Automatic Error Detection Techniques Based on Dynamic Invariants

Master’s Thesis

Alberto González Sánchez
Automatic Error Detection Techniques Based on Dynamic Invariants

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Alberto González Sánchez
born in Valladolid, Spain

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Universidad de Valladolid
ETSI de Ingeniería Informática
Valladolid, Spain
www.uva.es
Abstract

This document presents the methods used to test the suitability of some already existing automatic error detection techniques to embedded systems, and their performance when used as input for the program spectra fault detection algorithm. These error detection techniques are important for locating unstable states of the firmware of embedded systems, triggering recovery routines that will avoid a sure crash.
I came to the Delft University of Technology to make my Computer Engineering final project, but instead I found myself doing research in an interesting field of Embedded Systems, in that aspect I think I have been very lucky. I have had to learn the ways of research, enjoy the (few) moments of success, cope with the (lots of) moments of failure and fight against procrastination.

I want to thank my supervisors Arjan van Gemund, and Rui Abreu in the first place for giving me this opportunity, and for all the help during this months of work and all the advice they have given me.

There has to be also a bit of space for all the good friends that I’ve made here from all around the world, and for the friends I already had at home and that have missed me all this time. I don’t want to make a list, as I’m sure I would forget about someone. They know who they are.

I don’t want to forget about DUWO, the University’s housing company, for showing me that the luxuries of the modern world such as washing machine, microwave oven or freezer are superfluous and that life without them is still possible.

Last but not least, thank my parents, Alfonso and María, for all the dedication they have put on me all these years with the hope that I’ll earn more money than them to procure them a good and relaxed retirement. They are both university professors and researchers, and so with this work I feel as if I was continuing with the family business.

Alberto González Sánchez
Delft, the Netherlands
August 12, 2007
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

1.1 Motivation 1
1.2 The TRADER Project 2
1.3 Basic Concepts of Dependability 2
1.4 Spectrum-based Fault Localization 6
1.5 Research Question(s) 7

2 Automatic Error Detection 9

2.1 Program invariants 9
2.2 Instrumenting program points at C level 14
2.3 Instrumenting program points at binary level 16

3 The Delftgrind Error Detector 17

3.1 Overall architecture 17
3.2 Introduction to Valgrind 18
3.3 Implementation 20

4 The Ninjin Error Detector 27

4.1 Introduction to LLVM IR 28
4.2 Ninjin implementation 29

5 Experimental Setup 35

5.1 Benchmark Set 35
5.2 Data Acquisition 36
5.3 Evaluation Metrics 36
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Experiment Results</td>
<td>39</td>
</tr>
<tr>
<td>6.1 First experiments</td>
<td></td>
</tr>
<tr>
<td>6.2 Second experiments</td>
<td>43</td>
</tr>
<tr>
<td>6.3 Delftgrind’s Performance as Fault Localization Input</td>
<td>46</td>
</tr>
<tr>
<td>6.4 Third Experiments</td>
<td>48</td>
</tr>
<tr>
<td>6.5 Ninjin’s Performance as Fault Localization Input</td>
<td>52</td>
</tr>
<tr>
<td>7 Related Work</td>
<td>59</td>
</tr>
<tr>
<td>7.1 Invariant related work</td>
<td>59</td>
</tr>
<tr>
<td>7.2 Other dynamic error detectors</td>
<td>61</td>
</tr>
<tr>
<td>8 Conclusions and Future Work</td>
<td>63</td>
</tr>
<tr>
<td>8.1 Contributions</td>
<td></td>
</tr>
<tr>
<td>8.2 Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>8.3 Future work</td>
<td>66</td>
</tr>
<tr>
<td>Bibliography</td>
<td>67</td>
</tr>
<tr>
<td>A Result tables</td>
<td>71</td>
</tr>
</tbody>
</table>
List of Figures

1.1  Hierarchy of the basic concepts of dependability  . . . . . . . . . . . . . . . . . 3
1.2  Failure classification attributes  . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3  Example error propagation situation  . . . . . . . . . . . . . . . . . . . . . . 5
1.4  Main fault classes  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
1.5  False negatives and false positives  . . . . . . . . . . . . . . . . . . . . . . . 8
2.1  Instrumented program point and its entry in the invariant table  . . . . . . . . . 10
2.2  Block diagram of a bitmask invariant  . . . . . . . . . . . . . . . . . . . . . . 11
2.3  Range invariant block diagram  . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1  Overall architecture of Delftgrind  . . . . . . . . . . . . . . . . . . . . . . . 18
3.2  UML structure of Delftgrind version 1 backend  . . . . . . . . . . . . . . . . 21
3.3  Event dispatching sequence diagram  . . . . . . . . . . . . . . . . . . . . . . 22
3.4  Delftgrind 2 component diagram  . . . . . . . . . . . . . . . . . . . . . . . . 24
3.5  Delftgrind 2 program point table  . . . . . . . . . . . . . . . . . . . . . . . . 25
4.1  Ninjin component diagram  . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
6.1  print\_tokens  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.2  print\_tokens2  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.3  replace  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.4  schedule  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.5  scheduled  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.6  tcas  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.7  tot\_info  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
6.8  print\_tokens  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.9  print\_tokens2  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
6.10  replace  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.11  schedule  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.12  scheduled  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.13  tcas  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
List of Figures

6.14 tot.info .......................................................... 45
6.15 print.tokens ...................................................... 48
6.16 print.tokens2 ..................................................... 48
6.17 replace .............................................................. 48
6.18 schedule ............................................................ 48
6.19 schedule2 .......................................................... 49
6.20 tcas ................................................................. 49
6.21 tot.info .............................................................. 49
6.22 average .............................................................. 49
6.23 print.tokens ........................................................ 53
6.24 print.tokens2 ...................................................... 53
6.25 replace .............................................................. 54
6.26 schedule ............................................................ 54
6.27 schedule2 .......................................................... 54
6.28 tcas ................................................................. 54
6.29 tot.info .............................................................. 54
6.30 print.tokens ........................................................ 56
6.31 print.tokens2 ...................................................... 56
6.32 replace .............................................................. 56
6.33 schedule ............................................................ 56
6.34 schedule2 .......................................................... 56
6.35 tcas ................................................................. 56
6.36 tot.info .............................................................. 57
6.37 average .............................................................. 57
Chapter 1

Introduction

This chapter introduces all aspects that form the context and motivations of the graduate project. We will describe the main terms used throughout this thesis as well as the main research questions we will try to answer with our research.

1.1 Motivation

Modern technical systems such as embedded systems rely increasingly on software. Embedded computer programs monitor the whole system and ensure that the system accomplishes the tasks it was designed for. In such software-intensive systems reliability is of prime importance.

With the increasing complexity of today’s systems, and the pressure of the market, specially in the consumer electronics market, achieving an acceptable level of quality employing traditional techniques is unfeasible. The integration and construction of such complex systems in such a short period of time cannot be done without software accomplishing their integration.

A number of technical trends in embedded systems press a need for better development methods resulting in reliable products.

- **Complexity increase:** the extent and complexity of embedded systems and software has been exponentially increasing in recent years and there seems no end in sight for this trend.

- **Product life cycle decrease:** the innovation cycle for these products is decreasing continuously and has in many sectors now come down to a few months. Manufacturers want to have their products in the streets before the competence.

