# Static Error Trace Analysis Using Aberrant Trace Elements

**Muhammad Numair Mansur**

Chair of Software Engineering
Department of Computer Science
Albert-Ludwigs-University Freiburg, Germany

**First Examiner**: Prof. Dr. Andreas Podelski
**Second Examiner**: Prof. Dr. Peter Thiemann
**Supervision**: Christian Schilling, Dr. Matthias Heizmann

A thesis submitted for the Degree of Master of Science in
Albert-Ludwigs-University Freiburg
- 2018 -

*Jumping from failure to failure with undiminished enthusiasm is the big secret to success.*

Savas Dimopoulos

# Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, date

Signature

# Acknowledgment

I would like to thank my supervisors, Christian Schilling and Dr. Matthias Heizmann. Central aspects of this work are greatly influenced by their ideas and contributions. Without their patience, guidance and constant support, this work would not have been possible. I consider myself really lucky to have found them during my studies and would always cherish the time I spent with them. The love and passion they have for their work is a huge inspiration for me. Thank you.

# Abstract

Software model checkers are powerful tools for the verification of computer programs. When they find a violation, the typical way to communicate with the user is the output of an error trace – a sequence of statements through the program that, when executed, leads to an error. The ability to generate an error trace upon detecting a failing property is widely regarded as one of the major advantages of model checking in software verification. The diagnostic value of an error trace in localizing faults and other hidden issues in the program is undeniable. However, in practice, error traces can very quickly become prohibitively long and easily exceed the human capacity of analysis. We, therefore, need automatic trace analysis techniques if we wish to extract any value from an error trace.

This thesis presents one such technique based on the identification of statements that can single-handedly make the trace infeasible. We call such statements *aberrant*. Aberrant statements allow us to analyze the error trace from a new and quite useful perspective. In this thesis, we show that one of the applications of aberrance is in fault localization. The precise definition of aberrance not only allow us to locate faulty statements in a feasible error trace but also gives us a hint on how to fix the error. Another application of aberrant statements is in the detection of the presence of unvalidated inputs in a program, which is considered a typical security vulnerability. We also use aberrant statements to suppress and later rank error warnings. A large number of error warnings produced by static verification tools is a major hurdle in their adoption for debugging by average developers. Ranking error warning will, therefore, be a step in the right direction. We also discuss the use of aberrant statements in increasing the output precision of software model checkers.

We have implemented our algorithm to compute aberrant statements in the program analysis framework Ultimate, which is publicly available. The implementation allows us to test the above-mentioned applications on error traces from real-world programs.



### Keywords

Program Analysis, Error Trace, Fault Localization, Security, Angelic Verification

# Deutsche Zusammenfassung

Software Model Checker sind mächtige Werkzeuge für die Überprüfung von Computer Programmen. Finden diese einen Fehler, so geben sie dem Benutzer typischerweise ein Fehler-Trace zurück. Ein Fehler-Trace ist eine Reihe von Anweisungen, die zu einem Fehler führen, wenn sie ausgeführt werden. Die Fähigkeit einen Fehler-Trace zu erstellen, wenn ein Fehler auftritt wird weithin als einer der Hauptvorteile der Modellprüfung in Software Verifikation angesehen. Der Diagnosewert eines Fehler-Traces, bei der Lokalisierung von Fehlern und anderen versteckten Problemen in dem Program, ist unbestreitbar. In der Praxis können Fehler-Traces jedoch sehr schnell zu lang werden und die menschliche Analysekapazität leicht übersteigen. Um irgendeinen Wert aus einem Fehler-Trace zu extrahieren, benötigen wir automatische Analysetechniken.

Diese Thesis stellt eine solche Technik, die auf der Identifizierung von Anweisungen basiert, die die Protokollierung ganz alleine unausführbar machen, vor. Wir bezeichnen solche Anweisungen als aberrant. Aberrante Aussagen ermöglichen uns Fehler-Traces von einer neuen und praktischen Perspektive zu betrachten. In dieser Thesis zeigen wir, dass Fehlerlokalisierung eine der Anwendungen von Aberrance ist. Die präzise Definition von Aberrance erlaubt uns nicht nur die Fehlerlokalisierung in einem Fehler-Trace, sondern gibt uns auch einen Hinweis diesen Fehler zu beheben. Eine weitere Anwendung von aberranten Anweisungen besteht in der Erkennung von nicht validierten Eingaben in einem Programm, welche als typische Sicherheitslücke betrachtet werden. Wir verwenden auch aberrante Anweisungen um Fehlerwarnungen zu unterdrücken und später einzuordnen. Eine große Anzahl von Fehlerwarnungen, die von statischen Verifikationswerkzeugen erzeugt werden, stellen für den durchschnittlichen Entwickler beim Debugging eine große Hürde dar. Wir diskutieren auch die Verwendung von aberranten Anweisungen bei der Erhöhung der Ausgabegenauigkeit von Software Model Checkern.

Wir haben unseren Algorithmus, der aberrante Anweisungen berechnet, im Analyse Framework Ultimate implementiert. Dieser ist öffentlich verfügbar. Die Implementierung ermöglicht es uns die oben genannten Anwendungen auf Fehler-Traces von realen Programmen zu testen.

# Contents

# Chapter 1

# Introduction

As we advance into a future where autonomous vehicles move us around, cryptocurrencies gain an increasing amount of relevance in our financial system and we vote our representatives electronically, the importance of correct, robust, and secure software is becoming increasingly vital for the smooth functioning of our society. A report published last year by the insurance company Lloyd's claims that a single major global cyber attack can incur a loss of as much as $50 billion[1], a figure on par with a catastrophic natural disaster. In 2016, cybercrime cost the global economy over $450 billion, over 2 billion personal records were stolen and in the U.S. alone, over 100 million Americans had their medical records stolen[2]. And yet, developing robust and secure software still remains a challenge in 2018.

Just as in the past, today a developer's basic workflow consists of writing, testing and debugging program code. All of these steps have been greatly improved by (semi- or fully) automatic tools and algorithms developed by the research community. And yet, debugging software still takes a long time in this process, a major amount of which is usually dedicated to localizing faulty statements, i.e., the cause of the error.

Software model checkers are powerful tools for the verification of computer programs [1]. They function by exhaustively exploring the reachable state space of the model of a program. When a violation is detected, the typical way to communicate with the user is the output of an error trace – a sequence of statements through the program that, when executed, leads to an error. The diagnostic value of an error trace in understanding the cause of the error is undeniable. The usefulness of an error trace is directly proportional to our ability to effectively analyze the information present in it. However, in practice, an error trace can contain hundreds of statements and can easily become very tedious and time-consuming to analyze manually. This can happen for instance if the program is just too large or if the program has a *deep* bug, i.e., a bug that only manifests itself after a lot of loop unwindings. The need for automatic trace analysis techniques is, therefore, quiet obvious and a topic of research for

[1] https://www.cnbc.com/2017/07/17/global-cyberattack-could-spur-53-billion-in-losses-lloyds-of-london.html
[2] https://www.cnbc.com/2017/02/07/cybercrime-costs-the-global-economy-450-billion-ceo.html

many years now [2–6].

In this work, we consider statements that can single-handedly make a trace infeasible. We believe that such statements can provide us with an ability to analyze the trace from a different and quite useful perspective. We call such statements *aberrant*. A statement in a trace is aberrant if (a) it assigns a value to a variable $x$ and (b) there exists a different value that, when assigned to $x$ at this position instead, makes the trace infeasible. In Section 3 we formally define an aberrant statement in a trace and present an algorithm that statically computes the position of all aberrant statements.

To demonstrate how aberrant trace elements can be used to analyze an error trace, we present four applications in Section 4. The first application is in fault localization, where we discuss how aberrant statements can locate potentially faulty statements in an error trace. Given the precise definition of aberrance, the knowledge of which statements in the error trace are aberrant not only helps the programmer in locating the position where the bug might reside, but also suggests what can be done to fix it. Of course, localizing the fault in a program with a single error trace does have some limitations which become apparent when the trace passes through a branch (`then` part of an `if-then-else` branch). We also discuss how can we deal with such cases to improve the precision.

In the second application, we use aberrant statements to identify one of the subclasses of software security vulnerabilities known as *unvalidated inputs*. This security vulnerability exists when not all possible input values are checked to make sure if they are valid. Any input received by a program from an untrusted source (e.g., another user or a network) is a weak link when it comes to the security of the software. Hackers try to look for every source of input to the program and exploit it by passing input values that might cause the program to misbehave. Unvalidated-input exploits have been known to cause serious damage in real world software including data theft and corruption of hard disks. We present an example where an unvalidated input triggers a buffer overflow and show how we can use aberrant statements to detect the presence of unvalidated inputs in a program. We apply our approach on 3 real world examples from the Linux device drivers. These examples had unvalidated inputs which introduced vulnerabilities like an unavoidable doublelock in the Unix File System (UFS) and Aeroflex Gaisler device driver. Since the vulnerability could be exploited by an external input value, it was classified as a security vulnerability and was immediately fixed when found. We apply our algorithm on simplified programs, which model the problem with the unnecessary code removed. Given an error trace for the simplified programs, using our technique, we were able to confirm the existence of an unvalidated input in all the three programs.

Next, we discuss the application of aberrant statements to suppress uninteresting warnings in the context of open programs with an unconstrained environment. Open programs expose a set of external libraries or API methods. In the absence of precise environment specifications, static program verifiers tend to be conservative (over-approximate) and therefore generate a large number of warnings, all of which might not be useful to the programmer from a debugging perspective. These large number of warnings still hamper the adoption of

static verifiers for program debugging by an average software developer. We also present a statistical technique based on the number of aberrant statements in error traces to rank error warnings.

The fourth application considers error traces that have been overapproximated during the analysis by a software model checker. Accordingly, the feasibility status is unknown. If, however, all the overapproximated statements in the error trace are aberrant, we can conclude that the original trace is indeed feasible.

## 1.1 Contributions

In this thesis, we present an efficient technique to analyze an error trace and present it's application in fault localization, software security, verification of open programs and in increasing the precision of software model checkers. We also extend our basic approach to deal with programs with branches. Our technique does not require additional successful or failing executions other then the given error trace. Secondly, it does not use expensive model checking or constraint solving algorithms. We have implemented our algorithm in Ultimate Automizer, which is publicly available. To demonstrate the effectiveness of using our approach in fault localization and the detection of unvalidated input, we test our implementation on real world examples.

## 1.2 Outline

In Chapter 3, we formally define aberrant trace elements and present an algorithm to find all such trace elements in a trace. In Chapter 4, present four applications where aberrance can be used in practice. In Section 4.1, we show how aberrant trace elements can be used in fault localization. In Section 4.2, we use aberrance to detect the presence of unvalidated inputs in a program. In Section 4.3 , we present a technique to suppress and later rank uninteresting error warnings. In Section 4.4, we discuss how can aberrant trace elements be used to increase the precision of software model checkers. In Chapter 5, we discuss how our open source implementation can be used in Ultimate Automizer for programs written in C and Boogie.

## 1.3 Foundational Work For This Thesis

The work I did during my masters project forms the basis for this thesis. During my masters project, I along with my supervisors worked on finding statements in an error trace that "played a role in the error". Back then, we named them *relevant statements* and focused exclusively on fault localization. We also worked on the basic implementation in Ultimate Automizer. During the thesis we re-characterized the notion of *relevance* to *aberrance* and worked on extensions like e.g., dealing with error traces with branches and applications in security

and verification of open programs. The implementation, of course, was also heavily extended and tested thoroughly. We also evaluated our techniques on real world benchmarks to demonstrate their applicability.

# Chapter 2

# Preliminaries

**Program.** We follow the notation described in [7] and [1] and represent programs by their *control flow graph* (CFG) over a set of variables $V$. A CFG provides us with a simple abstraction for programs, allowing us not to worry about the syntax and semantics of programming languages. A node in a CFG corresponds to a program location and each edge is labeled with a statement. The statements are taken from a finite set $\Sigma$ called the *alphabet*. Formally, a CFG is a graph $P = (Loc, \delta, l_{init}, l_{err})$ where

- $Loc$ is a set of finite nodes (locations),
- $\delta \subseteq Loc \times \Sigma \times Loc$ is the transition relation,
- $l_{init}$ is the *initial location*,
- $l_{err}$ is the accepting node called the *error location*.

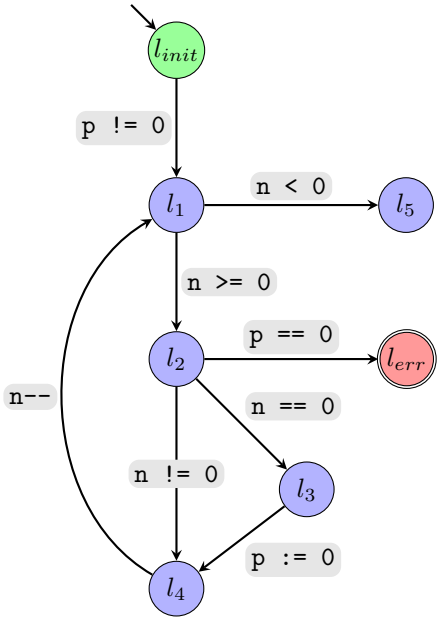
We sometimes consider another dead-end location called $l_{exit}$ to mark regular exit of the program. Figure 2.1 shows a program with it's CFG.

**Program state.** We employ a standard state based view of programs. Let $V$ be a fixed set of program variables. A *program state* is a valuation of program variables in $V$ from their respective domains. When we talk about a sequence of states, a variable $x \in V$ is denoted as $x'$ in the subsequent state. We express (sets of) program states by predicates over program variables, and statements by formulas over primed and unprimed variables. For example, the formula $p > 10$ represents the set of states where the program variable $p$ has a value strictly greater then 10. Given a predicate $\varphi$ over (unprimed) variables from $V$, we write $\varphi[V'/V]$ (or $\varphi'$ for short) for the predicate obtained by replacing all the occurrences of a variable $x \in V$ by $x'$. We sometimes switch between a symbolic and a set interpretation of states. For instance, when we write $s \in \varphi \subseteq \psi$ for some states $s$ and predicates $\varphi$, $\psi$, we mean $s \vDash \varphi$ and $\varphi \vDash \psi$ if interpreted in the symbolic view.

**Statements.** To describe program statements in control flow graphs, we use guarded commands [8]: the deterministic assignment `x := e` for some variable `x` and an expression `e`, the nondeterministic assignment `havoc x` for a variable

```
assume p != 0;
while(n >= 0)
{
    assert p != 0;
    if(n == 0)
    {
        p := 0;
    }
    n--;
}
```

(a) Program code




(b) Control Flow Graph

Figure 2.1: A program with it's Control Flow Graph (CFG). Example taken from [1]

x, and the assume statement `assume(`$\varphi$`)` for some predicate $\varphi$. It may come from statements like `if(`$\varphi$`)-then-else` or `assert(`$\varphi$`)`. In this dissertation, we mostly just write $\varphi$ for `assume(`$\varphi$`)` (for example, `y != 10` for the statement `assume(y != 10)`).

**Predicate transformers.** We recall the well-known predicate transformers WP (weakest precondition), PRE (precondition) and SP (strongest postcondition). Let $\varphi$, $\psi$ be predicates and $st$ be a statement.

$$WP(\psi, st) \equiv \forall V'.\ st \Rightarrow \psi[V'/V]$$
$$PRE(\psi, st) \equiv \neg WP(\neg\psi, st) \equiv \exists V'.\ st \wedge \psi[V'/V]$$
$$SP(\varphi, st) \equiv \exists V''.\ \varphi[V''/V] \wedge st[V''/V][V/V']$$

We assume the reader is familiar with the notion of a *Hoare triple*, which we write $\{\varphi\}$ st $\{\psi\}$ for precondition $\varphi$, statement st and postcondition $\psi$. A Hoare triple $\{\varphi\}$ st $\{\psi\}$ is valid if and only if the precondition $\varphi$ implies the weakest precondition WP($\psi$, st).

