, J.; Ottenstein, K. J.; et al. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.*, volume 9, no. 3, 1987: pp. 319–349, ISSN 0164-0925, doi:10.1145/24039.24041. Available from: `https://doi.org/10.1145/24039.24041`

[17] Harrold, M. J.; Rothermel, G.; et al. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, volume 10, no. 3, 2000: pp. 171–194, doi:10.1002/1099-1689(200009)10:3<171::AID-STVR209>3.0.CO;2-J.

[18] Xie, X.; Kuo, F.-C.; et al. Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In *Search Based Software Engineering*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-39742-4, pp. 224–238.

[19] Santelices, R.; Jones, J. A.; et al. Lightweight fault-localization using multiple coverage types. In *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 56–66.

[20] Wong, W. E.; Debroy, V.; et al. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability*, 2014: pp. 290–308, doi:10.1109/TR.2013.2285319.

[21] Kochhar, P. S.; Xia, X.; et al. A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System. *IEEE International Conference on Software Quality, Reliability and Security*, 2017, doi:10.1109/QRS.2017.22.

[22] Naish, L.; Lee, H. J.; et al. Spectral debugging: How much better can we do? In *ACSC*, 2012.

[23] Xie, X.; Chen, T. Y.; et al. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization. *ACM Trans. Softw. Eng. Methodol.*, volume 22, no. 4, 2013, ISSN 1049-331X, doi:10.1145/2522920.2522924. Available from: `https://doi.org/10.1145/2522920.2522924`

[24] Naish, L.; Lee, H. J.; et al. A Model for Spectra-Based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.*, volume 20, no. 3, 2011, ISSN 1049-331X, doi:10.1145/2000791.2000795. Available from: `https://doi.org/10.1145/2000791.2000795`

[25] Baah, G. K. *Statistical Causal Analysis for Fault Localization.* Dissertation thesis, Georgia Institute of Technology December, 2012.

[26] Steimann, F.; Frenkel, M.; et al. Threats to the Validity and Value of Empirical Assessments of the Accuracy of Coverage-Based Fault Locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, 2013, ISBN 9781450321594, pp. 314–324, doi:10.1145/2483760.2483767. Available from: `https://doi.org/10.1145/2483760.2483767`

[27] Denmat, T.; Ducassé, M.; et al. Data Mining and Cross-Checking of Execution Traces: A Re-Interpretation of Jones, Harrold and Stasko Test Information. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2005, ISBN 1581139934, pp. 396–399, doi:10.1145/1101908.1101979. Available from: `https://doi.org/10.1145/1101908.1101979`

[28] Campos, E. C.; de Almeida Maia, M. Common Bug-Fix Patterns: A Large-Scale Observational Study. *ACM/IEEE International Symposium*

on Empirical Software Engineering and Measurement, 2017: pp. 404–413, doi:10.1109/ESEM.2017.55.

[29] Yu, X.; Liu, J.; et al. Bayesian Network Based Program Dependence Graph for Fault Localization. *IEEE International Symposium on Software Reliability Engineering Workshops*, 2016, doi:10.1109/ISSREW.2016.35.

[30] Gong, D.; Su, X.; et al. State dependency probabilistic model for fault localization. *Information and Software Technology*, 2015: pp. 430–445, doi: 10.1016/j.infsof.2014.05.022. Available from: `https://doi.org/10.1016/j.infsof.2014.05.022`

[31] Ernst, M. D. *Dynamically Discovering Likely Program Invariants.* Dissertation thesis, University of Washington, 2000.

[32] Pytlik, B.; Renieris, M.; et al. Automated Fault Localization Using Potential Invariants. *International Workshop on Automated and Algorithmic Debugging*, 2003.

[33] Hangal, S.; Lam, M. S. Tracking down software bugs using automatic anomaly detection. *Proceedings of the 24th International Conference on Software Engineering*, 2002: pp. 291–301, doi:10.1145/581376.581377.

[34] Sahoo, S. K.; Criswell, J.; et al. Using Likely Invariants for Automated Software Fault Localization. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, 2013.