- **Open Systems:** embedded systems are opened to the outside world as a result of which security, reliability, and availability are emerging problems. They will not be solely developed by just one manufacturer. Increasingly, the provider of the basic functionality will be host to third parties who will add their own functionality. Furthermore, they will, during their lifetimes, become involved in networked environments that affect these systems in ways that are not foreseen during their creation.
No matter how thoroughly a system is tested, some bugs will remain hidden, and the final product will be shipped with them. These bugs will have to be addressed on future revisions. However, updating the controller software of an embedded system is more difficult (if possible at all) than updating traditional desktop applications. If the firmware of a digital television is faulty, replacing the version for a non-faulty one might not be easy. The firmware could be written in an ROM chip that has to be replaced, causing even more inconvenience for the user while the replacement part arrives.

It would be desirable that the controller component of the TV could detect the presence of an error, taking the appropriate actions in order to keep the quality of the service or at least a minimum level of it, instead of causing a complete failure.

1.2 The TRADER Project

It is the objective of the Trader product to develop methods and tools for ensuring reliability of consumer electronic products. This should result in minimizing product failures that are exposed to the user. We use cases from NXP's Innovation Center Eindhoven in the area of digital television. The Trader project has the following objectives in mind:

- Ensure reliability by studying and showing proof of concept of methods to be applied at design time, test time, and product run-time.
- Avoid user frustration by applying user-centric approaches. Ensure reliability not only in a single product, but also in a complete product line.

The Trader project is focused on the industrial sector of consumer electronic products, in particular on digital television. Consumer products present the most stringent conditions due to the high emphasis on both cost-effectiveness and ease of use.

Trader is a collaboration of the industrial and academic partners: NXP (previously Philips Semiconductors), Philips Consumer Electronics, NXP Research (previously Philips Research Laboratories), Philips TASS, Design Technology Institute, Delft University of Technology, University of Twente, University of Leiden, IMEC, and the Embedded Systems Institute.

1.3 Basic Concepts of Dependability

There are various concepts related to the security and dependability of computer systems that are going to be used throughout this Master’s Thesis which we need to clarify. There are plenty of previous works in classifying the main concepts of dependability. Most of the concepts explained in this section are taken from the work of Avizienis, Laprie and Randell [3][14].

The *dependability* of a computer system is the ability to deliver a service that can be justifiably be trusted. The service delivered by a system is its behaviour as it is perceived by its users; a *user* is another system or human that interacts with the former by means of its service interface. The function of the system is what the system is intended for as described in the system specification.
The concepts of dependability can be separated in three classes: means to achieve dependability, attributes of dependability, and threats to dependability. An outline of all the involved concepts can be seen on Figure 1.1.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Means</th>
<th>Threats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Fault prevention</td>
<td>Faults</td>
</tr>
<tr>
<td>Reliability</td>
<td>Fault tolerance</td>
<td>Errors</td>
</tr>
<tr>
<td>Safety</td>
<td>Fault removal</td>
<td>Failures</td>
</tr>
<tr>
<td>Confidentiality</td>
<td>Fault forecasting</td>
<td></td>
</tr>
<tr>
<td>Integrity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maintainability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.1: Hierarchy of the basic concepts of dependability

1.3.1 Threats to Dependability

We will focus on the threats that a dependable system has to overcome to provide a correct service. Correct service is delivered when the system implements the system function. If the service provided by the system deviates from the expected correct service, a failure happens. A failure is a transition from correct service to incorrect service, the opposite transition, i.e. from incorrect to correct service, is called service restoration.

Failures are caused by a part of the system state that is not consistent (an error) and eventually reaches the service interface, altering the service provided by the system. It is important to note that the failure will never happen unless the error reaches the service interface. An error is detected if the system can emit an error message or signal over it. Undetected errors are latent errors. A fault is the actual cause for the error. A fault that does not produce an error is a dormant fault, otherwise is an active fault.

Failures

Failures can be classified according to four main criteria as shown in Figure 1.2, the domain of the failure, the detectability of the failure, the consistency of the failure and the consequences of the failure.

Depending on the domain criterion, failures can be content related (the information delivered by the service interface deviates from the correct service), timing related (the time of arrival or the duration of the information delivered deviates from the correct service), or a mixture of both.

The detectability of a fault depends on the ability of the system to verify the correctness of the delivered service. When the system is able to detect and signal the failure, then a signaled failure occurs. Otherwise, they are unsignaled failures.
1.3 Basic Concepts of Dependability

Failures

<table>
<thead>
<tr>
<th>Domain</th>
<th>Consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>Minor</td>
</tr>
<tr>
<td>Timing</td>
<td>...</td>
</tr>
<tr>
<td>Mixed</td>
<td>Catastrophic</td>
</tr>
<tr>
<td>Detectability</td>
<td>Consequences</td>
</tr>
<tr>
<td>Signaled</td>
<td>Consistent</td>
</tr>
<tr>
<td>Unsigned</td>
<td>Inconsistent</td>
</tr>
</tbody>
</table>

Figure 1.2: Failure classification attributes

The consistency of failures is related to the way the service is provided to multiple clients in the service interface. If the incorrect service caused by the failure is perceived identically by all the users, then the failure is consistent. If the failure is perceived only by a part of the users while other part might perceive the service as correct, then the failure is inconsistent.

Finally, the severity of a failure can be rated, and several failure severities can be defined, ranging from minor failures, where the harmful consequences have a similar cost than providing a correct service, to catastrophic failures, where the cost of the harmful consequences is orders of magnitude higher than the cost of providing a correct service.

Errors

The total state of a complete system is the set of all its components’ states. An error is a part of the system’s state that may eventually lead to a failure. A single fault will at first cause an error in one or more components, but the system failure will not take place unless the error reaches the system interface, becoming a part of the external state of the system. Whether or not the error will actually lead to a failure depends on the structure of the system (existing redundancy), and its behaviour (the error may be corrected or overwritten before it reaches the system interface). Errors can be classified following the same criteria as the failures they cause, as explained in Section 1.3.1.

Errors propagate internally during the computation process, being transformed and causing more errors. Errors can propagate to other components through the service interface. An example situation of error propagation can be seen in Figure 1.3.

Faults

Faults and their sources are very diverse. Figure 1.4 shows the eight basic classification viewpoints. Faults can also be grouped in three major overlapping groups:

- **development faults** that occur during development,
- **physical faults** that affect hardware,
interaction faults that include all external faults.