**Trace.** A *trace* $\pi$ is a sequence of statements. Let $\pi = \langle st_1, \ldots, st_n \rangle$ be a trace of length $n$. For $1 \leq i \leq n$ we use $st_i$ to represent the $i$-th statement and for $1 \leq i < j \leq n$, we use $\pi[i, j]$ to represent the sub-trace from position $i$ to $j$. We also use $\pi[j, n]$ to represent the *suffix trace* starting at position $j$ and $\pi[1, i]$ to represent the *prefix trace* till the position $i$.

The predicate transformer functions WP(), PRE() and SP() can easily be lifted to traces by applying them recursively, where the application to the empty trace $\langle\rangle$ is the identity, e.g., $\mathrm{WP}(\psi, \langle\rangle) = \psi$.

**Execution.** An *execution* $\xi$ of a trace $\pi = \langle st_1, \ldots, st_n \rangle$ of length $n$ is a sequence of states $s_0, s_1, \ldots, s_n$ such that the formula $\mathrm{SP}(s_{i-1}, st_i)$ is satisfiable for all $1 \leqslant i \leqslant n$.

**Infeasibility of a trace.** A trace $\pi = \langle st_1, ..., st_n \rangle$ is called *infeasible* if there is no possible execution of $\pi$. Otherwise, the trace is called *feasible*.

**Error Trace.** An *error trace* is a trace from $l_{init}$ to $l_{err}$. Note that an error trace is not necessarily feasible.

**Reachability.** Given a trace $\pi = \langle st_1, \ldots, st_n \rangle$, a state $s$ is *reachable at position $i$* if there exists an execution $s_0, \ldots, s_i$ of the prefix trace $\pi[1, i]$ such that $s_i = s$. Similarly, a state $s$ is *coreachable at position $i$* if there exists an execution $s_i, \ldots, s_n$ of the suffix trace $\pi[i + 1, n]$ such that $s_i = s$. A state is *bireachable at position $i$* if it is both reachable and coreachable at that position. Note that every state is reachable at position 0 and coreachable at position $n$. For a trace $\pi$ and predicates $\varphi$, $\psi$, we may restrict the executions of $\pi$ to those starting in $\varphi$ (the *precondition*) and ending in $\psi$ (the *postcondition*), i.e., $\xi$ is of the form $s_0, \ldots, s_n$ such that $s_0 \vDash \varphi$ and $s_n \vDash \psi$.

# Chapter 3

# Aberrant Trace Elements

In this chapter, we establish the technical basis for *aberrance* in program traces, which will allow us to build upon the concept further and use it in analyzing feasible error traces. The word aberrance stems from the Latin word *aberrare*, which means "*to deviate*". In English, *aberrant* means "a person, thing or a group that departs or deviate from the normal or usual course". We can think of an entity as aberrant if it causes deviation from normality.

Given a feasible trace, we call a trace element aberrant if a modification in that trace element can single-handedly make the whole trace infeasible. In other words, aberrant trace elements can deviate a feasible trace towards infeasibility. In the next sections, we will formalize the notion of aberrance and present an algorithm to find all such trace elements in a trace. Since a trace element is a statement, from here onwards, we will use the term *aberrant statements* instead of aberrant trace elements.

## 3.1 Aberrance

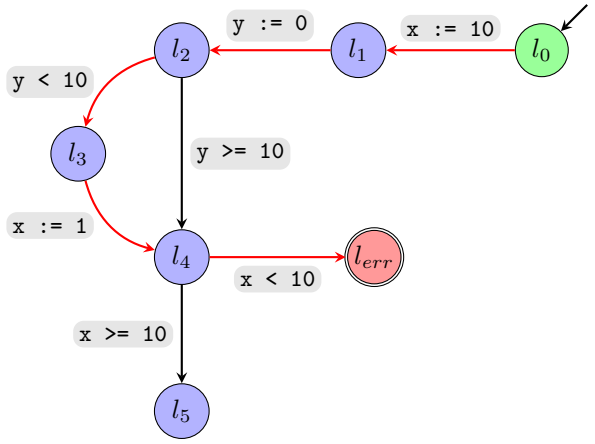
In the sense of playing a role in the feasibility of a trace, not all statements are created equal. Some assignment statements in the trace can be omitted or the value being assigned can be modified, without having any effect on the feasibility of the trace. This does not, however, necessarily mean that such statements are useless when analyzing a trace. Their effect might manifest only after modifying another statement.

We focus on those (possibly non-deterministic) assignment statements that can make the trace infeasible when the assigned value changes. Therefore, in the context of the feasibility of the trace, such statements are the most consequential assignments in the trace. Note that we do not consider conditional (i.e., `assume`) statements here. This is because *every* conditional can be changed to make the trace infeasible by modifying it to `assume(false)`.

We restrict ourselves to certain resonable modifications only. The variable that the value is being assigned to must remain the same and the new value must be a constant. That means that a deterministic assignment `x := e` and

```
1:  x := 10;
2:  y := 0;
3:  if(y < 10){
4:    x := 1;
5:  }
6:  assert(x >= 10);
```

(a) Program code




(b) Control Flow Graph

`x := 10` `y := 0` `y < 10` `x := 1` `x < 10`

(c) A feasible error trace

Figure 3.1: Simple program with a feasible error trace illustrating Aberrance

a non-deterministic assignment `havoc x` must be replaced by `x := c` for some constant `c`.

Observe that an assignment statement *st* of the form `x := e` is not aberrant only if the trace is feasible for *any* non-deterministic assignment to `x` at this point (i.e., we could just replace *st* with `havoc x`).

For the rest of this document, we use the term *assigning statement* to uniformly talk about deterministic and non-deterministic assignments. Before we give a formal definition for aberrant assigning statement, we define *blocked execution*.

**Definition 1** (Blocked Execution)**.** *Let* $\langle st_1, ..., st_n \rangle$ *be a trace of length* $n$*. We call an execution* $s_0, ..., s_{j-1}$ *of a sub trace* $\langle st_1, ..., st_j \rangle$ *a* blocked execution *if* $SP(s_{j-1}, st_j)$ *is unsatisfiable where* $1 \leq j \leq n$*.*

An assume statement `assume(`$\varphi$`)` for some predicate $\varphi$ at location $i$ can block the execution if $\varphi$ does not hold in the state $s_{i-1}$. Consider the following trace.

`x := 0` `y := 0` `assume(y < 0)` `x := 10` `assume(x != 10)`

The execution *true*, $x = 0$, $x = 0 \wedge y = 0$ of the sub trace $\langle$ `x := 0`, `y := 0`, `assume(y < 0)` $\rangle$ is a blocked execution. This is because SP($x = 0 \wedge y = 0$, `assume(y < 0)`) is not satisfiable. In fact, all executions of the above trace are blocked and the trace is infeasible.

```
1:  x := 10;
2:  havoc x;
3:  if(x < 10){
4:    y := 1;
5:  }
6:  assert(x >= 9);
```

(a) Program code

`x := 10` `havoc x` `x < 10` `y := 1` `x < 9`

(b) A feasible error trace

Figure 3.2: A program with non-deterministic assignment

We are now in a position to formally capture the definition of an aberrant assigning statement.

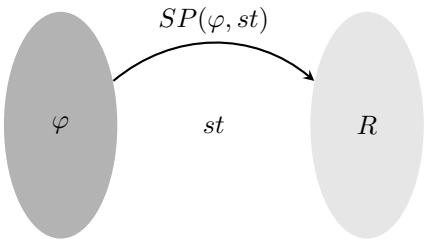
**Definition 2** (Aberrant Statement)**.** *Let $\pi$ be a feasible trace of length $n$ with $st_i$ being an assigning statement at position $i$, that assigns a new value to some variable $x$. $st_i$ is an* aberrant statement in $\pi$ *if there exists an execution $s_0, \ldots, s_n$ of $\pi$ and some value $v$, such that every execution of the suffix trace $\langle x := v, \pi[i+1, n]\rangle$ starting in the state $s_{i-1}$ is blocked.*

Let us make the intuition of aberrance more clear with the help of two examples. The example in Figure 3.1 shows a simple program with it's control flow graph. The only feasible error trace is shown in Figure 3.1c. The error trace contains two aberrant statements highlighted in red. The statement `x := 1` is aberrant because a modification of the right hand side of the assignment which in this case is a constant 1 to a constant 2 will violate the condition `x < 10` in the assume statement `x < 10`. As a result of this modification, no possible execution exits for this trace, rendering it infeasible. The statement `y := 0` is aberrant with a similar argument.

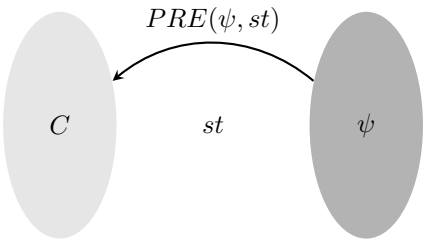
Consider another program in Figure 3.2 with a deterministic and a non-deterministic assignment to the variable `x`. The statement `x := 10` is not aberrant because no modification of the value `10` exists at this point, which makes the trace are infeasible. This is because the next statement, `havoc x`, can always assign a value to `x` such that the condition $x < 10$ is never violated in the statement `x < 10`. The statement `havoc x` will be aberrant because a value $v$ exists (e.g. `10`) such that if we assign `10` to `x` at this point, the trace becomes infeasible.

## 3.2 Finding Aberrant Statements

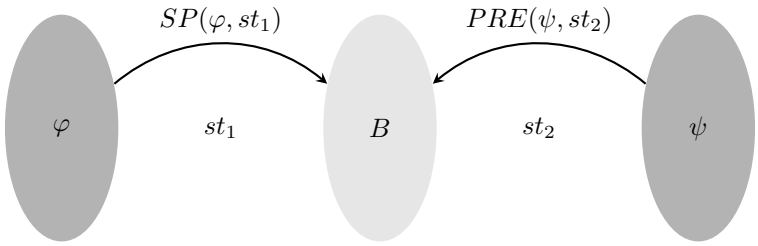
Now that we formally know what an aberrant statement is, we can turn to the question of how can we find such statements. Before we present an algorithm to find all aberrant statements in a feasible error trace, we rephrase the characterization of a (co/bi) reachable state in terms of PRE() and SP() operators. Assume we have a statement $st$, a precondition $\varphi$ and a postcondition $\psi$. The




(a) Computing set of reachable states $R$ from $st$ and $\varphi$




(b) Computing set of co-reachable states $C$ from $st$ and $\psi$




(c) Computing set of bi-reachable states $B$, where $B = R \cap C$

Figure 3.3: Computation of reachable, co-reachable and bi-reachable states.

set of reachable states $R$ can be computed as $R = \text{SP}(\varphi,\ st)$. The set of co-reachable states $C$ can be computed with the help of the PRE() operator as $C = \text{PRE}(\psi,\ st)$. $C$ represents a set of states such that for all states in $C$, there exists atleast one execution of $st$ that satisfies $\psi$. For two consecutive statements $st_1$, $st_2$ in an error trace with pre-condition $\varphi$ and post-condition $\psi$, the set of bi-reachable states between $st_1$ and $st_2$ can be computed as $\text{SP}(\varphi,\ st_1) \cap \text{PRE}(\psi,\ st_2)$.

In a trace $\pi = \langle st_1, \ldots, st_i, \ldots, st_n \rangle$, a precondition $\varphi$ and a postcondition $\psi$. A state $s$ is reachable at position $i$ if $s \in \text{SP}(\varphi, \pi[1, i-1])$. A state $s$ is co-reachable at position $i$ if $s \in \text{PRE}(\psi,\ \pi[i, n])$. A state $s$ is bireachable at position $i$ if $s \in \text{SP}(\varphi, \pi[1, i-1]) \cap \text{PRE}(\psi, \pi[i, n])$.

### 3.2.1 Algorithm

Let $\pi$ be a feasible trace of length $n$. In the first step of the algorithm, we compute the co-reachable and bi-reachable states along the trace. The co-reachable states $C_i$ can be iteratively computed as follows:

$$C_i := \begin{cases} true & i = n \\ \texttt{PRE}(C_{i+1}, \pi[i+1]) & i < n \end{cases}$$

Analogously, we can compute the reachable states $R_i$ as follows:

$$R_i := \begin{cases} true & i = 0 \\ \texttt{SP}(R_{i-1}, \pi[i]) & i > 0 \end{cases}$$

Finally, the bi-reachable states are simply obtained from the intersection, $B_i := R_i \cap C_i$. We note that in the setting where we get the error trace from a software model checker, the (co-)reachable states might already be available for free if the analysis was based on WP/SP [9].

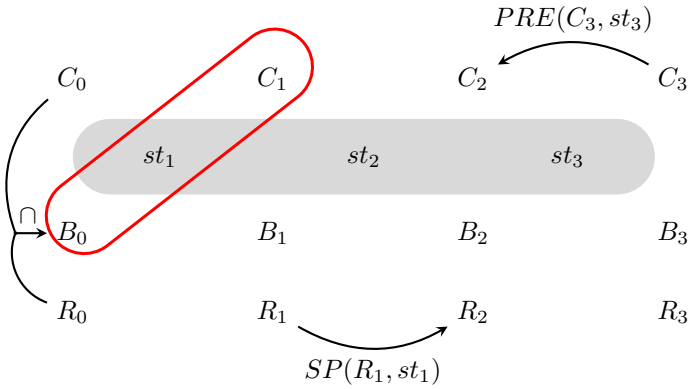



Figure 3.4: The trace is highlighted in grey. The co-reachable states $C_i$ and reachable states $R_i$ are computed iteratively. The set of bireachable states $B_i$ are obtained by the intersection of $C_i$ and $R_i$. A Hoare triple is marked in red.

Given the sequences $C_i$ and $B_i$, for any assigning statement $\pi[j]$ with assigned variable $x$, we construct the following Hoare triple.

$$\{B_{j-1}\}\ \texttt{havoc}(x)\ \{C_j\}$$

The statement will be aberrant if and only if the Hoare triple is not valid. Hoare triple validity can be checked by a theory solver for many theories that are used in practice [10]. We summarize the procedure in Algorithm 1. Note that the check for aberrance is independent for each statement, i.e, the loop in Algorithm 1 can be parallelized.

### 3.2.2 Algorithm Correctness

We now show that Algorithm 1 computes aberrant statements according to our formal definition of aberrance.

**Theorem 1.** *Let $\pi$ be a feasible trace of length $n$ with $\pi[i]$ being an assigning statement at position $i$ that assigns a value to some variable $x$. Let $\varphi$ be the set of bireachable states at position $i$ and $\psi$ be the coreachable states at position $i+1$. Then $\pi[i]$ is an aberrant statement iff*

$$\{\varphi\}\ \texttt{havoc}(x)\ \{\psi\}\ \textit{is invalid.}$$

**Algorithm 1:** Aberrant statement positions in a trace.

**Input:** $\pi[i]$ for $1 \leq i \leq n$: sequence of statements
$C_i$ for $0 \leq i \leq n$: sequence of co-reachable states
$B_i$ for $0 \leq i \leq n$: sequence of bi-reachable states
**Output:** res: list of aberrant statement positions

```
1 res ← [];
2 for j = 1 to n do
3 |   if π[j] is not an assigning statement then continue;
4 |   Let x be the assigned variable in π[j];
5 |   if {B_{j-1}} havoc(x) {C_j} is invalid then res.append(j);
6 end
```

*Proof.* Let $\mathcal{D}$ be the domain of the variable $x$.