[35] Alipour, M. A.; Groce, A. Extended Program Invariants: Applications in Testing and Fault Localization. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA 2012, Association for Computing Machinery, 2012, ISBN 9781450314558, pp. 7–11, doi: 10.1145/2338966.2336799. Available from: `https://doi.org/10.1145/2338966.2336799`

[36] Guo, L.; Roychoudhury, A.; et al. Accurately Choosing Execution Runs for Software Fault Localization. In *Proceedings of the 15th International Conference on Compiler Construction*, Berlin, Heidelberg: Springer-Verlag, 2006, ISBN 354033050X, pp. 80–95, doi:10.1007/11688839_7. Available from: `https://doi.org/10.1007/11688839_7`

[37] Hao, D.; Xie, T.; et al. Test input reduction for result inspection to facilitate fault localization. *Automated Software Engineering*, volume 17, 2009: pp. 5–31.

[38] Yoo, S.; Harman, M.; et al. Fault Localization Prioritization: Comparing Information-Theoretic and Coverage-Based Approaches. *ACM Transactions on Software Engineering and Methodology*, volume 22, no. 3,

2013, ISSN 1049-331X, doi:10.1145/2491509.2491513. Available from: `https://doi.org/10.1145/2491509.2491513`

[39] Zhang, X.; Towey, D.; et al. Using Partition Information to Prioritize Test Cases for Fault Localization. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, 2015, pp. 121–126.

[40] Xu, X.; Debroy, V.; et al. Ties within Fault Localization rankings: Exposing and Addressing the Problem. *International Journal of Software Engineering and Knowledge Engineering*, volume 21, 2011: pp. 803–827.

[41] Xie, X.; Chen, T. Y.; et al. Isolating Suspiciousness from Spectrum-Based Fault Localization Techniques. In *Proceedings of the 2010 10th International Conference on Quality Software*, USA: IEEE Computer Society, 2010, ISBN 9780769541310, pp. 385–392, doi:10.1109/QSIC.2010.45. Available from: `https://doi.org/10.1109/QSIC.2010.45`

[42] Jin, W.; Orso, A. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, New York, NY, USA: Association for Computing Machinery, 2013, ISBN 9781450321594, pp. 213–223, doi:10.1145/2483760.2483763. Available from: `https://doi.org/10.1145/2483760.2483763`

[43] Lucia, L.; Lo, D.; et al. Extended Comprehensive Study of Association Measures for Fault Localization. *Journal of software: Evolution and Process*, volume 26, no. 2, 2014: pp. 172–219, ISSN 2047-7473, doi: 10.1002/smr.1616. Available from: `https://doi.org/10.1002/smr.1616`

[44] Shaowei Wang; Lo, D.; et al. Search-based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 556–559.

[45] Lucia; Lo, D.; et al. Fusion Fault Localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Association for Computing Machinery, 2014, ISBN 9781450330138, pp. 127–138, doi:10.1145/2642937.2642983. Available from: `https://doi.org/10.1145/2642937.2642983`

[46] Xuan, J.; Monperrus, M. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 191–200.

[47] Zou, D.; Liang, J.; et al. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering*, 2019, ISSN 2326-3881, doi:10.1109/TSE.2019.2892102.

[48] Le, T.-D. B.; Lo, D.; et al. Should I Follow This Fault Localization Tool's Output? *Empirical Software Engineering*, 2015: pp. 1237–1274, ISSN 1382-3256, doi:10.1007/s10664-014-9349-1. Available from: `https://doi.org/10.1007/s10664-014-9349-1`

[49] Neelofar. *Spectrum-based Fault Localization Using Machine Learning.* Dissertation thesis, University of Melbourne, 2017.