<table>
<thead>
<tr>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Phase of creation/occurrence</strong></td>
</tr>
<tr>
<td>Development: occur during development/maintenance</td>
</tr>
<tr>
<td>Operational: occur during use phase</td>
</tr>
<tr>
<td><strong>Cause</strong></td>
</tr>
<tr>
<td>Natural: without human participation</td>
</tr>
<tr>
<td>Human-made: result from human actions</td>
</tr>
<tr>
<td><strong>System boundaries</strong></td>
</tr>
<tr>
<td>Internal: originate inside the system</td>
</tr>
<tr>
<td>External: originate outside and propagate inside</td>
</tr>
<tr>
<td><strong>Dimension</strong></td>
</tr>
<tr>
<td>Hardware: originated of affecting hardware</td>
</tr>
<tr>
<td>Software: affect programs or data</td>
</tr>
<tr>
<td><strong>Intention</strong></td>
</tr>
<tr>
<td>Deliberate: result of wrong decision</td>
</tr>
<tr>
<td>Non-Deliberate: introduced without awareness</td>
</tr>
<tr>
<td><strong>Objective</strong></td>
</tr>
<tr>
<td>Malicious: objective of causing harm</td>
</tr>
<tr>
<td>Non-malicious: without malicious intentions</td>
</tr>
<tr>
<td><strong>Capability</strong></td>
</tr>
<tr>
<td>Accidental: introduced inadvertently</td>
</tr>
<tr>
<td>Incompetence: from a lacking professional competence</td>
</tr>
<tr>
<td><strong>Persistence</strong></td>
</tr>
<tr>
<td>Permanent: presence continuous in time</td>
</tr>
<tr>
<td>Transient: presence bounded in time</td>
</tr>
</tbody>
</table>

Human-made faults include faults caused by absence of actions when actions should be performed (omission faults), and commission faults, i.e performing wrong actions. Depending on the objective of the faults, they can be divided in malicious and non-malicious faults. During this research we will only deal with non-malicious faults, specially non-deliberate non-malicious faults.
1.4 Spectrum-based Fault Localization

Fault localization through the analysis of program spectra is an automated debugging technique aimed at reducing the bottleneck of debugging in the testing phase of development. A program spectrum can be seen as a projection of the execution trace that shows which parts (blocks, statements) of the program were active during its execution. The diagnosis consists in analyzing which of the parts show the biggest correlation with the error pattern of different executions.

Program spectra are collected at run-time, and consist in a number of counters for each part of the program. If we were only interested in whether the parts have been executed or not, a boolean flag would be enough instead of a counter. Every run of a program produces a program spectrum, which is then placed as a row in a binary matrix of \( M \) rows (one for each run) and \( N \) columns (one for each block or statement). A binary vector of size \( M \) is constructed as well, using the error diagnosis information of each run. For each column of the matrix, its similarity to the error vector is evaluated. The block with the highest similarity is most likely to contain the fault.

\[
\begin{bmatrix}
    x_{11} & x_{12} & \cdots & x_{1N} \\
    x_{21} & x_{22} & \cdots & x_{2N} \\
    \vdots & \vdots & \ddots & \vdots \\
    x_{M1} & x_{M2} & \cdots & x_{MN} \\
\end{bmatrix}
\begin{bmatrix}
    e_1 \\
    e_2 \\
    \vdots \\
    e_M \\
\end{bmatrix}
\]

Table 1.1: The ingredients of fault diagnosis

The grade of similarity between block columns and the error vector is calculated using similarity coefficients taken from data clustering techniques and molecular biology.

The similarity coefficients are calculated using four counters of pairs of values that can occur on any position \( i \) in these two vectors:

<table>
<thead>
<tr>
<th>counter</th>
<th>( x_{ij} )</th>
<th>( e_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_{11} )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( a_{10} )</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( a_{01} )</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( a_{00} )</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In this table, \( a_{11}(i) \) is the number of positions in which the \( j \)-th column vector and the error vector share an entry 1, i.e., the number of spectra in which block \( j \) was recorded to be executed, and for which an error has been detected. \( a_{10}(i) \), on the other hand, is a run where the \( j \)-th block has been executed but no error has been reported. \( a_{01} \) represents the number of times the block wasn’t executed and there was an error, and \( a_{00} \) counts the number of runs where the block was not executed and no error was reported. These four counters sum up to the number of spectra \( M \).
In the experiments of this Master’s Thesis we will evaluate our error detection tech-
niques when used as inputs to four different similarity coefficients:

- **Jaccard:**
  \[
  s_j(i) = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i) + a_{10}(i)}
  \]  
  The Jaccard coefficient, which is well-known from the field of data clustering (see, e.g., [10]), is used in the Pinpoint framework [5].

- **Tarantula:**
  \[
  s_j(i) = \frac{a_{11}(i)}{a_{11}(i)+a_{01}(i) + a_{10}(i)}
  \]  
  This coefficient is used in the Tarantula system [12, 11].

- **Ochiai:**
  \[
  s_j(i) = \frac{a_{11}(i)}{(a_{11}(i) + a_{01}(i))(a_{11}(i) + a_{10}(i))}
  \]  
  This coefficient is used in [19] for computing genetic similarity in molecular biology.

- **Zoltar:**
  \[
  s_j(i) = \frac{a_{11}(i)}{a_{11}(i) + a_{01}(i) + a_{10}(i) + 10000 \cdot \frac{a_{01}(i) - a_{10}(i)}{a_{11}(i)}}
  \]  
  This coefficient has been developed in the Embedded Software Lab of the Delft University of Technology. It’s a modification of the Jaccard coefficient in where the denominator has a new term that helps to rule out non faulty blocks.

The blocks or statements will then be ranked according to the values of their similarity coefficients, with the most likely suspects on top of the list. The effectivity of the method will depend on the quality of the error detection as well as on the quality of the similarity coefficients.

### 1.5 Research Question(s)

There have been previous approaches to software error detection and fault diagnosis based on program invariants, such as DAIKON [6] that are very powerful (See Section 7.1.3). However, this approaches involve checking a great number of possible invariants, which involves costly operations. Moreover, the final model of the system is quite complex and not suitable for its use in embedded systems.

On the other hand, the DIDUCE tool [8] and Racunas’ fault screener [26] use lightweight unary invariants that are well suited for their use in an embedded environment (See Sections 7.1.1 and 7.1.2). We will evaluate the ability of such invariants to detect software errors of varying nature, and will try to find their strengths as well as their weaknesses and the conditions under which they can be most effective.
We will then evaluate the results of the error screening techniques researched when used as input of the fault localization techniques based on program spectra introduced in the previous section.

As with most approximate techniques, there has to be a point of somewhere in the middle of the training process where a good balance between the false alarms (false positives) and the error misses (false negatives) is achieved, and where the results for the fault localization are optimal.

Previous work on the accuracy of similarity coefficients has been carried out in a false positive free setup. Under real conditions there will be cases where the imperfections of the method will cause a perfectly correct case to be flagged as erroneous. Our study will show for the first time the effects of false positives in the quality of the fault diagnosis.
Chapter 2

Automatic Error Detection

In this chapter we present the basic methods and techniques of automated error detection based on program invariants. Most of the work of this chapter is based on the work of Hangal and Lam’s tool for Java, DIDUCE [8], Ernst’s similar tool for C, Daikon [6], and the recent work of Racunas that was presented at the HPCA this year [26].

2.1 Program invariants

A program invariant is an assertion about the conditions that have to be met by the state of the program for it to be correct. If there is an inconsistency in a running program, this means that some of the assertions will be violated and this will be reported as an error.

The normal procedure is to manually seed a program with assertions while programming, and letting the program check those assertions at runtime. However this is not always done or possible. There could be space or time constraints that disallow assertion checking, or the code may come from legacy and it might be not well understood, or we might not have even access to the source code. In addition, hand-specified assertions tend to capture only a few high-level and low-level invariants at the parts considered the most critical, but ignoring the rest of the program [8].