$\pi[i]$ is aberrant.

$\iff \exists s \in \varphi.\ \exists v \in \mathcal{D}.$ all the executions of $\langle x := v; \pi[i+1, n]\rangle$ starting in the state $s$ are blocking

$\overset{(*)}{\iff} \exists s \in \varphi.\ \exists v \in \mathcal{D}.\ \mathrm{SP}(s, x := v) \notin \psi$

$\overset{(*)}{\iff} \exists s \in \varphi.\ s \notin \mathrm{WP}(\psi, \texttt{havoc}(x))$

$\iff \varphi \not\subseteq \mathrm{WP}(\psi, \texttt{havoc}(x))$

$\iff \{\varphi\}\ \texttt{havoc}(x)\ \{\psi\}$ is invalid

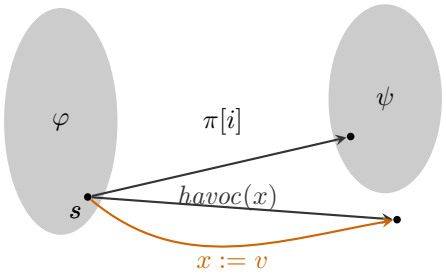



Figure 3.5: Additional explanation for the proof of Theorem 1.

We only explain the non-trivial steps marked with $(*)$. First note that, since $s \in \varphi$, there is at least one assignment to $x$ that leads to $\psi$. The aberrance of $\pi[i]$ implies that there exists a value $v$ such that the assignment of $v$ to `x` will lead us to a state in $\neg\psi$ (all executions the of the suffix trace are blocking if we start the execution from a state in $\neg\psi$). Hence $\mathrm{SP}(s,\ x := v) \notin \psi$

Moreover, since from state $s$ there is an assignment to $x$ that leads outside of $\psi$, this successor can also be reached by a nondeterministic assignment $x$, i.e., not all successors of $s$ under `havoc(x)` are in $\psi$. On the other hand, every successor of $s$ under `havoc(x)` can be reached by an assignment to $x$. We

graphically illustrate the situation in Figure 3.5. □

### 3.2.3 Example

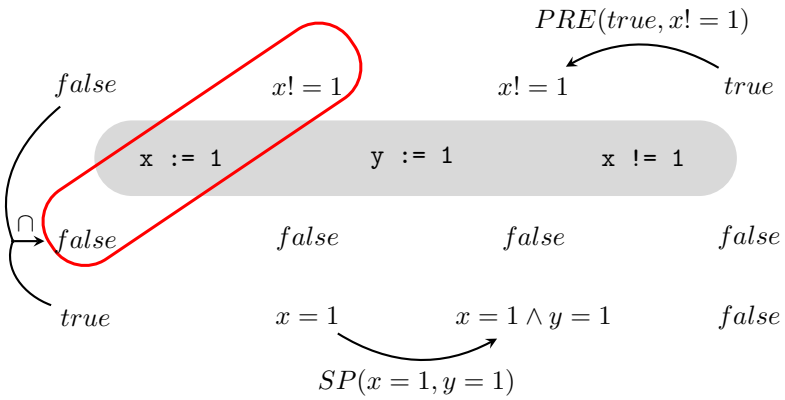



Figure 3.6: For the error trace highlighted in grey, the first statement is aberrant since the hoare triple $\{false\}$ `x:= 1` $\{x! = 1\}$ is invalid.

### 3.2.4 Discussion

#### Co-reachable States

To understand why the last co-reachable state $C_n$ is $true$, we have to ask what does the co-reachable states represent in our algorithm. Let $\pi[i]$ be an assigning statement of the form `x := e` at position $i$ in a trace $\pi$ of length $n$. Computation of aberrance of the assigning statement $\pi[i]$ is checking the validity of the hoare triple $\{B_{i-1}\}$ `havoc(x)` $\{C_i\}$. In simple words, this check allows us to conclude if there is an assignment to `x`, which leads to a state not in $C_i$ or $\neg C_i$. We want $\neg C_i$ to be a set of states from which all executions of the trace $\pi[i+1, n]$ are blocking ($\pi[i+1, n]$ is infeasible). Hence, $\neg C_i$ = WP($false$, $\pi[i+1, n]$) and $C_i = \neg$WP($false$, $\pi[i+1, n]$) = PRE($true$, $\pi[i+1, n]$).

#### Bi-reachable States

For a statement $\pi[i]$, the precondition $B_{i-1}$ in the hoare triple is the intersection between the set of reachable states $R_{i-1}$ and the set of co-reachable states $C_{i-1}$. The reason for using the bi-reachable states will become apparent with the help of the example in figure 3.7.
According to the definition of aberrance, `y := 7` should not be aberrant, as no assignment to `y` exists at this point such that the trace is infeasible. Let us dig a bit deeper and see what does Algorithm 1 compute . The hoare triple check for this statement would be $\{x = 5\}$ `havoc(y)` $\{x = 5 \vee y = 7\}$. This obviously is a valid hoare triple and the statement `y := 7` will not be marked aberrant by our

```
x := 5;
y := 7;
assert (x != 5 && y != 7)
```

(a) Program code

`x := 5` `y := 7` `x == 5 || y == 7`

(b) A feasible error trace

Figure 3.7: No statement in the trace is aberrant

algorithm. If instead of bi-reachable states, we used co-reachable states as the precondition, the hoare triple check would be $\{true\}$ `havoc(y)` $\{x = 5 \vee y = 7\}$ which is invalid and consequently `y := 7` would be marked as aberrant.

### Syntactic Algorithm

One might argue that Algorithm 1 is too complex for our purpose and the same result can be achieved with a much simpler algorithm based only on WP computations along the trace. In the proposed algorithm, for a trace $\pi = \langle st_1, ..., st_n \rangle$, a WP sequence $wp_0, ..., wp_n$ is computed along the trace where $wp_n = false$ and $wp_i = WP(false, \pi[i+1, n])$ for $0 \leq i < n$. An assigning statement $st_i$ is aberrant if $wp_i$ contains the variable in $st_i$.

The example in Figure 3.7 demonstrates why such an algorithm is not suitable for our notion of aberrance. The statement $st_2 =$ `y := 7` will be marked aberrant since $wp_2$ is $x! = 5 \wedge y! = 7$. The statement `y := 7` is not aberrant according to our notion since no assignment to `y` exist at this point which makes the trace infeasible.

# Chapter 4

# Applications

In this chapter, we present four applications of aberrance analysis that can be helpful in practice. In the first application, given a feasible error trace, we point out statements that are potentially faulty and are therefore candidates for a simple modification which fixes the bug. Our approach for fault localization using aberrance on a feasible error trace not only points out which statements in an error trace might be causing the error but also how the error can be fixed. In the second application, we extend this idea to identify the presence of unvalidated inputs in a program which is classified as a typical security vulnerability. The third application is in the verification of programs that expose external libraries and APIs. Here, we use aberrance to suppress and later rank error warnings based on how useful they might be for the user from a debugging point of view. In the fourth application we show how we can use aberrance to help a software model checker that over-approximates statements to give a definite answer.

## 4.1 Finding Faulty Statements In An Error Trace

We consider the setting where we are given a single feasible error trace $\pi$, e.g., from a software model checker. Assuming that the error specification is correct, we can conclude that at least one of the statements in the trace must be faulty. Our working assumption, for now, is that there is only a single fault in the trace. Given a long feasible error trace, usually not all statements are actually related to the fault. Here we consider the task of identifying possibly faulty statements, which is also known as *fault localization*. Observe that fixing the fault will make the error trace $\pi$ infeasible. If we assume that the fault manifests in a single statement $st_i$, a fix will correspond to a modification of $st_i$.

### 4.1.1 Assignment Statements

Given a feasible error trace $\pi$, if $st_i$ is an assigning statement and the prefix trace $\pi[1, i-1]$ is deterministic, then there exists a modification for the statement $st_i$ that can make $\pi$ infeasible if and only if $st_i$ is aberrant.

Thus we can classify assigning statements $st_i$ as follows. Assume $st_i$ is aberrant. In this case a modification of the assigned value will make the trace infeasible. Then we know both where and how we could fix the error trace. Now assume $st_i$ is not aberrant. In that case, a modification of the assigned value alone will not make the trace infeasible. Then we know that this statement is either not related to the fault or fixing the fault requires changing at least one more statement.

According to Ockham's razor, when in doubt, one should choose the approach with the simplest assumptions. In our setting, a reasonable strategy could hence be to first look at all assignments that are aberrant, because those are precisely the statements that would only involve a single modification. Only if no fault could be found one should consider the more complicated setting of multi-statement modifications.

Consider the following program.

```
1:    int *p1, *p2, i;
2:    p1 := 0;
3:    p2 := 0;
4:    i := 1; // bug: should be 0
5:    while (i < 10) {
6:        if (i == 0) {
7:            p1 := malloc(sizeof(int));
8:            p2 := malloc(sizeof(int));
9:        }
10:        assert p1 != 0 && p2 != 0;
11:        i := i + 1;
12:    }
```

The program initializes two pointers to `null` and enters a loop. The first loop iteration is supposed to allocate memory and assign it to the pointers, but the programmer accidentally initialized the loop counter `i` to 1 instead of 0, and hence this step is skipped. The safety specification is given by the `assert` statement which expresses that the pointers should not point to `null` at the end of the loop body.
The (unique) feasible error trace is given as follows.

`p1 := 0` `p2 := 0` `i := 1` `i < 10` `i != 0` `p1 == 0 || p2 == 0`

First observe that the trace corresponds to a single execution (i.e., there is no nondeterminism involved; every variable is initialized). If we apply the definition, then the assignment to `p1` is aberrant only if we can find another assignment to `p1` that makes this trace infeasible. However, this is not the case here because `p2` would still be initialized to 0 and thus the error condition at the end would still be satisfied. The assignment to `p2` is also not aberrant with an analogous argument. We can, however, find an assignment to `i` that makes the trace infeasible. One choice would be the value 0, which corresponds to the actual fix of the bug. Another choice would be the value 10.

```
1:    x := 10;
2:    y := 10;
3:    if(y <= 10) {
4:       x := 1;
5:    }
6:    assert x >= 10;
```

Figure 4.1: Two simple fixes can avoid the assertion failure in this program

Other bug fixes that one can imagine are to initialize the two pointers outside the loop or replacing the `if` condition with `i==1`. We could even try to restructure the control flow of the program. However, we would then open Pandora's box and enter the field of program synthesis, which we strictly want to avoid. We rather settle for the simple bug fix as suggested by our definition of aberrant statements. If we cannot find such a simple bug fix, the only help we can give to the developer is that no simple bug fix exists (according to our definition), which itself might be a valuable information, e.g., when estimating debugging time or prioritizing jobs. In such cases one may also fall back to other fault localization techniques (if they are applicable).

Consider another program in Figure 4.1. Variables `x` and `y` are initialized to `10`. The assertion failure is due to the assignment of `1` to the variable `x` in line 4. The assertion failure in the program can be avoided by modifying the assignment at line `4`. Another way to fix the program would be to modify the initialization value to the variable `y` at line `2`. If `y` is assigned a value greater then `10`, the statements inside the `then` branch would not be executed and `x` would not be assigned the value `1` which would avoid the assertion violation. The feasible error trace is given as follows (aberrant statements are marked in red):

`x := 10` `y := 10` `y <= 10` `x := 1` `x < 10`

Consider another program in Figure 4.2 where the assignment of `11` to the variable loc at line `3` can cause a buffer overflow in line `6`. In our setting, `loc := get_value()` where `get_value()` takes a value from the user is over-approximated and modeled as `havoc loc`. The assignment to `loc` is the only statement that can cause the assertion to fail in the end. The error trace is given as:

`havoc loc` `val := 10` `loc < 0 || loc >= 10`

Most fault localization techniques employ lots of different program executions for their analysis. The reason is that the information gained from a single error trace is limited. Even though the bug must be present on the error trace, fixing a trace is much simpler than fixing a program. Clearly, not every fix that makes a single error trace infeasible also fixes the bug in the program.

To summarize, aberrant statements have the following properties.

```
1:    int a[10];
2:    int loc, val;
3:    loc := get_value();   // get value from user
4:    val := 10;
5:    assert(loc >= 0 && loc<10)
6:    a[loc] := val;
```

Figure 4.2: Simple example of a buffer overflow

1. The aberrant statements are *statically computable from a single error trace* without the need for additional context like, e.g., sampling of executions along that trace.
2. The mere fact *that* a statement is declared aberrant indicates both *where* and *how* the bug *could* be fixed.
3. Conversely, the fact that a statement is declared *not* aberrant indicates where and how the bug *cannot* be fixed.

### 4.1.2 Error Traces With Branches

Analyzing a single feasible error trace to localize faults in a program has inherent limitations which become apparent when the error trace passes through a *branch*, say, the `then` part of an `if-then-else` conditional. This shortcoming is due to the nature of how error traces are encoded. To understand this, recall that an error trace is simply a path from the start state to the error state in a control flow graph of the program. An error trace, therefore, does not take into account the control flow of the program and lose all the information for the other branches in the CFG. The application of aberrance to find faulty statements, hence, will not give accurate results for error traces passing through a branch. That would mean that an assignment statement might be aberrant in an error trace, but modifying that assignment in the program will have no effect in fixing the error. Consider the program shown in Figure 4.3. A feasible error trace that visits the `then` branch is:

`x := 0` `havoc y` `y < 0` `y := 0` `x == 0`

In this trace, the assignment to `y` in line 2 is aberrant but modifying this assignment in the program will have no effect in fixing the error, as the assertion in the end will still be violated. The assignment to `y` is aberrant due to the conditional statement `y < 0`, which can block the execution of the trace. Modification of statements that are aberrant due to the conditional statements of branches may not always act as potential bug fixes like we described in Section 4.1. Consider the modified program in Figure 4.4 and the corresponding feasible error trace in Figure 4.4b

In this program, the faulty assignment statement lies in the `then` branch at line `4`. In the feasible error trace, the assignment to `y` at line `2` is again aberrant,

```
1:   int x := 0;
2:   int y := get_y();
3:   if (y < 0) {
4:       y := 0;
5:   } else {
6:       y := 1;
7:   }
8:   assert x != 0;
```

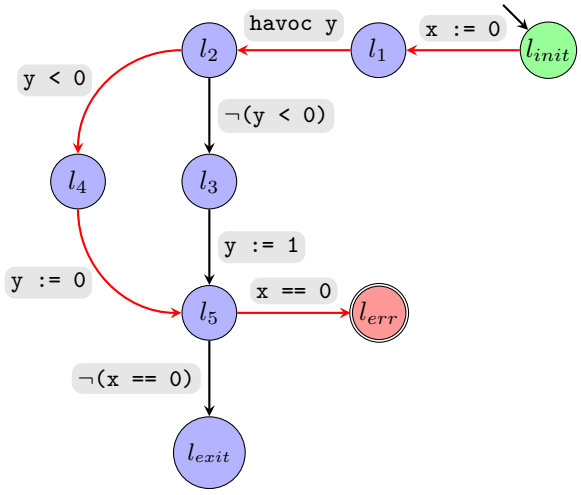



Figure 4.3: The assertion is violated no matter which branch of the conditional is taken.

but a modification to `y` can force the program execution to take the `else` branch which will in fact avoid the assertion failure in the end. Another option is also to fix the faulty assignment `x := 0` in the `then` branch, which is also computed to be aberrant by our algorithm.

```
1:    int x := 1;
2:    int y = get_y();
3:    if (y < 0) {
4:        x := 0;
5:    } else {
6:        y := 1;
7:    }
8:    assert x != 0;
```

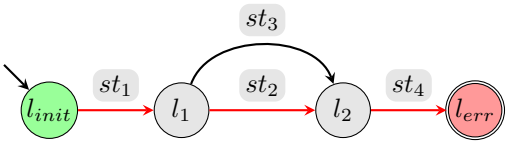
(a) Program code

`x := 1` `havoc y` `y < 0` `x := 0` `x == 0`

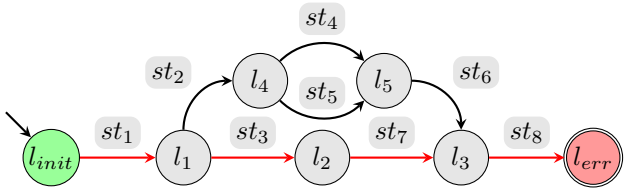
(b) A feasible error trace

Figure 4.4: The assertion is violated only if the `then` block is taken.