[50] Liblit, B.; Naik, M.; et al. Scalable Statistical Bug Isolation. *ACM SIGPLAN Notices*, volume 40, no. 6, 2005: pp. 15–26, ISSN 0362-1340, doi: 10.1145/1064978.1065014. Available from: `https://doi.org/10.1145/1064978.1065014`

[51] Liu, C.; Fei, L.; et al. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Transactions on Software Engineering*, volume 32, no. 10, 2006: pp. 831–848, ISSN 0098-5589, doi:10.1109/TSE.2006.105. Available from: `https://doi.org/10.1109/TSE.2006.105`

[52] Wong, W. E.; Debroy, V.; et al. Towards Better Fault Localization: A Crosstab-Based Statistical Approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, volume 42, 2012: pp. 378–396.

[53] You, Z.; Qin, Z.; et al. Statistical fault localization using execution sequence. *2012 International Conference on Machine Learning and Cybernetics*, volume 3, 2012: pp. 899–905.

[54] Feyzi, F.; Parsa, S. Inforence: effective fault localization based on information-theoretic analysis and statistical causal inference. *Frontiers of Computer Science*, volume 13, 2017: pp. 735–759.

[55] Landsberg, D.; Barr, E. T. Doric: Foundations for Statistical Fault Localisation. *CoRR*, volume abs/1810.00798, 2018, 1810.00798. Available from: `http://arxiv.org/abs/1810.00798`

[56] Wong, W. E.; Qi, Y. BP Neural Network-Based Effective Fault Localization. *International Journal of Software Engineering and Knowledge Engineering*, volume 19, no. 04, 2009: pp. 573–597, doi:10.1142/S021819400900426X, `https://doi.org/10.1142/S021819400900426X`. Available from: `https://doi.org/10.1142/S021819400900426X`

[57] Wong, W. E.; Debroy, V.; et al. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Transactions on Reliability*, volume 61, 2012: pp. 149–169.

[58] Modi, V.; Roy, S.; et al. Exploring Program Phases for Statistical Bug Localization. In *Proceedings of the 11th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, New

York, NY, USA: Association for Computing Machinery, 2013, ISBN 9781450321280, pp. 33–40, doi:10.1145/2462029.2462034. Available from: `https://doi.org/10.1145/2462029.2462034`

[59] Zeller, A.; Hildebrandt, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, volume 28, no. 2, 2002: pp. 183–200, ISSN 0098-5589, doi:10.1109/32.988498. Available from: `https://doi.org/10.1109/32.988498`

[60] Zeller, A. Isolating Cause-Effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2002, ISBN 1581135149, pp. 1–10, doi:10.1145/587051.587053. Available from: `https://doi.org/10.1145/587051.587053`

[61] Mayer, W.; Stumptner, M.; et al. Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models. 2002, pp. 417–421.

[62] Mayer, W.; Stumptner, M. Abstract Interpretation of Programs for Model-Based Debugging. In *IJCAI*, 2007.

[63] Könighofer, R.; Bloem, R. Automated error localization and correction for imperative programs. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011, pp. 91–100.

[64] Groce, A.; Clarke, E. M. Error explanation and fault localization with distance metrics. 2005.

[65] Gong, L.; Lo, D.; et al. Interactive fault localization leveraging simple user feedback. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, ISSN 1063-6773, pp. 67–76, doi:10.1109/ICSM.2012.6405255.

[66] Zhang, S.; Zhang, C. Software Bug Localization with Markov Logic. In *Companion Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA: Association for Computing Machinery, 2014, ISBN 9781450327688, pp. 424–427, doi:10.1145/2591062.2591099. Available from: `https://doi.org/10.1145/2591062.2591099`

[67] Le Goues, C.; Holtschulte, N.; et al. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering (TSE)*, volume 41, no. 12, December 2015: pp. 1236–1256, ISSN 0098-5589, doi:10.1109/TSE.2015.2454513.

[68] Hutchins, M.; Foster, H.; et al. Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. In *Proceedings of the 16th International Conference on Software Engineering*, IEEE Computer Society Press, 1994, ISBN 081865855X, pp. 191–200.

[69] Jones, J. A.; Harrold, M. J. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, Association for Computing Machinery, 2005, ISBN 1581139934, pp. 273–282, doi:10.1145/1101908.1101949. Available from: `https://doi.org/10.1145/1101908.1101949`

[70] Wang, N.; Zheng, Z.; et al. FLAVS: A Fault Localization Add-in for Visual Studio. In *Proceedings of the First International Workshop on Complex FaUlts and Failures in LargE Software Systems*, IEEE Press, 2015, pp. 1–6.