Instead, we can have a program infer the invariants for us [6] [8] during a training phase, creating a model of the program based on its dynamic behaviour without any human intervention. During the training phase new invariants are created at every program point, and are progressively adapted to fit as tightly as possible to the program’s behaviour. As with other dynamic methods, the model is not perfect and depends on the quality of the training set and the training itself and the optimality of the technique. Ideally speaking, after training with the whole training set, the invariants should only accept the correct (training) cases and reject the rest. However, the techniques we will use are not ideal, and if the training is too exhaustive the invariants might be very relaxed and the number of false negatives will be too high for the method to be useful. We have to find an intermediate point with low false positive and false negative indices. After the training has been completed, the finished model of the program can be used to check new runs of it, and whenever an invariant is not satisfied, report it as an error.
The invariants we want to test are from the two kinds most suited for embedded systems thanks to their small footprint: bitmask invariants \[8, 26\] and dynamic range invariants \[26\]. Other kinds of invariants have been proposed \[6, 7, 26\], but they require costly operations and much more storage space, and they are not so well suited for application on embedded systems. Figure 2.1 shows the instrumentation of a program point and the entry that is added to the invariant table with both the bitmask and the range invariant data.

2.1.1 Bitmask invariants

A bitmask invariant is composed of three fields: the first value that was observed, the number of times that it has been checked, and a bitmask that represents for what bits different values have been observed. The bitmask invariants used by Hangal \[8\] also include a second bitmask for the delta between two consecutive values. We decided not to include this second bitmask as it will duplicate the storage space needed by each invariant and the time required for checking it.

The bitmask is initially set to all 1’s. An example of the life of an 8 bit invariant can be seen in Table 2.1.

Every time the invariant is used, the counter is increased, and the bits are checked with the following formula, to check whether there are any new variant bits:

\[
\text{violation} = (\text{new} \oplus \text{first}) \land \text{mask}
\]

(2.1)

where \(\oplus\) and \(\land\) are the bitwise xor and and operators respectively. If the result of equation 2.1 is not zero, an invariant violation has to be reported, and the invariant has to be relaxed with the following formula:

\[
\text{mask} = \neg(\text{new} \oplus \text{first}) \land \text{mask}
\]

(2.2)
Table 2.1: The typical life cycle of an invariant since it is created in training until it is used in the checking phase. During the training only odd values are observed, and thus the first bit is a 1 (fixed) in the bitmask. During the checking we observe an even value, and it is reported as a violation.

<table>
<thead>
<tr>
<th>Times checked</th>
<th>First</th>
<th>Bitmask</th>
<th>New</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0001 0101</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0001 0101</td>
<td>1111 1111</td>
<td>0001 0111</td>
<td>0000 0010</td>
</tr>
<tr>
<td>2</td>
<td>0001 0101</td>
<td>1111 1101</td>
<td>0101 0001</td>
<td>0100 0100</td>
</tr>
<tr>
<td>3</td>
<td>0001 0101</td>
<td>1011 1001</td>
<td>0001 1111</td>
<td>0000 1000</td>
</tr>
<tr>
<td>Checking</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0001 0101</td>
<td>1011 0001</td>
<td>0000 0010</td>
<td>0000 0001</td>
</tr>
</tbody>
</table>

With just these three values we are able to keep a record of some interesting properties of the values observed in a program point:

- If only negative or only positive values have been observed, the first bit will be 1.
- If only odd or even values have been observed, the last bit will be 1.
- An approximate upper bound can be established.

Another interesting property is that if the variable is increasing monotonically, i.e., it is being used as a counter, the number of violations will decrease exponentially, as every violation will duplicate the size of the set of accepted values.

In addition, a confidence level for this invariant has been proposed [8] as a way of prioritizing invariant violations to select the most important among all the noise caused by minor violations:

\[
C = \frac{n_{chk}}{2^v} \tag{2.3}
\]
where \( n_{chk} \) is the number of times the invariant has been checked, and \( z \) the number of bits set to zero in the mask. When an invariant has been checked many times with only a few violations, its confidence level will be very high. The value of \( 2^z \) represents the number of accepted values by the invariant. Every time an invariant is violated, the drop in the confidence level can be used to decide whether the violation represents an important change in program behaviour or not.

**Limitations**

Bitmask invariants are not free of problems. First of all, they cannot be used for floating point values, although in embedded environments this kind of variables are not common (the Philips TV code does not use floating point at all). But the most important limitation is that their ability to represent ranges is quite limited.

For example, consider the variable `token_ind` in the following piece of code, based in one of the programs in the Siemens set:

```c
while(ch = get_char()) {
    token_str[token_ind++] = ch;
}
```

The array `token_str` acts as a statically allocated buffer of size 80. During the training phase all token lengths have been less than 80, therefore the value of the bitmask for `token_ind` is 0x1000000. However, this bitmask’s upper bound is 127, so when a token of length 81 comes the invariant won’t be violated even if it’s a clear buffer overflow error. This kind of error would not pose a big problem in other programming languages like Java, where array bounds are checked every time the array is accessed.

Another issue is the problem with functions that can return 0 or -1. If an invariant is created with an initial value of 0 (0x00000000) and then the value -1 (0xFFFFFFFF) is returned in a subsequent pass, the bitmask will accept any value, even if only 0 and -1 are correct. For example, consider the return value of `strcmp()` in the following code:

```c
if (strcmp(a, b) == 0) {
    printf("a = b");
} else {
    printf("a shorter than b");
}
```

During the training phase only values of `a` shorter or equal than `b` have been used, and `strcmp()` has always returned 0 or -1. In the checking phase a value of `a` longer than `b` is passed to the program, making `strcmp()` return 1. This should be reported as a violation, but this will not happen because the invariant is already at its most general form.

### 2.1.2 Range invariants

Range invariants are used by Racunas et Al in their hardware perturbation screener [26], and try to build multiple variable ranges for each program point. In our implementation...
instead of multiple ranges, each program point has only one range either for integers or for real numbers. Every time a new value is observed, it is checked against the current bounds of the range. If the value is outside those bounds, the range is extended to fit the new value. Table 2.2 shows the evolution of a range invariant.

<table>
<thead>
<tr>
<th>Times checked</th>
<th>Lower</th>
<th>Upper</th>
<th>New</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Training</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td><strong>Checking</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2.2: The typical life cycle of a range invariant since it is created in training until it is used in the checking phase. During the training values between 1 and 10 are observed. During the checking we observe a value greater than 10, and it is reported as a violation.

Range invariants have also their own disadvantages, as we will soon see. They are slower in the learning phase specially in the presence of counters, but that is the price that has to be paid to get tighter bounds. Furthermore, range invariants are unable to represent some conditions that the bitmasks can, such as even/odd properties.

Another point to note is that range invariants do not have a confidence level defined. Although in our experiments the confidence level was not used, it would be interesting to have an indicator of the confidence of this kind of invariant as well. Taking the bitmask confidence formula as a base, where the value of $2^z$ represents the total number of different values that are accepted by the invariant, we could redefine that value as the difference between the upper and the lower bound (the range of the range). However, we have not done any testing regarding its effectiveness or suitability.
Almost every statement in a C program can be of interest. There are many interesting aspects of a C program we would like to monitor: reads and writes from variables, function call parameters, function return values, array indexing bounds, pointer addresses, branches taken on if statements or number of iterations in loops.