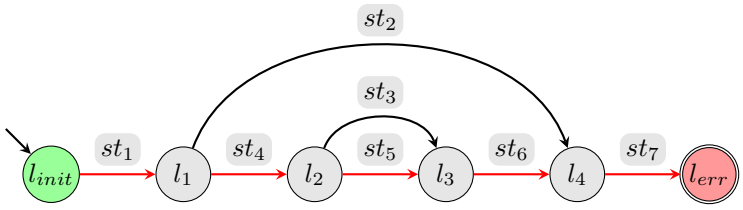
From the two examples discussed above, it is clear that using aberrance, our ability to accurately find simple potential bug fixes is hindered for error traces that pass through branches. To overcome this limitation, two possible solutions come to mind. (a) Analyze multiple feasible error traces independent of each other and consider only those statements as potential bug fixes if they are aberrant in all the considered feasible error traces. (b) Modify the encoding of error traces to include branch information. The program in Figure 4.3 has two feasible error traces, but the statement `havoc y` is aberrant in both of them. Analyzing multiple feasible error traces, therefore, will not help us in this case.



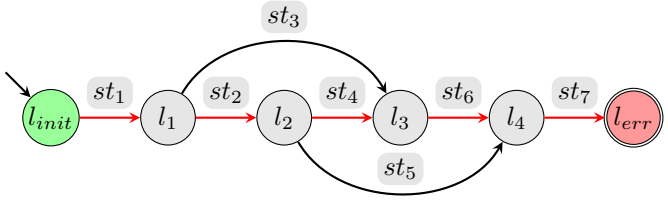

(a) Two paths from $l_1$ to $l_2$




(b) Three paths from $l_1$ to $l_3$




(c) Three paths from $l_1$ to $l_4$




(d) Three paths from $l_1$ to $l_4$

Figure 4.5: Possible branching structures in a CFG

We will, therefore, explore the second option, i.e., modifying the encoding of the error trace in a such a way, that the branching information from the control flow graph of the program is taken into account.

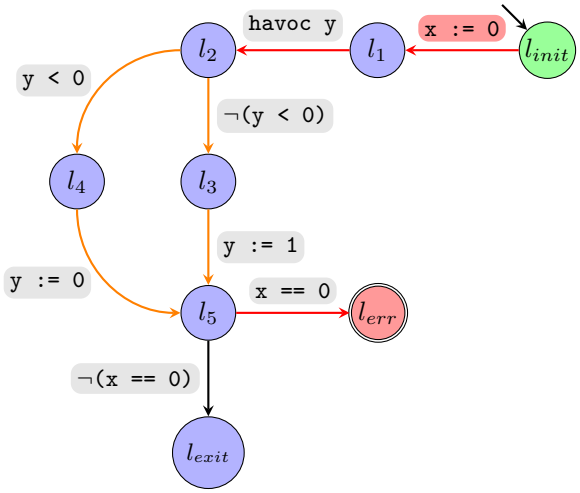
### Branch

Before we explain how we can encode the branching information from the CFG into an error trace, we formally describe a *branch in an error trace*.

Given a feasible error trace $\pi = \langle st_1, ..., st_n \rangle$, let $\rho = \langle l_{init}, ..., l_{err} \rangle$ be the sequence of locations along the error trace in the control flow graph starting from the initial location $l_{init}$ to the error location $l_{err}$. A location $l_i \in \rho$ is called a *must-visit* location if every path from $l_{init}$ to $l_{err}$ passes through this location in the control flow graph. Let $\rho_{must}$ be the sequence of must-visit locations in

```
1:    int x := 0;
2:    int y := get_y();
3:    if (y < 0) {
4:        y := 0;
5:    } else {
6:        y := 1;
7:    }
8:    assert x != 0;
```

(a) Simple prgram with an if then else




(b) Control Flow Graph

`x := 0` `havoc(y)` `y < 0 && y' := 0 || y >= 0 && y' := 1` `x == 0`

(c) Error trace with conglomerate

Figure 4.6: Program with it's control flow graph. The conglomerate is shown in orange.

$\rho$. Let $\pi_{l_{out},l_{in}}$ be the error sub-trace between the locations $l_{out}$ and $l_{in}$.

**Definition 3** (Branch)**.** *Given a feasible error trace $\pi$ of length $n$, an error sub-trace $\pi[i,j]$ is called a branch if $\pi[i,j] = \pi_{l_{out},l_{in}}$ such that $l_{out} = \rho_{must}[s]$ and $l_{in} = \rho_{must}[s+1]$ and there are more then one paths between the locations $l_{out}$ and $l_{in}$ in the CFG. $l_{out}$ is called a* branch-out *location and $l_{in}$ is called a* branch-in *location.*

Consider the examples of possible branching structures in Figure 4.5. The error traces $\pi$ are shown in red in the corresponding CFGs. In Figure 4.5a $\pi = \langle st_1, st_2, st_4\rangle$, $\rho_{must} = \rho = \langle l_{init}, l_1, l_2, l_{err}\rangle$, $\pi[1] = st_2$ is a branch, since $\pi_{l_1,l_2} = st_2$ and $l_1 = \rho_{must}[1]$ and $l_2 = \rho_{must}[2]$ and there are more then one paths between the location $l_1$ and $l_2$. In the example in Figure 4.5d, $\pi = \langle st_1, st_2, st_4, st_6, st_7\rangle$, $\rho = \langle l_{init}, l_1, l_2, l_3, l_4, l_{err}\rangle$, $\rho_{must} = \langle l_{init}, l_1, l_4, l_{err}\rangle$. $\pi[1,3] = \langle st_2, st_4, st_6\rangle$ is a branch, since $\pi_{l_1,l_4} = \langle st_2, st_4, st_6\rangle$ and $\rho_{must}[1] = l_1$ and $\rho_{must}[2] = l_4$ and there are more then one paths between the locations $l_1$ and $l_4$.

### Encoding Branch Information In The Trace

From the program in Figure 4.3, it is clear that using aberrance to locate simple potential bugs fixes in a feasible error trace would require considering both

```
1:   x := 10;
2:   y := 0;
3:   if(y < 10){
4:     x := 1;
5:   }
6:   assert(x >= 10);
```

(a) Program code

`x := 10` `y := 0` `y < 10 && x := 1 || ¬(y < 10)` `assume(x < 10)`

(b) Branch replaced by it's respective conglomerate

`x := 10` `y := 0` `assume(y < 10)` `x := 1` `assume(x < 10)`

(c) Error trace with the `then` branch

Figure 4.7: The conglomerate is aberrant due to the presence of an aberrant statement in the `then` branch

the `then` and the `else` branch of the `if-then-else` conditional. This example brings to light an underlying limitation of fault localization with single error traces for programs that contain conditionals that might come from programming constructs like `if-then-else`, loops and goto statements. Such statements introduce multiple outgoing edges for a single location in the control flow graph of the program raising the possibility of multiple paths between two locations. We encode the error trace in a way such that all of these multiple paths are taken into account during the computation of aberrant statements. We do this by replacing all the statements in a branch by single statement called a *conglomerate*.

**Definition 4** (Conglomerate)**.** *Given a feasible error trace $\pi$ of length $n$, let $\pi[i, j]$ be a branch such that $1 \leq i \leq j \leq n$. Let $l_{out}$ and $l_{in}$ be the corresponding branch-out and branch-in locations in the CFG for the branch $\pi[i, j]$. Suppose there are $m$ number of paths between $l_{out}$ and $l_{in}$ other the $\pi[i, j]$. A conglomerate is a disjunction between all the paths between $l_{out}$ and $l_{in}$, including $\pi[i, j]$.*

In the case where an error trace passes through a `then` branch, the conglomerate would then be a disjunction of the `then` branch and the `else` branch. Consider again the program in Figure 4.3 where the error trace $\pi$ is highlighted in red in the CFG. For the branch $\pi[2, 3] = \langle$ `y < 0` , `y = 0` $\rangle$, there is only one other path ( $\langle$ `¬(y < 0)` , `y = 1` $\rangle$ ) in the CFG from the branch-out location $l_2$ to the branch-in location $l_5$. The error trace where the branch is replaced by a conglomerate is shown in Figure 4.6c where only the first statement `x := 0` is now aberrant.

```
1:  y := 1;
2:  havoc x;
3:  if(x = 10){
4:    x := 2;
5:  } else {
6:    y := 2;
7:  }
8:  assert(y != 1);
```

(a) Program code

`y := 1` `havoc x` `x == 10 && x' := 2 || x != 10 && y' := 2` `y == 1`

(b) Branch replaced by it's respective conglomerate

`y := 1` `havoc x` `x == 10` `x := 2` `y == 1`

(c) Error trace with the `then` branch

Figure 4.8: The conglomerate is aberrant because of an aberrant statement in the `else` branch

### Aberrant Conglomerates

We cannot simply replace all the branches by their respective conglomerates because we also want to be able to find a bug fix inside a branch. This is where aberrant conglomerates come into play.

Let us consider the program in Figure 4.7a with an `if-then-else` statement where the else branch is empty. The feasible error trace where the branch is replaced by it's respective conglomerate is shown in Figure 4.7b, where the conglomerate is aberrant. This is due to the presence of an aberrant statement `x := 1` in the `then` branch. For an aberrant conglomerate, we therefore run the aberrance analysis again on the feasible error trace, where the branch is not encoded as a conglomerate.

An important point to note here is that an aberrant conglomerate does not necessarily mean that there is a simple bug fix (or an aberrant statement) in the corresponding branch. Consider the example in Figure 4.8, where the conglomerate is aberrant but there is no aberrant statement in the `then` branch. The conglomerate is aberrant due to a feasible `else` branch and the presence of an aberrant statement in the `else` branch. In fact, an aberrant conglomerate does not even guarantee that there is a single aberrant statement in either the `then` branch or the `else` branch. This is for the cases where two statements are aberrant only if they are considered together. Consider error trace in Figure 4.9b. The statements `x := 1` and `y := 1` are not aberrant. But consider the following trace in Figure 4.9c, where the `then` branch and the `else` branch is encoded as a conglomerate. The conglomerate is aberrant.

```
1:   a := 1;
2:   if(a==1) {
3:     x := 1;
4:     y := 1;
5:   }
6:   assert(x != 1 && y != 1);
```

(a) Program code

`a := 1` `a == 1` `x := 1` `y := 1` `x == 1 || y == 1`

(b) Error trace with the `then` branch

`a := 1` `a == 1 && x := 1 && y:=1 || a != 1` `x == 1 || y == 1`

(c) Error trace with the branch replaced by a conglomerate

Figure 4.9: The conglomerate is aberrant even tough there is no single aberrant statement present in either the `then` branch or the `else` branch.

### 4.1.3 Call and Return Statements

For interprocedural programs, a buggy call statement might be the reason for the error. Or there might be a bug in the return statement. Consider a very simple program in Figure 4.10 where a faulty value passed to the procedure `writeToArray` in the call statement can cause a buffer overflow to occur.

```
int globalArray[10];
void writeToArray(int value, int location){
    globalArray[location] = value;
}
void main(void) {
    writeToArray(0,11);
}
```

Figure 4.10: Simple example of a buffer overflow that can be triggered by a buggy call statement

In this program, clearly the error is being caused because of the faulty call statement. Call or return statements can simply be modeled as assignments.

Suppose we have a procedure `p` with one input parameter `x` and one output parameter `res`. The input parameter `x` is assigned to a variable $\texttt{x}_p$ in `p`. On return, the value of the procedure's output variable `res` is stored into the special variable $\texttt{res}_p$ of the caller. The procedure call appear in the following form.

$$\texttt{res}_p = \texttt{call p(x)}$$

and in `p` the return statement appear as

```
return res
```

A call statement is aberrant if the assignment $\texttt{x} = \texttt{x}_p$ is aberrant. A return statement is aberrant if the assignment $\texttt{res}_p = \texttt{res}$ is aberrant.

### 4.1.4 TCAS Experiments

In this section we demonstrate the capability of using aberrant statements as a technique to localize faults by testing our implementation (see Chapter 5) on TCAS (Traffic Collision Avoidance System) programs from the Siemens test suite. The Siemens programs were assembled by Tom Ostrand *et al.* at Siemens Corporate Research for a study of the fault detection capabilities of control-flow and data-flow coverage criteria [11]. Programs in the Siemens test suite perform a variety of tasks. TCAS is aircraft conflict detection and resolution system. It continuously monitors the radar information to check whether there is any neighboring aircraft that could represent a potential threat by getting too close. Depending on different parameters like vertical separation between two aircrafts and trajectory, TCAS issues a Resolution Advisory (RA) suggesting the pilot either climb or descend to avoid a collision. The programs used for our experiments in this section focuses on this very component of TCAS which is responsible for finding the best RA. The component is made up of 173 lines of code. The authors from Siemens created 41 versions of the correct program by injecting one or more faults. The goal was to introduce as realistic faults as possible. Ten people performed the fault seeding, working mostly without knowledge of each other's work. The Siemens programs are described in detail in the original paper [11]. Along with the faulty versions, the test suite also provides the original program along with a number of test vectors. The TCAS program used in this dissertation is obtained from the Software-artifact Infrastructure Repository of University of Nebraska[1]. We ran our implementation on five faulty versions which we refer to as v1, v2, v6, v7 and v11. The assertion checks used in our experiments were obtained from a previous work using symbolic execution by Porisini *et al.* [12].

Table 4.1 shows the results of running our implementation in Ultimate Automizer on 6 buggy versions of the TCAS program. The column "Line number" shows the line number where the bug is introduced in the original program. "Correct version" shows the code at Line number in the original program and "Buggy Version" shows the mutated code in the faulty version. v11 contains 3 bugs. With the exception of the third bug in v11, we were able to pin point all the bug locations using our approach.