[71] Dallmeier, V.; Lindig, C.; et al. Lightweight Bug Localization with AMPLE. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, New York, NY, USA: Association for Computing Machinery, 2005, ISBN 1595930507, pp. 99–104, doi:10.1145/1085130.1085143. Available from: `https://doi.org/10.1145/1085130.1085143`

[72] Artzi, S.; Dolby, J.; et al. Practical Fault Localization for Dynamic Web Applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA: Association for Computing Machinery, 2010, ISBN 9781605587196, pp. 265–274, doi:10.1145/1806799.1806840. Available from: `https://doi.org/10.1145/1806799.1806840`

[73] Abreu, R.; Zoeteweij, P.; et al. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, 2009, pp. 88–99.

[74] Wang, Y.; Patil, H.; et al. DrDebug: Deterministic Replay based Cyclic Debugging with Dynamic Slicing. In *CGO '14*, 2014.

[75] Park, S.; Vuduc, R. W.; et al. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, New York, NY, USA: Association for Computing Machinery, 2010, ISBN 9781605587196, pp. 245–254, doi:10.1145/1806799.1806838. Available from: `https://doi.org/10.1145/1806799.1806838`

[76] Riboira, A.; Abreu, R. The GZoltar Project: A Graphical Debugger Interface. In *Proceedings of the 5th International Academic and Industrial*

Conference on Testing - Practice and Research Techniques, Berlin, Heidelberg: Springer-Verlag, 2010, ISBN 3642155847, pp. 215–218.

[77] Chilimbi, T. M.; Liblit, B.; et al. HOLMES: Effective statistical debugging via efficient path profiling. *2009 IEEE 31st International Conference on Software Engineering*, 2009: pp. 34–44.

[78] Ju, X.; Jiang, S.; et al. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. *Journal of Systems and Software*, volume 90, 2014: pp. 3–17, ISSN 0164-1212, doi:https://doi.org/10.1016/j.jss.2013.11.1109. Available from: `http://www.sciencedirect.com/science/article/pii/S0164121213002823`

[79] Ribeiro, H. L.; de Souza, H. A.; et al. Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018: pp. 404–409.

[80] Jones, J. A.; Harrold, M. J.; et al. Visualization for Fault Localization. In *Proceedings of the Workshop on Software Visualization (SoftVis), 23rd International Conference on Software Engineering*, May 2001, pp. 71–75.

[81] Janssen, T.; Abreu, R.; et al. Zoltar: a spectrum-based fault localization tool. In *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-681-6, pp. 23–30, doi:http://doi.acm.org/10.1145/1596495.1596502.

[82] Just, R.; Jalali, D.; et al. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA*, 2014, pp. 437–440.

Appendix A

Experimental Evaluation Details

A.1 Faulty Elements Determination

In this section we show and comment some complicated cases regarding the determination of faulty elements.

- Switch branching construct is a common syntax sugar for chained `else if` statements. Consider the example in Figure A.1a. We consider both `switch` and `case` lines as faulty because `case 32` is equivalent to `else if (state == 32)` which would be blamed if `if/else` construct would be used.
- Predicates which span multiple lines. Consider the example in Figure A.1b. We consider the whole span of the predicate as faulty even though the second line is technically to be blamed only. The rationale is that when the expression on the second line is incorrect, it is also true that the entire predicate is faulty.
- A non-unusual developer practice is to assign the result of a predicate to a temporary variable and then use this variable in the branching statement. Consider the example in Figure A.1c. The variable assignment on the first line is faulty without a doubt. It is, however, questionable if we should also blame the `if` statement where the predicate is used. We chose not to.
- When values of non-mutable constants are incorrect, apart from blaming the assignment line, we also consider all statements that use that particular constant as faulty. The rationale is that a constant is essentially a named literal value, and so the statement is faulty because is using an incorrect literal value. An example is depicted in Figure A.1d.
- In case of fault of omission, the inserted statement could be sometimes added at a different location and it would be equally valid choice. See

```
 switch (state) {
     case 16:
         return XOR;
-    case 32:
-        return EQUALGREATER;
 }
```