Instrumenting C code proved to be a daunting task in all but the simples cases. Most of the C in the wild does not conform to the standard grammar, and uses many compiler specific extensions. Moreover, C’s grammar is ambiguous, so parsing it is not easy and requires a typechecking pass. Most of the previous experiments in dynamic invariant detection have been applied to Java bytecode, which is easier to deal with than C [7, 8].

We had, however, a C frontend written in C (cfront) that could parse all the source code of the test set, and we set our first goal: inspect writes to variables (globals, locals, arguments). It seemed quite straightforward in theory, like in the following code:

```c
int var;
var = 42;
```

Every time a variable is modified, a call to the logging system is made. The file and line number where the logging call is made could be used as an identifier for the program point, as well as a simpler numeric identifier.

```c
int var;
var = 42;
_log_write(var);
```

However, theory and practice are only equal in theory, and thus, for the following example code recording the memory writes in this expression requires more changes:

```c
int st, i;
char *p;
...
while (st = next_state(stream[i++])) {
  *p++ = get_char();
}
```

For this expression to be easily instrumented, it would have to be rewritten like this:

```c
int st, i;
char *p;
int temp0;
```
Automatic Error Detection

2.2 Instrumenting program points at C level

... temp0 = (st = next_state(stream[i]));
i = i + 1;
_log_write(i);
_log_write(st);

while(temp0) {
        *p = get_char();
    _log_write(*p);
    p = p + 1;
    _log_write(p);

    temp0 = (st = next_state(stream[i]));
i = i + 1;
    _log_write(i);
    _log_write(st);
}

If we wanted to instrument the reads as well, and then function calls and returns, the transformation would be more and more complicated. In other words, we needed to do the same transformation that GCC does from C to the GIMPLE intermediate representation [18]. For this task we had three possible alternatives: GCC, CIL and LLVM.

Using GCC as the instrumentation tool, creating our own tree-SSA pass, would give us almost unlimited parsing abilities, as we would be able to parse all the source languages supported by GCC with all the possible extensions. This ideal approach has the drawback of the complexity and not very good documentation of GCC’s source tree and APIs.

CIL [21] is a C simplifying tool written in Ocaml that takes a program written in ANSI C and transforms it to a clean subset of the C grammar. All loops are transformed to a single form, and all the syntactic sugar is removed. This process yields an easy to analyze and transform version of the original program that retains the typing information. We have been able to parse all the code from the Siemens set with CIL, but not the code from the Philips TV.

LLVM is a compiler and low level virtual machine for C and C++ written in C++ [15][16]. Instrumenting code with LLVM is in the middle between the binary and source code approaches. The code is compiled to a bytecode representation in a similar fashion to the bytecodes from the JVM or .NET, that keeps all the typing information, while eliminating all the burden of the C grammar.

LLVM’s output bytecode can be easily instrumented implementing the transformation as an optimization pass. The transformed code can be run inside the virtual machine or translated to the native instruction set of the underlying architecture, which is significantly faster.
2.3 Instrumenting program points at binary level

For the first stage of the research we discarded working directly with the source code, as then we didn’t yet need much high level information, in favour of a binary level instrumentation tool implemented as a Valgrind tool [24]. We thought that the higher level information we could possible need could be obtained from the debugging symbols and that keeping as low level as possible would be better if later on we were to take these methods to an embedded environment.

Binary instrumentation takes a program binary and disassembles it (or just works at bytecode level), allowing the insertion of function calls in interesting points of the program. Then the program is recompiled and the instrumented version executed. We have tested two binary instrumentation tools: PIN [17] and Valgrind.

PIN tools are easier to program than Valgrind tools. They are written in C++ using PIN’s callback system, which includes callbacks to work at instruction, basic block, routine or image level. The type of the instruction (integer, memory, etc) can be obtained and instrumentation function calls can be added before or after it is executed. Despite its ease of use, PIN is closed source, which is a setback. Moreover, PIN’s instrumentation only allows adding function calls, while Valgrind would allow us to do a full rewrite of the code on each basic block if it was necessary.

Valgrind had its own drawbacks, such as forbidding the use of standard or external libraries, or a poorer documentation. Programming a Valgrind tool requires a bit of automake and autoconf hacking as well. Even with all these limitations, the prospect of contributing code for a free tool made us choose Valgrind instead of PIN.
Chapter 3

The Delftgrind Error Detector

The Delftgrind error detector is implemented in C as a Valgrind [24] tool. Valgrind provides a series of hooks where developers can implement most of their instrumenting facilities. The most important of them is the tool.instrument hook, that provides the means of rewriting the guest program’s code.

Valgrind first translates the guest program’s instructions into its own internal RISC-like instruction set: Valgrind’s execution engine intermediate representation (VEX IR). Valgrind currently provides translation engines for x86, x86_64 and PowerPC architectures, that allow the construction of cross-platform instrumentation tools. The code is split into basic blocks of instructions (blocks with a single entry point and multiple exit points) that are handed one at a time to the tool’s instrument callback function.

3.1 Overall architecture

Delftgrind is best understood as a three stage pipeline architecture, although the actual implementation is slightly different. The first stage of the pipeline is the Negi frontend\(^1\), that parses the test case info and passes each one to Valgrind. Then comes Valgrind itself, which does all the needed transformations and where the instrumented guest program is run. Every interesting event (currently memory loads and stores) is then reported to the Negi backend that contains an event dispatcher, which post-processes the information and then hands it down to the invariant checkers. The invariant checkers are in charge of selecting what events they want to monitor, as well as constructing, training and enforcing invariants. Currently only two invariant checkers have been developed, bitmask invariants and range invariants. The violation reports from each run are stored in text files and then processed by a collection of AWK scripts that produce some graphs and statistics for performance evaluation purposes.

\(^1\)The name *Negi* comes from the Japanese word for a kind of spring onion.
3.2 Introduction to Valgrind

3.2.1 Valgrind’s IR

Valgrind uses an architecture neutral, single static assignment (SSA) IR similar to the ones used in compilers. A program is composed of basic blocks that are actually superblocks: single-entry, multiple-exit pieces of code.

IR blocks are formed by a list of statements, operations with side effects such as register writes, memory stores and assignments to temporary registers. Statements are formed by expressions that represent operations without side effects such as constant values, memory loads and arithmetic operations. Expressions can be nested up to arbitrarily complicated levels. Thus, before the instrumentation pass they have to be flattened introducing temporary registers.

Valgrind’s approach to instrumentation is called disassemble and re-synthesise (D&R). The binary code of the executable is disassembled and translated to IR. The obtained IR is instrumented by a tool (Delftgrind in our case), and then translated back to machine code. D&R has the disadvantage of losing some low level information during the translation, such as access to the original instruction the translated block comes from. Valgrind tries to overcome this limitation providing a special marker statement (IMark) indicating the boundaries, addresses and lengths of the original instruction.

Code blocks go through eight phases:

1. **Disassembly**: the machine code is converted into unoptimised tree (non-flat) IR.

2. **Optimisation**: the IR is flattened and several optimisations are done, such as constant propagation, constant folding or dead code removal.