Figure 4.11 gives an overview of a faulty version of TCAS (version 2), where a bug is injected at line 5. The correct version is displayed commented out in line 6. The bug is a mutation of a constant value from 100 to 300 in the function `Inhibit_Biased_Climb()`. The initializations and declaration of variables and

[1] http://sir.unl.edu/portal/index.php

Table 4.1: Bug details about TCAS programs in our experiments

| Version | Line number | Correct version | Buggy Version |
|---|---|---|---|
| v1 | 82 | !(Down_Separation >= ALIM()); | !(Down_Separation > ALIM()); |
| v2 | 70 | Up_Separation + NOZCROSS | Up_Separation + MINSEP |
| v6 | 111 | Own_Tracked_Alt < Other_Tracked_Alt | Own_Tracked_Alt <= Other_Tracked_Alt |
| v7 | 57 | res = Positive_RA_Alt_Thresh_2; | res = 700; |
| v11 | 111 | Own_Tracked_Alt < Other_Tracked_Alt | Own_Tracked_Alt <= Other_Tracked_Alt |
| | 118 | Own_Tracked_Alt < Other_Tracked_Alt | Own_Tracked_Alt <= Other_Tracked_Alt |
| | 138 | if() condition | **condition removed** |

functions that are not relevant for the bug have been omitted in the code in Figure 4.11. The safety property violated by the program is shown in line 58. We take one failing test case out of 69 [15] for this version to get a feasible error trace and analyze it to find aberrant statements. Our algorithm returned 15 aberrant statements in the trace, which means there are 15 potential bug locations (shown in red). The user now has to look into only 15 statements to fix the bug which is which is 8.6% of the total number of lines in the program.

Let us now take a closer look at the reported aberrant statements.

1. The aberrant statement at line 55 is too weak for a fix as the return value can always be modified in a way which can avoid the assertion violation in the end. Same argument holds for statement at line 49, where `alt_sep` is the value that is returned.

2. The assignment to the variable `enabled` at line 39 is based on input variables and can be quickly checked by the programmer.

3. The assignments at line 44 and 45 are based on the value returned by the functions `Non_Crossing_Biased_Climb()` and `Non_Crossing_Biased_Descend()` respectively. Both functions call the buggy function `Inhibit_Biased_Climb()`. Both the functions `Own_Below_Threat()` and `Own_Above_Threat()` return a value based on the input parameters and hence we can ignore them for now and look at the other 2 functions.

4. The return values at lines 18 and 30 are again too weak for a fix as in point 1.

5. The aberrant statements at line 12 and 24 is where the functions `Non_Crossing_Biased_Climb()` and `Non_Crossing_Biased_Descend()` call the faulty function respectively.

6. The aberrant statements at line 14 and 26 are also potential locations for a bug fix. This is where the wrong evaluation happens because of the faulty value returned by `Inhibit_Biased_Climb`.

7. The aberrant statement at line 5 is where the actual bug lies.

### 4.1.5 Performance

To demonstrate the performance and applicability of our algorithm on error traces originating from complicated programs, we run our implementation in Ultimate Automizer (see Chapter 5 ) on programs from the competition on software verification (SV-COMP). We performed the experiments on a PC with an Intel Core i7 CPU @ 2.7 GHz running the linux operating system with 15.6 GB RAM. The result is presented in Table 4.2.

Table 4.2: Time comparison with and without aberrance analysis (time in seconds)

| Program Name | Trace length | Time for aberrance analysis | Overall time | Difference |
|---|---|---|---|---|
| loops1 | 13 | 0.1 | 0.2 | 0.1 |
| loops2 | 14 | 0.1 | 0.2 | 0.1 |
| loops3 | 48 | 0.2 | 1.6 | 1.4 |
| recursive1 | 125 | 1.7 | 6.3 | 4.6 |
| recursive2 | 328 | 6.1 | 14.6 | 8.5 |
| recursive3 | 190 | 10.1 | 24.8 | 14.7 |
| recursive4 | 30 | 0.2 | 1.6 | 1.4 |
| recursive5 | 126 | 1.2 | 5.8 | 4.6 |
| recursive6 | 532 | 14.4 | 28.3 | 13.9 |
| ssh1 | 245 | 2.8 | 33 | 30.2 |
| ssh2 | 267 | 5.6 | 38.6 | 33 |
| ssh3 | 139 | 0.9 | 6.0 | 5.1 |
| ssh4 | 233 | 3.8 | 49.8 | 46 |
| ssh5 | 244 | 4.0 | 19.7 | 15.7 |
| systemc | 376 | 26.7 | 32.1 | 5.4 |

### 4.1.6 Related Work

Fault localization is a vast topic in the field of program analysis and an abundance of techniques and methods has been proposed over the years. Vaguely speaking, the general goal in the field is to identify program statements that "have something to do with the error". But the exact notion of what that means differ quiet a lot. It depends on the program, the programmer and the error type. Even after decades worth of research, we still have not yet seen *the best* fault localization technique. A lot of different approaches have their own complementary strengths and weaknesses. Nonetheless, as the importance of software systems increase, the importance of coming up with robust fault localization techniques has never been more important.

Analyzing a counterexample produced by a software model checker [1,13,14] has become an accepted technique for fault localization and can be quite useful for debugging [5,6,15–20]. However, as the size of the counterexample increases, it becomes almost prohibitively difficult for a human to analyze it manually.

```
1   int ALIM () {
2    return Positive_RA_Alt_Thresh(Alt_Layer_Value);
3   {
4   int Inhibit_Biased_Climb () {
5       return (Climb_Inhibit ? Up_Separation + 300 : Up_Separation);
6       /* return (Climb_Inhibit ? Up_Separation + 100 : Up_Separation); */
7   }
8   bool Non_Crossing_Biased_Climb() {
9       int upward_preferred;
10      int upward_crossing_situation;
11      bool result;
12      upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
13      if (upward_preferred) {
14          result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM())));
15      } else {
16          result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM());
17      }
18      return result;
19  }
20  bool Non_Crossing_Biased_Descend() {
21      int upward_preferred;
22      int upward_crossing_situation;
23      bool result;
24      upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
25      if (upward_preferred) {
26          result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Down_Separation >= ALIM());
27      }  else {
28          result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
29      }
30      return result;
31  }
32  bool Own_Below_Threat() {
33      return (Own_Tracked_Alt < Other_Tracked_Alt);
34  }
35  bool Own_Above_Threat() {
36      return (Other_Tracked_Alt < Own_Tracked_Alt);
37  }
38  int alt_sep_test() {
39      enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep > MAXALTDIFF);
40      tcas_equipped = Other_Capability == TCAS_TA;
41      intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
42      alt_sep = UNRESOLVED;
43      if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)) {
44          need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
45          need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
46          if (need_upward_RA && need_downward_RA)
47              alt_sep = UNRESOLVED;
48          else if (need_upward_RA)
49              alt_sep = UPWARD_RA;
50          else if (need_downward_RA)
51              alt_sep = DOWNWARD_RA;
52          else
53              alt_sep = UNRESOLVED;
54      }
55      return alt_sep;
56  }
57  int main() {
58          assert(alt_sep_test() != UPWARD_RA);
59  }
```

Figure 4.11: Sample TCAS program. The mutation is in line 5. The correct version is shown in line 6

Minimization of counterexample length [3,4] as well as semantic minimization [5] is, therefore, an ongoing area of research. Apart from just minimization, many techniques have been developed to analyze counterexamples effectively and extract useful information that can be used to help the programmer in finding the bug in the program.

Ball *et al.* localize faulty statements in the context of model checking by comparing feasible error traces to "correct" traces (i.e., traces that terminate correctly) [16]. They call a statement relevant if it does not occur on a correct trace. In contrast, our analysis based on aberrant statements, does not require a correct trace.

Jose *et al.* initiated a line of formula-based fault localization works [15]. Given a feasible error trace, they construct a Boolean *unsatisfiable trace formula* consisting of an input $I$ that triggers the error, the trace formula $\pi$, and the negated error condition $E$:

$$I \wedge \pi \wedge \neg E$$

Passing the formula to a partial MAX-SAT solver, where $\phi$ and $\neg E$ are considered as so-called hard clauses, the solver returns a maximal number of clauses (from $\pi$) that can still be satisfied. All other clauses are then considered relevant because removing them all would make the trace formula satisfiable, which corresponds to that the error condition may hold. Our approach based on aberrant trace elements works on first-order formulas, and we query the theory solver several times with smaller formulas. When there are several possible error causes, the partial MAX-SAT solver will choose one with the fewest clauses. The result of our algorithm is unique.

Ermis *et al.* adapted the idea to so-called *error invariants* [21] which are defined similarly to Craig interpolants. Error invariants are predicates, one between each statement, that overapproximates the reachable states but are still strong enough to imply the error condition. If a given predicate is an error invariant at positions $i$ and $j > i$, the authors say that the statements in between are irrelevant and can be dropped, or equivalently, replaced by a `skip` statement. Given a set of predicates, the authors find a smallest such covering of the error trace with error invariants. Error invariants are in general not unique, and thus two implementations may differ in their output. As another difference, the authors used the weakest precondition function WP() of the trace for $I$ in the formula above. We use the precondition function PRE() which allows us to handle traces that contain nondeterminism. This is a general limitation of trace analysis techniques based on the unsatisfiable trace formula because the formula needs to be unsatisfiable. The error invariants approach was later extended to "flow-sensitive" analysis that supports error conditions inside `if-endif` blocks [22], to concolic testing [23], and to concurrent traces [24].

Schäf *et al.* generalize the idea of error invariants to *error invariant automata* [25]. The locations are annotated with error invariants. In its basic form, the automaton accepts a single feasible error trace where no error invariants repeat, i.e., the trace where all irrelevant statements have been dropped. The automata can then be mapped back to the original program they were con-

structed for and be interpreted as a smaller version of this program. In another view, only those statements that remain in the error invariant automaton are relevant.

Chao *et al.* locate portions of an error trace where an error may reside. The goal is to find out how the error propagates through the error trace and triggers a failure [6]. They also use a path-based weakest precondition computation algorithm. They find a minimal set of assigning statements in the error trace that can make the trace infeasible. This approach, however, is not robust since it does not take into account the order of the assigning statements in the error trace.

**Other approaches.** In the following, we discuss some other techniques that are not so closely related to our approach of trace based fault localization.

Fault localization techniques that are not based on single error trace analysis include *spectrum-based* techniques. A program spectrum is a measurement of run-time behavior, such as code coverage [26]. Spectrum based techniques compare passing and failing executions obtained from a test suite (usually satisfying a certain coverage criterion) and then rank the statements according to their relevance [27–30]. *Mutation-based* techniques extend this idea by also taking into account how often a statement mutation (i.e., replacing a statement by another statement) affects the test outcome [31, 32]. We also replace statements, but we do that symbolically, contrary to performing a concrete replacement and re-execution of the program. The above mentioned spectrum and mutation based ranking techniques have been evaluated in a large-scale comparison in [33].

*Slice-based* techniques delete parts from a program such that it still retains the original behavior w.r.t. a certain specification [34]. A program statement is relevant for the error if it occurs in the slice. Static slicing only uses the source code and accounts for all possible program executions. Dynamic slicing focuses on one execution for a specific input. A dynamic slice will contain all the statements that may affect the the values in the slicing criterion for a specific execution. Zhang *et al.* evaluated several dynamic slicing techniques for the purpose of fault localization [35]. Since slicing techniques often only remove statements that are not relevant for the error, they can be used as a preprocessing step to other, more precise and relatively expensive fault localization techniques like the one we presented using aberrant trace elements. *Program state-based* techniques, as the name suggests, compare the program states of several executions, e.g., via delta debugging [36]. Delta debugging isolate relevant variables by systematically narrowing the state difference between a passing and a failing run [17, 20]

While reporting relevant statements to the developer can be a tremendous help in debugging, the even better solution would be to automatically repair the program in a problem closely related to fault localization called *program repair*. Several recent approaches to this challenging task have been proposed [37, 38]. At first glance, our fault localization technique based on aberrant statements, where we can replace an aberrant statement by another one, may seem similar, but we do not have a guarantee that the new assignment removes the bug for all executions (nor that it does not introduce new bugs).

## 4.2 Detecting Unvalidated Input

### 4.2.1 Introduction

"All input is evil", writes Michael Howard in his book Writing Secure Code [39]. A programmer should always make sure that the data received by the program as an input is reasonable. Any input received by a program from an untrusted source is a potential target for an attack (in this context, an ordinary user or an external library is an untrusted source). Hackers look at all sources of input to the program and attempt to pass in corrupted data of every type they can imagine. If the program crashes or otherwise behaves in a way it was not supposed to, the attacker tries to find a way to exploit the problem. Unvalidated input exploits have been used to take control of operating systems, steal data, corrupt user's disks, and more. The issue of input validation is even more difficult yet critical in web based applications and services because of the presence of a large number of entry points. Most common security related issues in web applications is due to the failure to properly validate the input from the client. Issues like SQL injection, file system attacks and buffer overflows.

In the context of an error trace, we will formally define the subclass of security vulnerabilities where a user can directly influence the reachability of an undesired program state. An undesired state here could mean a state which can lead to consequences like, denial of service (users may be denied access to legitimate services), degradation of performance (performance can be so poor that the system is not usable), disclosure of information (the user gains unauthorized access to protected information) or modification of data (the user modified information in an unauthorized manner).

For the sake of consistency with the rest of the thesis, we will call this undesired state an error state. We will discuss how to detect the vulnerability that arises when the reachability of an error state can be controlled by a program's input value. Consider the program in figure 4.12 that demonstrates this scenario.

```
1:   int get_location(void);
2:   int get_value(void);
3:
4:   void main(void) {
5:       int a[10];
6:       int loc, v;
7:       loc = get_location();    //Input value from the user
8:       v = get_value();         //Input value from the user
9:       a[loc] = v;
10:   }
```

Figure 4.12: Simple example of a buffer overflow

A buffer overflow can be triggered if the user inputs the value 11 for the

variable `loc`. Moreover, notice that for the value 11, the user has a guarantee that the program will end up in an error state. Now, consider the program in Figure 4.13, where the user no longer has this guarantee because of a condition on the variable `s` which is true only if the values assigned to `loc` and `v` are validated by the function `validate_input()`.

```
1:   int get_location(void);
2:   int get_value(void);
3:   bool validate_input(int l, int v);
4:
5:   void main(void) {
6:       int a[10];
7:       int loc, v;
8:       loc = get_location();    //Input value from the user
9:       v = get_value();         //Input value from the user
10:      s = validate_input(loc,v);
11:      if (s) {
12:           a[loc] = v;
13:       }
14:  }
```

Figure 4.13: A check on input variables avoid buffer overflow

From here onwards, we say that the program has an *unvalidated input* if there is a location where the program gets an input and, for a certain input value, the user has a guarantee that the program will end up in an error state.

### 4.2.2 Defining Unvalidated Inputs

We detect the presence of an unvalidated input in the program with the help of a feasible error trace. A program has an unvalidated input if there exists a feasible error trace where the following three properties are satisfied:

1. There is some statement $st_i$ in the error trace which assigns a value taken from the user to a variable.

2. There is some input value such that continuing the execution of the trace from $st_i$, we definitely reach the error.

3. There is some input value such that continuing the execution of the trace from $\pi[i]$, we do not reach the error.

These three properties precisely capture our notion of the existence of an unvalidated input in a program. "Definitely" in property 2 provides the guarantee to the attacker that for the right value, the program execution will always end up in an error state. In property 3, existence of one or more values such that the program execution does reach the error state makes it harder for testing

based approaches to detect the bug and makes it possible for the bug to slip into production systems.