(a)

```
 else if ((isalnum(src[*i - 1]))
-    && (src[*i - 1] < src[*i + 1]))
+    && (src[*i - 1] <= src[*i + 1]))
```

(b)

```
-int status = put_end(prio, new_proc);
+int status = put_end(prio, old_proc);
 if (status) {
     return status;
 }
```

(c)

```
-#define MAXLEN 25
+#define MAXLEN 256

 for (int i = 0; i < MAXLEN; i++) {
     // ...
 }
```

(d)

```
 switch (state)
 {
     case 23: return QUOTE;
     case 25: return COMMA;
+    case 32: return EQUALGREATER;
     default: return ERROR;
 }
```

(e)

Figure A.1: Several examples of complicated cases when determining faulty statements. Orange lines indicate what we consider to be faulty elements.

Figure A.1e for an example. Semantically speaking, the new statement could be inserted anywhere in between the third and sixth line. The first line in this example is considered faulty for the same reason as in Figure A.1a.

We applied this methodology for making labels for the Siemens dataset.

A.2 Siemens Dataset

All programs in the Siemens dataset were compiled using (omitting Aardwolf-related flags)

```
clang -g -O0 -Wno-return-type -std=c89 <program>.c -lm
```

Table A.1: Hardware configuration used for scalability benchmarks.

Parameter	Value
CPU Model	Intel® Core™ i5-4200M
CPU Clock Frequency	2.50 GHz
Architecture	x86-64
RAM Memory	8 GiB SODIMM DDR3 Synchronous 1600 MHz
Disk	Samsung SSD 840
Filesystem	ext4
Linux kernel	5.5.5-arch1-1

Debug information (`-g`) and disabled optimizations (`-O0`) are required for Aardwolf to function properly. The standard of the language is set to C89 because programs use implicit return types, which was allowed in C89, but forbidden in later versions of the standard. Some programs do not specify the return type in their `void` functions. The default, implicit return type in C is `int` and thus the compiler raises the error that non-void function does not return a value. We ignore this error using `-Wno-return-type` and assume that usage of these functions is correct, that is, their return values are not used, as it would cause undefined behavior according to the standard. There are also several warnings even in the default mode (i.e., without `-Wall`, `-Wextra` or similar options that turn on extra warnings) but we ignore them.

A.3 Data Collection for Scalability Experiment

All time and memory consumption data for test suite execution were obtained by using GNU time utility with the format `%e` (elapsed real time in seconds) and `%M` (maximum resident set size of the process during its lifetime in kilobytes). In case of LibTIFF, values were collected on per-test basis, while in case of matplotlib, we grouped individual tests based on test file which they are defined in (limited to 30 out of 86 test files). Running time of Aardwolf components was measured by the internal timer. Binaries sizes (`.o` files for LibTIFF and `.pyc` files for matplotlib) were computed with `du` utility from GNU coreutils.

A.4 Hardware Configuration

All benchmarks presented in Chapter 4 were performed on a computer whose relevant hardware details are listed in Table A.1.

Appendix B

Acronyms

AST Abstract Syntax Tree. 56

CFG Control Flow Graph. 12, 56

FL (Software) Fault Localization. 7

IDE Integrated Development Environment. 2, 4, 52, 58, 77, 80

PDG Program Dependence Graph. 12, 13, 18, 19, 22

PPDG Probabilistic Program Dependence Graph. 20, 21, 69

SBFL Spectrum-Based Fault Localization. 7, 14–16, 22, 66, 69, 78–80

SFL Software Fault Localization. 1, 5, 78

SLOC Source Lines of Code. 65

Appendix C

Contents of enclosed CD

```
README.txt ........................ the file with CD contents description
├─ code ........................... the directory of program source codes
├─ text ................. the directory of LaTeX source codes of the thesis
└─ thesis.pdf ........................... the thesis text in PDF format
```