3. **Instrumentation**: each code block is passed to the tool instrumentation callback, which has to construct a new code block with the added instrumentation.

4. **Second optimisation**: second simpler optimisation pass.
5. **Tree building**: the flat IR is transformed back to a tree IR.

6. **Instruction selection**: the tree IR is converted to a list of instructions on the target architecture which uses virtual registers.

7. **Register allocation**: the virtual registers are replaced with real host registers.

8. **Assembly**: the instruction list with host registers is encoded and written to a block of memory.

The following sample code shows the state of a code block after phase 1:

```
0x24F275: movl -16180(%ebx,%eax,4),%eax
1: ------ IMark(0x24F275, 7) ------
2: t0 = Add32(Add32(GET:I32(12),# get %ebx and 
   Shl32(GET:I32(0),0x2:I8)), # %eax, and
   0xFFFFC0CC:I32) # compute addr
3: PUT(0) = LDle:I32(t0) # put %eax

0x24F27C: addl %ebx,%eax
4: ------ IMark(0x24F27C, 2) ------
5: PUT(60) = 0x24F27C:I32 # put %eip
6: t3 = GET:I32(0) # get %eax
7: t2 = GET:I32(12) # get %ebx
8: t1 = Add32(t3,t2) # addl
9: PUT(32) = 0x3:I32 # put eflags val1
10: PUT(36) = t3 # put eflags val2
11: PUT(40) = t2 # put eflags val3
12: PUT(44) = 0x0:I32 # put eflags val4
13: PUT(0) = t1 # put %eax

0x24F27E: jmp*l %eax
14: ------ IMark(0x24F27E, 2) ------
15: PUT(60) = 0x24F27E:I32 # put %eip
16: t4 = GET:I32(0) # get %eax
17: goto {Boring} t4
```

### 3.2.2 Instrumentation pass

Every basic block is instrumented separately in Valgrind. Tools must implement a callback that gets a pointer to a basic block as a parameter, and returns a pointer to an instrumented copy of that basic block.

The function can loop through each statement in the input basic block, normally copying it to the output basic block. When an interesting statement is found, the instrumenter can add its own statements. Adding statements can be cumbersome, so the easiest way to perform
3.3 Implementation

The instrumentation is adding custom function calls that can be implemented in C inside the tool’s own source code. In the following code an example block instrumenting loop is presented. Every statement carries a tag that identifies its kind. If we were interested in catching memory writes, we would only have to provide code for the Ist_Store case. In the example code, an external function is called with two parameters: the instruction pointer of the operation and the address that is being written.

```c
for (i = 0 ; i < bbIn->stmts_used; i++) {
    IRStmt* st = bbIn->stmts[i];
    /* Select type of IR instruction */
    switch (st->tag) {
        case Ist_Store:
            /* First do the real store */
            addStmtToIRBB( bbOut, st );
            /* Construct expression for the parameters */
            IRExpr* addr = st->Ist.Store.addr;
            IRExpr* ipad = IRExpr_Const(IRConst_U32(ip));
            IRExpr** argv = mkIRExprVec_2( ipad, addr );
            /* Construct dirty statement */
            IRDirty* di = unsafeIRDirty_0_N( 2, "trace_store", &trace_store, argv );
            addStmtToIRBB( bbOut, IRStmt_Dirty(di) );
            break;
        default:
            addStmtToIRBB( bbOut, st );
            break;
    }
}
```

```c
static VG_REGPARM(2) void trace_store(Addr ip, Addr addr) {
    VG_(printf)( "sto %08p %08p\n", ip, addr );
}
```

3.3 Implementation

Delftgrind has had two major revisions. In the first revision both the frontend and the backend were implemented in Ruby, both running in the same script for performance reasons. The Valgrind tool only generated log messages that the backend then parsed and processed. In our second revision the backend was moved inside Valgrind, alongside the instrumenting code. This provided a performance boost at the price of less extensibility (adding new invariant types was very easy in the first version thanks to Ruby’s dynamic nature.)
3.3.1 Delftgrind revision 1

In Delftgrind version 1, the Negi frontend is in charge of reading the test case file and result file, separating test case in ‘failed’ (cases that fail) and ‘passed’ (cases that don’t fail) runs. Then for each case a new instance of Valgrind is created with the appropriate command line parameters and redirected input. Its implementation was quite straightforward in Ruby. The Negi backend on the other hand uses an event listener pattern to allow adding new Delftgrind event messages and invariants very easily.

Both frontend and backend run in the same process for performance’s sake. The frontend attaches the input stream of Valgrind to the test case input file, and the output stream to the backend trace event listener. For each event that Delftgrind produces, the event listener parses it into a new event object and then dispatches it to each invariant screener, that can then do whatever they please with it. Each screener keeps its own separate table of invariants, that need to be loaded only once, as the Negi process is never terminated during the experiment. As the Negi backend does not have direct access to the debugging information of the target program, an extra screener has been added that listens to debug events. The rest of the screeners can access this information if they want to provide more verbose output messages. The basic structure of the backend can be seen in Figure 3.2 and its operating procedure sequence diagram in Figure 3.3.

![Figure 3.2: UML structure of Delftgrind version 1 backend](image-url)
3.3 Implementation

The Delftgrind Error Detector

The first version of Delftgrind supports three types of events:

- **store events** that are produced every time a value is written in memory, with the following format: \texttt{sto <EIP> <ADDRESS> <DATASIZE> <VALUE>}

- **load events** that are produced every time a value is loaded from memory, with the following format: \texttt{loa <EIP> <ADDRESS> <DATASIZE> <VALUE>}

- **debug events** that are sent each time a new program point is instrumented, with the following format: \texttt{dbg <TYPE> <EIP> <FILE> <LINE> where <TYPE> can be either sto or loa}

Invariants print their output messages using a common format followed by their specific information with the following structure:

\texttt{+++<INVARIANT> <EVENT> <EIP> <INVARIANT DATA>}

The invariant data section is different for each type of invariant. Bitmask invariants include the first value seen and the bitmask, as well as the confidence drop that followed. Range invariants include the value of the current and former lower and upper limits.

Two command line options were added to the Delftgrind component: one to instrument only static and global variables and other to restrict the instrumentation to only the code of the main program. The first option proved to be mostly useless in the experiments as the programs in the test suite use very few global and static variables. On the other hand, the second option flag turned out very useful, as Valgrind by default would instrument all the
code that is run, which includes the standard library as well as the interesting parts of the
code, causing a flood of uninteresting events that delayed the experiments a good deal.

### 3.3.2 Delftgrind revision 2

In order to boost the speed of the experiments and reduce the overhead of the message
passing between Delftgrind and the Negi backend, the backend was rewritten in C as a
component of Delftgrind. The invariant tables were unified to a single table with one entry
per program point that contains a set of supported invariants, depending on the program
point nature and data type.

The instrumented program points remain the same (memory loads and stores), but now
with one extra supported type: floating point numbers. As the program points now are
stored inside Delftgrind, the program point table has to be dumped to a file and loaded
again for each of the test cases.