**Definition 5** (Unvalidated Input)**.** *Let $\pi$ be a feasible error trace of length $n$ and $\pi[i]$ be an assigning statement that assigns a value to the variable $x$. The program has an unvalidated input if $\pi[i]$ assigns an input value taken from the user and there exists an execution $s_0, \ldots, s_n$ of $\pi$ and some values $v$ and $w$ such that, if we start in the state $s_i$, every execution of the trace $\langle x := v, \pi[i+1, n]\rangle$ is blocking, and additionally, the trace $\langle x := w, \pi[i+1, n]\rangle$ has no blocking execution.*

In the above definition, the existence of an execution $s_0, \ldots, s_n$ and a value $v$, such that all executions of the trace $\langle x := v; \pi[i+1, n]\rangle$ are blocking if we start in the state $s_i$ is the definition of aberrance. What is new here is the existence of a value $w$ such that no execution is blocking for the trace $\langle x := w; \pi[i+1, n]\rangle$. We call such trace elements *spurring*, since for a value $w$ and a state $s_i$, assigning $w$ to `x` unstoppably spurs all the executions of the trace $\langle x := w; \pi[i+1, n]\rangle$ towards the error. We can formally define a *spurring trace element* as:

**Definition 6** (Spurring Trace Element)**.** *Let $\pi$ be a feasible trace of length $n$ with $\pi[i]$ being an assigning statement that assigns a new value to some variable $x$. The statement $\pi[i]$ is a* spurring trace element *in $\pi$ if there exists an execution $s_0, \ldots, s_n$ of $\pi$ and some value $w$ such that no execution of the trace $\langle x := w, \pi[i+1, n]\rangle$ starting in $s_i$ is blocking.*

Armed with the formal definitions of aberrant and spurring statements and an invalidated input, we can simply write "*A program has an unvalidated input if in one of its feasible error traces, an assigning statement which is taking its value from a user is both an aberrant and a spurring trace element.* "

### 4.2.3 Algorithm

Given a feasible error trace, we explain how we can use Algorithm 1 to detect the existence of an unvalidated input in the program. A user input is modeled as a nondeterministic `havoc(x)` statement in the error trace, where `x` is the variable which is being assigned the user input value. To detect the existence of an unvalidated input, we simply have to check if the last aberrant `havoc` statement in the error trace is modeled from a user input statement. The procedure is shown in Algorithm 2. We use Algorithm 1 to find the aberrant statements in the error trace and check if the last `havoc` statement represents a user input.

#### Algorithm correctness

We now show that Algorithm 2 accurately detects if there is an unvalidated input in the program in accordance with Definition 5.

**Theorem 2.** *Let $\pi$ be a feasible error trace of length $n$ and $\pi[i]$ be an over-approximated user input statement of the form `havoc(x)` at position $i$ that assigns a non-deterministic value to the variable $x$. If $\pi[i]$ is the last aberrant `havoc` statement in the error trace, then it is also spurring.*

**Algorithm 2:** Unvalidated input detection in a program

**Input:** A feasible error trace $\pi$ of length $n$
**Output:** Boolean value indicating the presence of an unvalidated input

```
1 trace ← error trace π;
2 aberrPos ← aberrance(trace);
3 for j = aberrPos.length() - 1 to 0 do
4 |   if trace[aberrPos[j]] is not a havoc then continue;
5 |   if trace[aberrPos[j]] is a user input then return true;
6 |   return false;
7 end
```

*Proof.* Let $\pi[i]$ be a havoc statement of the form `havoc(x)` which assigns a nondeterministic value to the variable `x` at position $i$. Since $\pi$ is a feasible error trace, we know that there exists an execution $s_0, \ldots, s_n$ and some value $w$ such that, if we assign $w$ to `x`, the trace $\langle x := w; \pi[i+1, n]\rangle$ starting in the state $s_i$ has a non-blocking execution. Furthermore, since $\pi[i]$ is the last aberrant `havoc` statement, no execution of the trace $\langle x := w; \pi[i+1, n]\rangle$ will be blocking. Hence, $\pi[i]$ is a spurring statement. □

### 4.2.4 Discussion

Please note that Algorithm 2 takes as input a feasible error trace $\pi$ where the branches are encoded as conglomerates as explained in Section 4.1.2. A simple check on an input value is not necessarily a validation of that input. By validating an input, we mean that a user cannot influence the reachability of an error state by changing the input value. There will be some cases where there seems to be a check on the value taken by a user but our algorithm will still detect the presence of an unvalidated input. Consider the example in Figure 4.14.

```
1:   int get_from_user(void);
2:   void main(void) {
3:     int x = 1;
4:     int y = get_from_user();      // Unvalidated Input
5:     if (y < 10) {
6:       y++;
7:     } else {
8:       x = 2;
9:     }
10:    assert (x != 1);
11:  }
```

Figure 4.14: Unvalidated input even tough there is a check on the variable `y`

The variable y gets it's value from the user. There seems to be a check on the value of the variable y and it does seem that the input value is validated. But actually the user can still control the reachability of the error by changing the input value. This example helps bring to light an important property of unvalidated inputs in a program. Input validation is about validating that the input value cannot directly control the reachability of the error state. Let's get back to our motivating example of a buffer-overflow. In the example in Figure

```
1:   int get_location(void);
2:   int get_value(void);
3
4:   void main(void) {
5:     int array[10];
6:     int location,value, r;
7:     value = get_value();           // from user
8:     location = get_location();     // from user
9:     if(location >=0 && location <=10){
10:      array[location] = value;
11:    }
12:  }
```

Figure 4.15: Unvalidated input even tough there is a check on the variable y

4.15, a buffer-overflow can occur on line 10 because the programmer made a mistake. The `if` condition should have `location < 10` instead of `location <=10`. The algorithm tells us that there is a possibility of a buffer overflow. But the problem is more serious because it can be controlled by a user input. So it is a security issue and must be taken more seriously.

In case the error trace contains more then one unvalidated inputs, algorithm 2 cannot detect all unvalidated inputs in the trace at once. But we can say that there is atleast one unvalidated input in the program which also is a valuable security information for the user.

### 4.2.5 Experiments

We apply our technique to three (now fixed) faults in the Linux device drivers. All of these faults were classified as a security vulnerability and were present in the official Linux kernel repository[2] at some point in time and were fixed as soon as they were found. We apply our technique to simplified programs that model the faults with all the unnecessary kernel code removed. The simplified programs were provided to us by the people at the Linux Driver Verification[3](LDV) project at the Russian Academy of Sciences. They were also responsible for finding and reporting the faults.

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/
[3] linuxtesting.org

```
[L7]                -   int *p;
[L19]               #   int state = 0;
[L60]               -   struct inode *dir;
[L62]               %   dir = allocate_data()
[L64]  CALL         #   ufs_mkdir(dir)
[L52]  CALL         -   lock_ufs()
[L22]  COND FALSE   -   !(state==1)
[L23]  RET          #   state = 1
[L52]               -   lock_ufs()
[L53]  FCALL        -   inode_inc_link_count()
[L54]  CALL         #   ufs_new_inode(dir)
[L34]  COND FALSE   -   !(!dir)
[L37]  CALL, EXPR   -   dir->inode
[L37]  RET, EXPR    #   dir->inode
[L37]  COND FALSE   -   !(!dir->inode)
[L40]  FCALL        -   f()
[L40]  COND TRUE    -   f()
[L41]  CALL         -   dir->a=0
[L41]  RET          -   dir->a=0
[L47]  CALL         -   lock_ufs()
[L22]  COND TRUE    -   state==1
[L22]               -   __VERIFIER_error()
```

Figure 4.16: Error trace for doublelock in the UFS program

We briefly provide the context of the problem along with the commit history of the patches that fixed the faults. We also explain how our technique is able to detect the presensce of an unvalidated input with the help of an error trace generated by the software model checker Ultimate Automizer [1].

**Experimental Setup.** We will use our implementation in Ultimate Automizer (see Chapter 5) to detect the presence of unvalidated inputs in all the three examples. The output contains four columns. The first column contains the line number, the second column contains information about the type of the statement, the third column contains the aberrance information and the fourth column the actual statement. We are interested in the third column. `#` represents that the statement is aberrant. `%` means that the statement was over-approximated and the over-approximated statement is aberrant. If the last statement marked with `%` is an assignment, where the value is coming from the user (or an outside source), then this means that there is an unvalidated input in the program.

### Doublelock in Unix File System

In the Unix File System (UFS), an unvalidated input could introduce an unavoidable double-lock. The fault was introduced by a patch (commit history in Appendix B.1.1) which merged two locks, since they were generally locked and unlocked at the same time. But this patch introduced a bug where a double-lock could be caused in the functions `ufs_new_inode()` and `ufs_free_inode()` which depended only on the initial input data provided by `allocate_data()`.

The problem was eventually found by the LDV project and fixed. We ran our algorithm on an error trace generated for the simplified program (Appendix

```
[L31]                #   static int ldv_spin_bgc_lock = 1;
[L101]               %   struct grgpio_priv *priv = get_grgpio_priv();
[L102]               %   unsigned int irq = get_irq();
[L103]  CALL         #   grgpio_irq_unmap(priv, irq)
[L78]                -   int index;
[L79]                -   int i;
[L80]                -   struct grgpio_lirq *lirq;
[L81]   CALL, EXPR   #   priv->ngpio
[L81]   RET, EXPR    #   priv->ngpio
[L81]                #   int ngpio = priv->ngpio;
[L83]   CALL         #   ldv_spin_lock_bgc_lock(&priv->bgc_lock)
[L35]   COND FALSE   -   !(ldv_assert(ldv_spin_bgc_lock == 1))
[L36]   RET          #   ldv_spin_bgc_lock = 2
[L83]                #   ldv_spin_lock_bgc_lock(&priv->bgc_lock)
[L85]                -   index = -1
[L86]                #   i = 0
[L86]   COND TRUE    -   i < ngpio
[L87]                #   lirq = &priv->lirqs[i]
[L88]   CALL, EXPR   -   lirq->irq
[L88]   RET, EXPR    #   lirq->irq
[L88]   COND TRUE    -   lirq->irq == irq
[L89]   CALL         #   grgpio_set_imask(priv, i, 0)
[L63]   EXPR, FCALL  -   get_mask(priv, offset)
[L63]                -   unsigned long mask = get_mask(priv, offset);
[L66]   CALL         #   ldv_spin_lock_bgc_lock(&priv->bgc_lock)
[L35]   COND TRUE    -   ldv_assert(ldv_spin_bgc_lock == 1)
[L35]                -   ldv_assert(ldv_spin_bgc_lock == 1)
```

Figure 4.17: Error Trace for the simplified Aeroflex GRGPIO driver

B.1.2) and were able to confirm the existence of an unvalidated input which could control the reachability to an error state (double-lock in this case). In the program, the input statement where the variable `dir` gets its value from `allocate_data()` is modeled as `havoc(dir)` in our setting and is the last aberrant `havoc` statement. The trace along with the aberrance information is shown in Figure 4.16.

### Doublelock in Aeroflex Gaisler GRGPIO

In one of the drivers for Aeroflex Gaisler GRGPIO, a double-lock could be caused in the method `grgpio_irq_unmap()` where it locks the spinlock based on the value of the variable `priv` which gets it's value from an outside source. The method then called `grgpio_set_imask(priv, i, 0)` which unconditionally locked the spinlock by itself. This could lead to a double lock. The link to the merged patch that fixed the problem is included in Appendix B.2.1. The link to the program that models the problem is given in Appendix B.2.2 and the link to original driver is included in Appendix B.2.3.

The error trace can be seen in Figure 4.17. In the error trace, the variable `irq` gets a value from an outside source (`get_irq()` in the program) and is modeled as `havoc(irq)` in our setting. `havoc(irq)` is the last aberrant `havoc` in the error trace and hence our algorithm detect the presence of an unvalidated input.

```
[L30]               *  static int ldv_mutex_ctx_lock = 1;
[L52]               -  struct imon_context *context;
[L103]  EXPR, FCALL -  get_interface()
[L103]              -  struct usb_interface *interface = get_interface();
[L104]  CALL, EXPR  *  imon_probe(interface)
[L55]               *  int retval = 0;
[L56]               -  int lirc_minor;
[L57]   CALL, EXPR  *  kzalloc(sizeof(struct imon_context))
[L14]   EXPR, FCALL *  calloc(1, size)
[L14]   EXPR, FCALL *  calloc(1, size)
[L14]   RET         *  return calloc(1, size);
[L57]   EXPR        *  kzalloc(sizeof(struct imon_context))
[L57]               *  context = kzalloc(sizeof(struct imon_context))
[L58]   COND FALSE  -  !(!context)
[L61]   CALL        #  ldv_mutex_lock_ctx_lock(&interface->ctx_lock)
[L34]   COND FALSE  -  !(ldv_assert(ldv_mutex_ctx_lock == 1))
[L35]   RET         *  ldv_mutex_ctx_lock = 2
[L61]               #  ldv_mutex_lock_ctx_lock(&interface->ctx_lock)
[L63]               %  lirc_minor = lirc_register_driver()
[L64]   COND FALSE  -  !(lirc_minor < 0)
[L91]   RET         *  return retval;
[L104]  EXPR        *  imon_probe(interface)
[L104]              *  int ret = imon_probe(interface);
[L105]  COND TRUE   -  ret==0
[L106]  CALL        -  imon_disconnect(interface)
[L95]   CALL        -  ldv_mutex_lock_ctx_lock(&interface->ctx_lock)
[L34]   COND TRUE   -  ldv_assert(ldv_mutex_ctx_lock == 1)
[L34]               -  ldv_assert(ldv_mutex_ctx_lock == 1)
```

Figure 4.18: Error trace for simplified program dependent on the LIRC driver

#### imon_probe() exits with mutex acquired

A fault was introduced by a patch (commit link can be seen in Appendix B.3.1) when it wrongly removed `mutex_unlock()` and consequently the method `imon_probe()` can exit with the lock acquired if `lirc_register_driver` fails. This could therefore cause an unavoidable double lock based on the value returned by `lirc_register_driver()`. The attacker can control the reachability of the error state (double lock in this case) by causing the lirc register driver to fail, which obviously is a security vulnerability. LIRC is an open source package that allows users to receive and send infrared signals with a Linux-based computer system. The merged patch by the Linux Driver Verification Project fixed the problem (Appendix B.3.2). The programs that models the problem with all the unnecessary code removed can be seen in Appendix B.3.3. The link to the Linux driver where the problem was introduced is given in Appendix B.3.4.

The error trace can be seen in Figure 4.18. The variable `lirc_minor` gets it's value from the driver (`lirc_register_driver()` in the program) and is modeled as `havoc(lirc_minor)` in our setting. `havoc(lirc_minor` is the last aberrant `havoc` in the trace.

### 4.2.6 Related Work

Input validation is widely accepted as one of the most common causes of many vulnerabilities in web based services and hence taken very seriously in the web development community. Buffer overflows, injection attacks, DoS attacks, mem-

ory leakage, identity theft, privacy compromise and information disclosure are some of the more serious problems that can be cause be incorrect input validation. Software systems nowadays work in an increasingly complex and connected environment where not only there are many sources of inputs to the program, but there are also a wide range of sources that might be passing the input. An input can come from a user or a database or another program or a network and must be validated as early as possible in the data flow.

Programmers use a number of techniques to check the *syntactic* (syntax) and *semantic* (value) correctness of the input values. These techniques are usually testing based (for example, [40]). The idea to analyze the reachability of an error state controlled by an input value using error traces produced by a verification tool (to the extent of our knowledge) is new. Nonetheless, the importance of applying formal methods to analyze security vulnerability of software systems has been recognized for quiet some time [41]. Denning presented a mathematical framework to examine the information flow in the program [42]. The aim in his work was to come up with a formal analysis technique that can guarantee *secure information flow* in the program. A lot of work has been done in formally analyzing secure information in a program [43–48]. We however are not concerned with secure information flow in the program, where the goal is to regulate the dissemination of information among objects throughout the systems and guarantee the absence of illicit information leakages through program execution. Our goal mainly is to make sure that the input data received by the program cannot control the reachability of the error because a secure information flow through the program can still cause the program to fail.