The format of the violation log messages was refined, adding information of the num-
ber of times an invariant has been checked. The fields included in a log message are the
following:

- **EIP**: instruction pointer of the program point
- **Data type**: information about the size of the data as well as whether it is an integer
  or a floating point number.
- **Invariant type**: a four character code for the invariant type, currently bitmask, integer
  and floating point ranges.
- **Times checked**: number of times the invariant has been checked across all test cases.
- **Invariant information**: a variable number of fields with information about the vio-
  lation. For bitmasks, the first and the bitmask value. For ranges, the lower and upper
  bound.
- **Location**: source file and line number of the program point.

The following is an excerpt of the output produced by the second version of Delftgrind.

```plaintext
+++ Test case 271
0x08048A29 I32 busk 00118082 0x00000011 0xFFFFFFC0 print_tokens.c:628
0x08048A29 I32 irng 00118082 0x00000000 0x00000020 print_tokens.c:628
0x08048A2C I32 bmsk 00118082 0x00000011 0xFFFFFFC0 print_tokens.c:628
0x08048A2C I32 irng 00118082 0x00000000 0x00000020 print_tokens.c:628
```

The command line option for instrumenting only static and global variables was kept
although never used. On the other hand, the command line option to instrument only the
main program image was replaced by a command line option to select what source file
to instrument, which allowed us to have more accuracy than before. In version 1, some
code related to the initialization of the program was instrumented (for example, the `_start`
function) and was causing some unwanted violations that this new command line options allowed us to remove.

The program code is split into seven components that can be seen in Figure 3.4:

- **main**: initializes the Valgrind hooks and performs the instrumentation callback.
- **instrument**: is called by the instrumentation loop to add the instrumentation code.
- **invariant**: contains the invariant creation and checking functions.
- **statement**: contains the statement counter for producing the program spectra.
- **output**: dumps and reads the program point table.
- **clo**: processes command line options.
- **decoder**: converts an IMark instructions to the original IA32 instruction. Used to filter instructions that we don’t want to interfere with the results.

The instrumenter builds an IMark hash table indexed by instruction pointer address. Each position contains an array of instrumented instructions: the program points. The combination of instruction pointer and micro operation number is unique and works as program point identifier. The program point contains the list of invariants that are kept for it. Figure 3.5 shows the translation from the identifier to the invariants.

The identifier is actually never used during the execution of the program, only during the instrumentation to load and unload program points from external storage. The lookup process adds unnecessary overhead and once the program point is built, the instrumenter uses the pointer to the program point as the parameter for the screening function that is added after the load or store as part of the instrumentation.
Figure 3.5: Delftgrind 2 program point table
Chapter 4

The Ninjin Error Detector

Delftgrind is a binary instrumenter, and as such, it works at a very low level of abstraction, where almost all the relevant information from the source code has been lost. This poses a number of limitations, specially in terms of typing information.

To overcome the limitations of binary level instrumentation it is necessary to introduce some degree of source level information. However, instrumenting C code is cumbersome, as it has to be transformed into a simplified version of C with simple expressions and statements. Thankfully, there are two tools already available for that task: CIL and LLVM.

CIL \[21\] stands for *C Intermediate Language*. It is a source to source compiler that takes a normal C program as input, and outputs an equivalent version in a simplified version of C. The CIL compiler is compliant enough to compile the full Siemens set, so it was an interesting option, despite the fact that plugins for it had to be written in Ocaml, a programming language we had a lack of experience with.

Fortunately, we found another suitable tool: LLVM \[15\], which stands for *Low Level Virtual Machine*. The LLVM defines a bytecode similar to the one of the Java VM, but with a lower level of abstraction. Despite being low level, it retains the typing information of the original source code (except the signed or unsigned condition of integer variables), which makes it suitable for our purpose. Furthermore, the produced bytecode is Statically Single Assigned (SSA) code.

LLVM comes with a GCC backend designed to output LLVM bytecode files, which can then be instrumented using the libraries that are provided for modifying the bytecode. Having a GCC backend is a great advantage, because it means that almost every language and language extension supported by the original GCC will be supported by LLVM-GCC (currently only C, C++ and ObjectiveC). LLVM bytecodes can be executed in the virtual machine (interpreter or JIT, depending on the host architecture), or translated to a range of target architectures, such as IA32, IA64, ARM, Alpha, Sparc, and even again to C source code.
4.1 Introduction to LLVM IR

LLVM programs are composed of Modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker, which merges function (and global variable) definitions, resolves forward declarations, and merges symbol table entries.

4.1.1 Global variables

Global variables define regions of memory allocated at compilation time instead of run-time that may optionally be initialized.

Global variables are SSA-values that define pointer values that are in scope for all basic blocks in the program. Global variables describe a region of memory and thus, always define a pointer to the type of their contents, because all memory objects in LLVM are accessed through pointers.

4.1.2 Functions

A function definition contains a list of basic blocks. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and ends with a terminator instruction (such as a branch or function return).

The first basic block in a function is special in two ways: it is the first one executed on entrance to the function, and it is not allowed to have predecessor basic blocks (i.e. there can not be any branches to the entry block of a function). The purpose of this first block is to reserve the memory needed for the local variables of the function. LLVM functions are identified by their name and type signature. Hence, two functions with the same name but different parameter lists or return values are considered different functions, and LLVM will resolve references to each appropriately.

4.1.3 Type system

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed on the IR directly, without having to do extra analyses before the transformation. A strong type system makes it easier to read the generated code and allows easy implementation of code analyses and transformations. These are the most interesting data types:

**void** primitive type without a value.

**integer** a very simple derived type that simply specifies an arbitrary bit width for the integer type desired. Any bit width from 1 bit to $2^{23} - 1$ (about 8 million) can be specified. The most common integers are i8, i16 and i32.

**float** a 32-bit floating point value.

**double** a 64-bit floating point value.
The Ninjin Error Detector

4.2 Ninjin implementation

pointer represents a pointer or reference to another object, which must live in memory.

array a very simple derived type that arranges elements sequentially in memory. The array
type requires a size (number of elements) and an underlying data type. Variable sized
arrays can be implemented in LLVM with a zero length array.

struct used to represent a collection of data members together in memory. The packing of
the field types is defined to match the ABI of the underlying processor. The elements
of a structure may be any type that has a size.

Unfortunately, starting from version 2.0, LLVM does not differentiate between signed
and unsigned integers, but between signed and unsigned operations and casts. Although
during this stage of the research it is not a big drawback as all the code in the Siemens set
uses signed integers, a solution must be found before the program can be used with real
world code.

4.2 Ninjin implementation

Ninjin’s architecture is very similar to the architecture used in the first version of Delft-
grind. An experiment launcher was created reusing most of the code from the second ver-
sion of Delftgrind, and the invariant screener code was extracted to a standalone program
that parsed the logging output of the instrumented program. While Delftgrind provided
protection between the instrumented program and the instrumentation code, our current in-
strumentation would have been completely unprotected and a wrong pointer in the code
could modify the information collected by the instrumentation. Ninjin, unlike Delftgrind,
performs the instrumentation of the code at compile time instead of at runtime.
The target program is instrumented and linked with the libninjin runtime library,
which contains all the logging calls that produce the output that ninjin will parse. They
take an integer parameter that serves as program point identifier, and a second parameter
with the value written or read in such program point.