## 4.3 Verification Of Open Programs

The use of static verifiers for debugging is still not very popular among common developers mainly due to the large number of uninteresting warnings generated by the verifier. Since verifying program correctness is, in general, undecidable, false warnings are inevitable no matter how sophisticated is the verification tool. This problem is especially more visible in *open* programs (programs that expose a set of external libraries or API methods). Most verifiers aim to verify that a program does not fail assertions under all possible feasible executions of the program. This approach works well when the program is *closed*, i.e. its execution starts from a well-defined state, and external library methods are included or their specifications are accurately defined. In the case of open programs with an unconstrained environment, program verifiers generate a lot of uninteresting warnings in the absence of precise environment specifications. Such warnings can overwhelm a user and deviate their attention from the real bug. Using program verifiers in such circumstances thus require extensive initial investment in time and effort to model the external libraries.

```
1  var m:[int]int;
2  procedure FooBar()
3    modifies m;
4  {
5      var w: int;
6      call w := env1(); // Demonic Environment
7      // assume w >= 1;
8      assert w != NULL;
9     m[w] := 2;
10 }
11 procedure Baz(z:int)
12   modifies m;
13 {
14     assert z != NULL; // failure due to true bug
15    m[z] := 4;
16 }
17 // Entry point
18 procedure Foo()
19   modifies m;
20 {
21     call FooBar();
22     call Baz(NULL); // TRUE BUG
23 }
24 // Environment
25 procedure env1() returns ( r : int );
```

Figure 4.19: An example of an open program

Program in Figure 4.19 shows an example of a program with unconstrained environment. This example is a shortened version of the motivating example presented in [49]. The program has three procedures `Foo`, `FooBar`, `Baz` and one external library procedure `env1`. The variables in the program can be scalars

(of type int) or arrays (e.g. m) that map int to int. `Foo` is the entry position of the program. The return value of the environment procedure `env1` is the source of *unknown* or unconstrained value in the program. Most verifiers are bound to return assertion failures for each assertion in the program. This is due to the fact that the assertion in line 8 is an assertion over an unknown value and verifiers tend to be conservative (over-approximate) in the face of unknowns. Such tendency of verifiers force them to see every unknown environment as *demonic*. Which in this case will lead to a warning of a possible assertion failure at line 8.

In this section, we present a technique to provide only high-quality warnings for open programs. Warnings that will help the programmer in finding and fixing a bug without the overhead of modeling external libraries and APIs. For an assertion and a feasible error trace that leads to the assertion violation, we will use aberrance analysis on the feasible error trace to judge the quality of the assertion failure and whether it should be reported to the user.

### 4.3.1 Using Aberrance To Reduce Uninteresting Warnings

Given a program with an unconstrained environment, our goal is to only report *interesting* warnings (assertion failures). Interesting warnings, here, are in the context of program debugging and will help the user in finding and fixing the bug. Intuitively, a warning is interesting if it is not dependent only on a value coming from an external library or the environment. For the program in Figure 4.20, `assert(y!=0)` is an assertion on a variable that is taking it's value from the environment. The failure of the assertion `assert(z!=10)` points to a possible buggy initialization either to the variable `x` in line 1 or a buggy assignment to the variable `z` at line 4. From this point of you, reporting the assertion failure at line 5 is more interesting to the user then the one at line 3, since reporting this assertion failure can help the user in fixing the bug.

```
1:   x = 5;
2:   y = external_library();
3:   assert (y != 0);
4:   z = x + 5;
5:   assert (z != 10);
```

Figure 4.20: A program with two assertions. Reporting the second failure assertion is more useful from debugging perspective

For a feasible error trace $\pi_1$ whose execution leads to a failure of the assertion $\alpha_1$, we can use aberrant trace elements in $\pi_1$ to determine if the assertion failure is interesting enough to be reported to the user. The assignment of a value coming in from the environment/external library is modeled as a `haovc` statement in our setting. For the program in Figure 4.20, we have the follow-

ing feasible error trace, $\pi_1$, for which there exists an execution such that the assertion `assert(y!=0)` is violated.

```
x = 5  havoc(y)  y == 0
```

The only aberrant statement in $\pi_1$ is an assignment where the value is coming from an external library. From a debugging point of you, the only fix for the programmer is to modify this assignment. Assuming that `external_library()` is correct, this assertion violation is of no use to the user and will be an uninteresting warning.

For the assertion `assert(z!=10)`, we have the following feasible error trace $\pi_2$.

```
x = 5  havoc(y)  y != 0  z = x+5  z == 10
```

In $\pi_2$, the existence of aberrant statements other then just the assignment to a variable from an external library points to multiple fixes in the program that can fix the error trace. Hence this assertion failure is more interesting to the user because it provides the programmer with options for possible fixes that can avoid the assertion failure.

We want to suppress those assertion violations for which, in the corresponding feasible error trace, only the assignments that are taking it's value from the environment (over-approximated as `havoc` statements in our setting) are aberrant. This is because we want to suppress those warnings that are only dependent on the external libraries or the environment.

If all the aberrant statements in an error trace are taking their value from the environment, then we say that the trace and the assertion violation is *angelically safe* and should not be reported to the user. Otherwise, we call the trace and the assertion violation *angelically unsafe*.

### 4.3.2 Using A Scoring Function To Rank Warnings

In some programs, a user might not want to strictly classify an assertion violation as angelically safe or unsafe. Consider, for example, the case when a program contain statements that modifies the values returned by an external library and the assertion is on the modified value. The assertion is dependent on the environment, but not directly. There is a chance that the modification is buggy and that is the cause for the assertion violation. Or consider a case where, the program contains many statements that assign a value to a variable taken from the environment. In this case, the probability that there is a buggy modification in the trace is statistically low and vice versa. Strict classification of an assertion failure might not be the most helpful solution to the programmer in such scenarios.

We, therefore, use a scoring function that assigns a score to an error trace based on the ratio between aberrant assignment statements and aberrant `havoc` statements (that model the environment). Consider a very simple program in Figure 4.21. In this program, the assertion `assert(y > 0)` is indirectly

dependent on the value returned by the environment. After getting the value from the environment, the assignment statement `y=x-10` modifies it. This can be a buggy modification.

```
1:  x = env();      // environment
2:  y = x - 10;
3:  assert (y > 0);
```

Figure 4.21: A program where it is hard to strictly classify an assertion failure

The score depends on the number of aberrant assignment statements and the number of aberrant `havoc` statements. In a trace, where only `havoc` statements are aberrant, the scoring function returns the minimum score and when all the aberrant statements are deterministic assignment statements, then it returns the maximum score.

$$score := \frac{number\ of\ aberrant\ assignment\ statements}{total\ number\ of\ aberrant\ statements}$$

For the case when all the aberrant statements in a trace are deterministic assignment statements, we will get the highest score 1. We get a score of zero when only `havoc` statements are aberrant in a trace. For all the cases in between, we get a score between zero and one based on the number of aberrant havoc and assignment statements. For the program in Figure 4.21, with the following error trace, we get a ranking score of 0.5 .

```
havoc(x) y = x - 10 y <= 0
```

The assertion can be violated due to the value returned by the environment or because of a buggy modification of the environment value.

For the program in Figure 4.20, the assertion failure at line 3 (`assert(y!=0)`) will have a score of 0 since in the feasible error trace that leads to the assertion failure, the only aberrant statement is a `havoc` statement that models the assignment of a value from the environment. The assertion failure at line 5 will have a score of $2/3 = 0.6$.

Consider the identical programs in Figure 4.22. In the program code in Figure 4.22b, the variable `x` gets its value from the external library, and then modified two times. A variable `p` is initialized at line 4, which is then modified in line 5. The probability that the assertion failure at line 6 is caused by a faulty assignment/modification is higher then in program in Figure 4.22a. From a debugging perspective, the assertion failure in the second program must be scored higher then the program in Figure 4.22a.

### Discussion

We presented a ranking scheme based on a simple statistical measure to rank error warnings for open programs. One of the major hurdles in the adoption

```
1:  x := external_library();
2:  y := external_library2();
3:  z := y + 1;
4:  p := external_library3();
5:  q := p + x;
6:  assert (q + z > 10);
```

(a) Program code with many environment dependent assignments

```
1:  x := external_library();
2:  y := x + 1;
3:  z := y + 1;
4:  p := 5;
5:  q := p + 1;
6:  assert (q + z > 10);
```

(b) Program code with assignments and modifications

```
havoc x | havoc y | z := y + 1 | havoc p | q := p + 1 | assume(q + z <= 10)
```

(c) Error trace for program in 4.22a

```
havoc x | y := x + 1 | z := y + 1 | p := 5 | q := p + 1 | assume(q + z <= 10)
```

(d) Error trace for program in 4.22b

Figure 4.22: Comparison between two identical programs

of static verification tools is the generation of a large number of error reports. This is because such tools often use approximation schemes which leads to false positives. These false positives can quickly damage user's confidence in such a tool by hiding real error warnings amidst the false ones. In fact, tools that effectively find errors can have as much as 30-100% false positives [50]. Ranking schemes based on statistical models, that rank error warnings would be a step in the right direction. These ranking schemes should be able to assign high confidence score to real or true error warnings and a low score to false positives. Users tend to immediately discard the tool if the first few error warnings they see are false positives. Ranking real error warnings higher can help us avoid just that.

### 4.3.3 Related Work

Verifying program correctness can be challenging in the presence of an unconstrained environment. If a program contains calls to an external library or an API, the verifier tends to be conservative (over-approximate) and will report a warning if it can find a return value for the library call such that the assertion fails. Such warnings are not that interesting for a developer because they are not very useful for debugging. Ankush *et al.* called them *dumb warnings* and proposed a technique called *angelic verification* to filter out warnings that occur due to *demonic* assumptions about the *environment* by the verifier [49]. The verifier is constrained to report warnings only when no acceptable environment specification exits to prove the assertion. Saurabh *et al.* explore the possibility of automatically prioritizing warnings that result due to imprecisions in the *harness* [51]. A harness is used to specify the preconditions before calling an

external method. If such a harness is not specified correctly, the static analysis will report a warning in the program calling the external method. However, the warning does not reveal a bug in the program, rather a problem in the harness that was used to invoke the external method. Sam *et al.* [52] propose a parameterized framework for prioritizing assertion failures. However, their method is expensive and can be only applied intraprocedurally. The task of analyzing error warnings is also closely related to the work done by Dillig *et al.* [53] on classifying error reports in the context of *abductive inference* problem, where the goal is to find an explanatory hypothesis for a desired outcome. Their technique involve the computation of small relevant queries presented to the user that captures the information required to either validate or discard the error warning. The goal in *angelic non-determinism* [54] is to check if the assertion failures caused by non-deterministic code can be avoided by replacing it with deterministic code. Kremeneck [50] pose the problem of ranking error warnings as a typical classification problem. The classification system has to rank an error warning as either a real warning (true error) or a false positive. The ranking is then simply based on the confidence in the classification.

## 4.4 Over-Approximated Statements

Most programming languages have a rich syntax and provide standard libraries such as `math.h` with complicated functions. Software model checkers typically over-approximate such statements if they cannot be handled exactly. When a software model checker returns an error trace with over-approximated statements, the feasibility status of that error trace is *unknown*. The reason is that the error trace might only be feasible due to the over-approximation that was injected for the analysis. Accordingly, in such cases one cannot say if the safety property of the program is really violated or not.

Consider now the case that we have an over-approximated assignment statement which is *not* aberrant. This is equivalent to saying that there is no concrete assignment that makes the trace infeasible. From this fact we can conclude that this particular over-approximation did not affect the feasibility of the trace.

We can generalize this observation to multiple over-approximated statements: If all the over-approximated assignment statements in the error trace are not aberrant, then we can conclude that the original error trace is feasible and the program is unsafe.

We illustrate the idea with the following feasible error trace.

```
i = 0  x = 3.14159  y = sin(x)  assume(i == 0)
```

The following is a simple overapproximation that avoids the trigonometry.

```
i = 0  x = 3.14159  havoc(y)  assume(i == 0)
```

Assume that a software model checker has shown feasibility of this over-approximated error trace. Observe that in both traces only the first statement is aberrant. In particular, the third statement that was over-approximated is *not* aberrant. So we can infer that the original error trace must also be feasible.

To conclude, in certain cases we can use aberrance to improve a software model checker's result from "*unknown*" to "*unsafe*".

# Chapter 5

# Aberrance In Ultimate Automizer

We have implemented our algorithm to compute aberrant statements in feasible error traces (including error traces with branches) in Ultimate Automizer[1] [55] which is a toolchain in Ultimate[2] software analysis framework. Ultimate consists of several libraries and plugins that perform various steps of program analysis, for example, parsing source code or transforming program representation. Ultimate Automizer verifies program safety properties (e.g. validity of assert statements, validity of procedure contracts, reachability of an error function/error location or memory safety) based on an automata theoretic approach to software verification [1, 55–58]. The implementation allows us to test the application of aberrance in fault localization as described in Section 4.1, detection of unvalidated inputs as described in Section 4.2 and verification of open programs as described in Section 4.3.

## 5.1 Basic Aberrance Analysis

The algorithm to compute aberrant statements is implemented in the Ultimate Automizer tool chain and allows us to directly analyze any error trace generated by Ultimate Automizer for programs that violate a safety property. The basic implementation of the algorithm which analyze an error trace without taking into account the branch information can be activated in the settings for Automizer by selecting the option "SINGLE_TRACE" under the setting "Highlight relevant statements in error traces" as shown in Figure 5.1.

In the implementation, for convenience, we call it "single trace analysis" as opposed to "multi-trace analysis" where we take the branches in the error trace into account. Enabling the setting will run the aberrance analysis on the trace and while reporting the error trace, Ultimate Automizer will report

[1] https://ultimate.informatik.uni-freiburg.de/automizer/
[2] https://ultimate.informatik.uni-freiburg.de

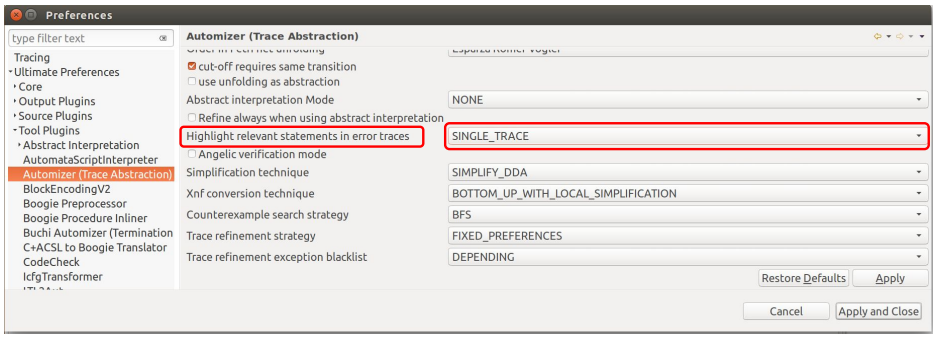



Figure 5.1: Setting to enable basic aberrance analysis on error traces in Ultimate Automizer

the aberrance status for each statement in the trace. With the setting enabled, the output will contain four columns. The first column shows the line number. The second column specifies the kind of element displayed in this line. The third column indicates if the statement is aberrant or not. The fourth column shows the statement. Figure something something shows the result of running an example program from the publicly available Ultimate repository. Figure 5.2 shows a snapshot for the example program.

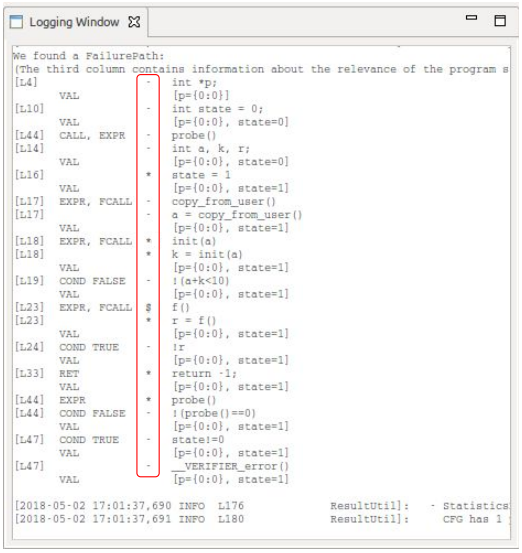



Figure 5.2: Output with the setting to compute aberrant statements enabled. The third column (marked in red) contains the aberrance information.

## 5.2 Aberrance On Traces With Branches

As described in Section 4.1.2, we also can take into account the branching information from the CFG of the program to improve precision in fault localization. In the implementation, for convenience, we call it multi-trace analysis. We get the branching information from the CFG which is available in Ultimate Automizer. Enabling the setting will replace the branches in an error trace with conglomerates. For aberrant conglomerates, the implementation will revert the conglomerate back to the original branch in the hope to find aberrant statements inside the branch. This process is recursive to deal with the error traces with nested branches.

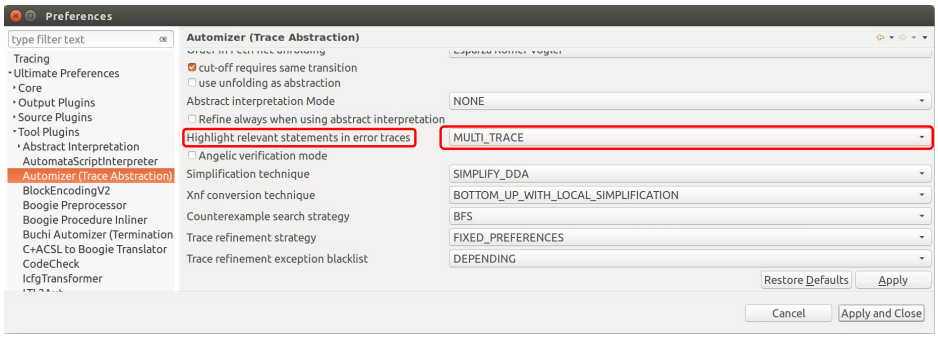



Figure 5.3: Setting to enable aberrance analysis for error traces with branches in Ultimate Automizer

## 5.3 Verification Of Open Programs

As described in Section 4.3, we can use the information from the aberrance analysis on a feasible error trace to suppress an error warning. In case of programs

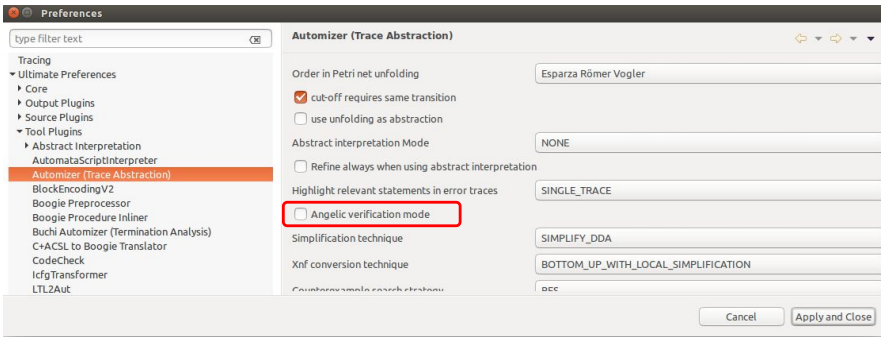



Figure 5.4: Setting to suppress angelically safe error traces from displaying in the output console

with multiple assertions, enabling the setting “Angelic Verification Mode” will not display those error traces where only the over-approximated assignment statements are aberrant. Enabling the setting will also calculate and “angelic score” for each trace as described in Section 4.3.2. The angelic score can be seen under “ErrorLocalizationStatistics” in the output console.

# Chapter 6

# Conclusion

Coming up with automatic techniques to extract value from error traces produced by software model checkers remain an important task. We presented a new approach to statically analyze error traces and discussed its application in fault localization, security, verification of open programs and increasing the output precision for software model checkers. The technique is based on finding those trace elements that can single-handedly make the trace infeasible. We presented an efficient algorithm to find such statements and prove its correctness.

In fault localization, our approach can help the programmer in finding simple bug fixes. We presented a modification in the encoding of the error traces to accommodate branches. The approached was tested on some faulty versions of TCAS programs from the Siemens test suite. We showed how our technique helped us in pin pointing the fault in one of the sample TCAS programs.

Another application we discussed was in security, where we presented a technique to detect the presence of unvalidated inputs in a program given a feasible error trace. We showed the applicability of our technique by testing it on real world security errors from the Linux device drivers.

Suppressing and later ranking error warnings in programs that expose external libraries and APIs is yet another interesting application of aberrant statements. Lastly, we showed that aberrant statements can be used to increase the precision of software model checkers for error traces where some statements are over-approximated.

The algorithm is implemented in a publicly available framework for program analysis, Ultimate, that helped us to test the applicability of our approach in fault localization and security on real world benchmarks with promising results.

# Bibliography


[1] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2013.

[2] Vincent Ribaud, Ciprian Teodorov, Zoé Drey, Luka Leroux, and Philippe Dhaussy. Techniques and Challenges for Trace Processing from a Model-Checking Perspective. In *International Joint Conferences on Computer, Information, Systems Sciences, & Engineering - CISSE 2014*, Proceedings of the International Joint Conferences on Computer, Information, Systems Sciences, & Engineering, Bridgeport, United States, December 2014. University of Bridgeport.

[3] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 31–45. Springer, 2004.

[4] Paul Gastin, Pierre Moro, and Marc Zeitoun. Minimization of counterexamples in SPIN. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2004.

[5] Alex Groce and Daniel Kroening. Making the most of BMC counterexamples. *Electr. Notes Theor. Comput. Sci.*, 119(2):67–81, 2005.

[6] Chao Wang, Zijiang Yang, Franjo Ivancic, and Aarti Gupta. Whodunit? causal analysis for counterexamples. In Susanne Graf and Wenhui Zhang, editors, *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006.*, volume 4218 of *Lecture Notes in Computer Science*, pages 82–95. Springer, 2006.

[7] Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. Predicate abstraction for program verification. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, chapter 15. Springer International Publishing, to appear in 2018.

[8] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] Daniel Dietsch, Matthias Heizmann, Betim Musa, Alexander Nutz, and Andreas Podelski. Craig vs. newton in software model checking. In *ESEC/SIGSOFT FSE*, pages 487–497. ACM, 2017.

[10] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View, Second Edition*. EATCS. Springer, 2016.

[11] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*, ICSE '94, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[12] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying safety-critical systems. In A. Min Tjoa and Volker Gruhn, editors, *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering 2001, Vienna, Austria, September 10-14, 2001*, pages 142–151. ACM, 2001.

[13] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking - History, Achievements, Perspectives*, volume 5000 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2008.

[14] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 1982.

[15] Manu Jose and Rupak Majumdar. Cause clue clauses: error localization using maximum satisfiability. In Mary W. Hall and David A. Padua, editors, *PLDI*, pages 437–446. ACM, 2011.

[16] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In Alex Aiken and Greg Morrisett, editors, *POPL*, pages 97–105. ACM, 2003.

[17] Holger Cleve and Andreas Zeller. Locating causes of program failures. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, pages 342–351. ACM, 2005.

[18] Alex Groce. Error explanation with distance metrics. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2004.

[19] Manu Jose and Rupak Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011.

[20] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*, pages 1–10. ACM, 2002.

[21] Evren Ermis, Martin Schäf, and Thomas Wies. Error invariants. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM*, volume 7436 of *LNCS*, pages 187–201. Springer, 2012.

[22] Jürgen Christ, Evren Ermis, Martin Schäf, and Thomas Wies. Flow-sensitive fault localization. In *VMCAI*, volume 7737 of *LNCS*, pages 189–208. Springer, 2013.

[23] Chanseok Oh, Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Concolic fault abstraction. In *SCAM*, pages 135–144. IEEE Computer Society, 2014.

[24] Andreas Holzer, Daniel Schwartz-Narbonne, Mitra Tabaei Befrouei, Georg Weissenbacher, and Thomas Wies. Error invariants for concurrent traces. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM*, volume 9995 of *LNCS*, pages 370–387, 2016.

[25] Martin Schäf, Daniel Schwartz-Narbonne, and Thomas Wies. Explaining inconsistent code. In *ESEC/SIGSOFT FSE*, pages 521–531. ACM, 2013.

[26] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Softw. Test., Verif. Reliab.*, 10(3):171–194, 2000.

[27] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[28] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE*, pages 88–99. IEEE Computer Society, 2009.

[29] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 273–282. ACM, 2005.

[30] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In Will Tracz, Michal Young, and Jeff Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 467–477. ACM, 2002.

[31] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *ICST*, pages 153–162. IEEE Computer Society, 2014.

[32] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Softw. Test., Verif. Reliab.*, 25(5-7):605–628, 2015.

[33] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 609–620. IEEE / ACM, 2017.

[34] Mark David Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD thesis, Ann Arbor, MI, USA, 1979. AAI8007856.

[35] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. Experimental evaluation of using dynamic slices for fault location. In Clinton Jeffery, Jong-Deok Choi, and Raimondas Lencevicius, editors, *Proceedings of the Sixth International Workshop on Automated Debugging, AADEBUG 2005, Monterey, California, USA, September 19-21, 2005*, pages 33–42. ACM, 2005.

[36] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In Stephan Diehl, editor, *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*, volume 2269 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2001.

[37] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *CAV (2)*, volume 9780 of *LNCS*, pages 383–401. Springer, 2016.

[38] Roopsha Samanta, Oswaldo Olivo, and E. Allen Emerson. Cost-aware automatic program repair. In *SAS*, volume 8723 of *LNCS*, pages 268–284. Springer, 2014.

[39] Michael Howard and David E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002.

[40] Jane Huffman Hayes and Jeff Offutt. Input validation analysis and testing. *Empirical Software Engineering*, 11(4):493–522, 2006.

[41] David Byers, Shanai Ardi, Nahid Shahmehri, and Claudiu Duma. Modeling software vulnerabilitieswith vulnerability cause graphs. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 411–422. IEEE Computer Society, 2006.

[42] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[43] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[44] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Commun. ACM*, 15(3):157–170, 1972.

[45] Michael A. Harrison. On models of protection in operating systems. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1975, 4th Symposium, Mariánské Lázne, Czechoslovakia, September 1-5, 1975, Proceedings*, volume 32 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 1975.

[46] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*, pages 100–114. IEEE Computer Society, 2004.

[47] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005.

[48] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

[49] Ankush Das, Shuvendu K. Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 324–342. Springer, 2015.

[50] Ted Kremenek and Dawson R. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In Radhia Cousot, editor, *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings*, volume 2694 of *Lecture Notes in Computer Science*, pages 295–315. Springer, 2003.

[51] Saurabh Joshi, Shuvendu K. Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 19–30. ACM, 2012.

[52] Sam Blackshear and Shuvendu K. Lahiri. Almost-correct specifications: a modular semantic framework for assigning confidence to warnings. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 209–218. ACM, 2013.

[53] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 181–192. ACM, 2012.

[54] Rastislav Bodík, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 339–352. ACM, 2010.

[55] Matthias Heizmann. *Traces, interpolants, and automata: a new approach to automatic software verification*. PhD thesis, University of Freiburg, Freiburg im Breisgau, Germany, 2015.

[56] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Refinement of trace abstraction. In *SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2009.

[57] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. Ultimate automizer with smtinterpol - (competition

contribution). In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 641–643. Springer, 2013.

[58] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Nested interpolants. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 471–482. ACM, 2010.

# Appendix A

# Program Names

**loops1** svcomp/c/loop-acceleration/multivar_false-unreach-call1_true-termination.i.

**loops2** svcomp/c/loop-acceleration/phases_false-unreach-call2_false-termination.i.

**loops3** svcomp/c/loop-acceleration/underapprox_false-unreach-call1_true-termination.i.

**recursive1** svcomp/c/recursive-simple/fibo_2calls_5_false-unreach-call_true-termination.c.

**recursive2** svcomp/c/recursive-simple/fibo_7_false-unreach-call_true-termination.c.

**recursive3** svcomp/c/recursive-simple/id_i25_o25_false-unreach-call_true-termination.c.

**recursive4** svcomp/c/recursive/Addition02_false-unreach-call_true-no-overflow_true-termination.c.

**recursive5** svcomp/c/recursive/Fibonacci04_false-unreach-call_true-no-overflow_true-termination.c.

**recursive6** svcomp/c/recursive/Fibonacci05_false-unreach-call_true-no-overflow_true-termination.c.

**ssh1** svcomp/c/ssh-simplified/s3_clnt_1_false-unreach-call_true-termination.cil.c.

**ssh2** svcomp/c/ssh-simplified/s3_clnt_3_false-unreach-call_true-termination.cil.c.

**ssh3** svcomp/c/ssh-simplified/s3_srvr_10_false-unreach-call_false-termination.cil.c.

**ssh4** svcomp/c/ssh-simplified/s3_srvr_13_false-unreach-call_false-termination.cil.c.

**ssh5** svcomp/c/ssh-simplified/s3_srvr_1_false-unreach-call_false-termination.cil.c.

**systemc** svcomp/c/systemc/transmitter.15_false-unreach-call_false-termination.cil.c.

# Appendix B

# Unvalidated Input Detection

## B.1 Doublelock in Unix File System

### B.1.1 Patch that introduced the bug

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9ef7db7f38d0472dd9c444e42d5c5175ccbe5451

### B.1.2 Simplified program

https://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/programs/toy/errorLocalization/vadim/paths_32_7a_ufs.c

## B.2 Doublelock in Aeroflex Gaisler GRGPIO

### B.2.1 Patch that fixed the problem

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/drivers/gpio/gpio-grgpio.c?id=7fa25937542358bfa01ef5c5a1e9a00bd164c000

### B.2.2 Simplified code

https://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/programs/toy/errorLocalization/vadim/paths_39_7a_gpio_grgpio.c

### B.2.3 Original driver

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/gpio/gpio-grgpio.c

## B.3 imon_probe() exits with mutex acquired

### B.3.1 Buggy commit

Commit that introduced the bug: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/drivers/staging/media/lirc/lirc_imon.c?id=af8a819a2513df4be461c6a29e3bdee6e23cf3be

### B.3.2 Patch that fixed the problem

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/drivers/staging/media/lirc/lirc_imon.c?id=b833d0df943d70682e288c38c96b8e7bfff4023a

### B.3.3 Simplified code

https://github.com/ultimate-pa/ultimate/blob/dev/trunk/examples/programs/toy/errorLocalization/vadim/paths_32_7a_lirc_imon.c

### B.3.4 Driver

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/staging/media/lirc/lirc_imon.c?id=0d2b7ea9287d39e87531d233ba885263e6160127