4.2.1 Instrumentation process

The instrumentation process is implemented as an optimization pass for the LLVM bytecode
optimizer in C++. As now we are instrumenting bytecode, there is no instruction pointer
that can be used as identifier for the program point. Instead, a sequential integer identifier
that corresponds to a position in the program point table is used.

The first step in the instrumentation is declaring all the instrumentation function proto-
types. Adding a function prototype in LLVM is very easy with the getOrInsertFunction() method of the Module class.

For example, to declare the function void __ninjin_ld_int(int, int) we just have
to write the following:

```c++
M.getOrInsertFunction("__ninjin_ld_int", Type::VoidTy,
                      Type::Int32Ty, Type::Int32Ty, NULL);
```
4.2 Ninjin implementation

After declaring all the functions that will be used for the instrumentation, the next step is instrumenting function call sites. This has to be done in the first place as the next steps will insert new function calls that we don’t want to instrument. For every parameter in the function, an instrumentation log call is added, as well as for the return value if there’s one.

The next two steps can be performed in any order, and involve the instrumentation of loads and stores. The logging calls for stores are inserted before the actual operations, whereas the logging call for the load operations have to be inserted after the load has taken place. Otherwise, the value would not be available.

The last step before the finishing touch is adding instrumentation for the spectrum generation. LLVM inserts line number information in the form of dummy function calls that are translated to debugging symbols when the executable is generated. Line number information is contained in \texttt{llvm.dbg.stoppoint()} function calls. This function takes three parameters: an integer representing the line number, another integer for the column number, and a pointer to the compilation unit descriptor that can be used to extract the source file name. For now we assume that the programs come from a single source file, so the only interesting parameter is the first one.

The finishing step is adding a call at the beginning of the \texttt{main()} function to initialize \texttt{libninjin}'s internal structures. The initialization adds a signal handler to catch segmentation faults and log them. The init function takes three parameters: the number of instrumented points, the number of source code lines (for the spectrum generator) and the version of the instrumentation so we can tell if a program point id set will find the correct program points during runtime.

The following example code shows a piece of LLVM assembly code before the instrumentation has taken place.

```assembly
call void @llvm.dbg.stoppoint( i32 398, i32 0, ... )
%tmp8 = load %struct.FILE** @stdout
%tmp9 = getelementptr [47 x i8]* @.str, i32 0, i32 0
%tmp810 = bitcast %struct.FILE* %tmp8 to i8*
%tmp11 = call i32 @fwrite( i8* %tmp9, i32 1, i32 46, i8* %tmp810 )

call void @llvm.dbg.stoppoint( i32 398, i32 0, ... )
call void @exit( i32 1 )
```

After the instrumentation is done, a great number of function calls are added. The instrumentation is a bit coarse with regard to types, and creates are a great number of useless casts that could be easily removed just adding a bit more intelligence to the instrumenter. As can be seen, the actual arguments to the function call don’t change from the uninstrumented version to the instrumented one, guaranteeing the same result.

```assembly
call void @llvm.dbg.stoppoint( i32 398, i32 0, ... )
call void @__ninjin_spectrum_count( i32 398 )
%tmp8 = load %struct.FILE** @stdout
```
The Ninjin Error Detector

4.2 Ninjin implementation

Stores:
when a value is written into memory, the program emits a logging message with the value and the program point. Memory writes include global and local variables and possibly function arguments. Every time a store is found, a function of the form __ninjin_st_* is called.

Loads:
each time a value is loaded from memory it is logged. Memory loads include global variables, local variables and function parameters as well. For loads, the analog function __ninjin_ld_* is inserted.

Arguments:
in addition to variable loads and stores, function arguments are also inspected. The instrumentation is done adding calls to the function __ninjin_arg_*.
4.2 Ninjin implementation

The Ninjin Error Detector

Return values: the bytecode instruction for function calls yields the return value of the called function. We also log this value inserting calls to __ninjin_ret_*.

There is still room for new program points. For example arguments used as function outputs (passed by reference) or array indexing. Because we would have to dereference pointers we don’t know if are valid, return arguments are difficult to instrument. There’s no easy and portable way to know that. In Linux it’s possible to read the /proc/self/maps file, but it only works in Linux and it’s slow as the map file has to be read every time. There is previously work on precise runtime bounds checking [28], but the method is too complex to implement it now when compared with the small improvement we would get.

Array indexing operations can be instrumented by looking at getElementPtr instructions in the bytecode. This is not as easy as it seems at first sight, because this instruction can have a variable number of arguments and can be used for accessing struct elements.

4.2.3 Instrumentation runtime library (libninjin)

The instrumentation runtime library contains the code for the instrumentation log functions as well as a signal handler for segmentation faults. In order to avoid parsing text logs with C, which can be cumbersome when not using an external library, binary logging is used.

All logging functions build a log packet that contains:

- **program point id**: the position of the program point in the table.
- **program point kind**: either load, store, return value, argument or index. There are two additional special program point types: spectrum count points and segmentation faults. The id field is set to zero for these special cases.
- **program point type**: the data type of the value being logged (int, double, pointer). Spectrum and segmentation fault program points have type integer.
- **value**: the value being logged (32 bits for integers, 64 bits for doubles, system dependent for pointers). For spectrum program points, the value contains the line number. For segmentation faults, the number of the signal (11 for SIGSEGV).

The initialization function will first send a packet with the version number, the total number of program points instrumented and the source code line count, so the screener can initialize its internal structures, allocating all the needed memory for the tables.

4.2.4 Invariant screener

The invariant screener (the ninjin executable file) is implemented in C reusing most of the invariant screening code from the second version of Delftgrind. However, we used this opportunity to rearrange it internally, adding a new invariant type and redefining the output format to be more explicit about data types and program point kinds.

Figure 4.1 depicts the components of Ninjin after the restructuring of Delftgrind’s code. The instrumentation components (instrument and decoder) have disappeared as that step
is done in the instrumentation step. Despite the fact that most of the dependencies have been greatly simplified, there is still one circular dependency that will have to be addressed in the following version. The dependency of `output.c` over `propoint.c` is data related (`output.c` reads and writes program points to file) and the most likely solution is to split the behaviour of `output.c` across the client components.

![Figure 4.1: Ninjin component diagram](image)

The new invariant type come as a result of the increased source level information, that allows us to know the type of every variable that is being read or written. The new invariant type is the **null pointer invariant**. It monitors if the value of a pointer has been null or non-null across different executions. If a pointer has always been null previously, and in the current execution a non-null value is observed, then the invariant will be violated. The reverse situation is very similar: if a pointer has always been non-null previously and we observe a null value, the invariant is violated.
Chapter 5

Experimental Setup

5.1 Benchmark Set

In order to test the performance of both the bitmask invariants and the range invariants error detectors a series of experiments were conducted, testing our tools against a set of test programs known as the Siemens set [9]. The Siemens set is formed by 7 different C programs. Each of them comes with a correct version as well as a set of faulty versions. Each of the faulty versions has only one fault, although the fault may be spread in different lines of the source code. In addition, the set provides a vast amount of test cases for each program, designed to provide full coverage of the program. Most of them produce correct results, but some of them cause failures such as errors in the output or memory protection faults. The Siemens set was not designed with the purpose of testing error detection techniques, however, it is commonly used by the research community for testing purposes. Table 5.1 lists the general characteristics of the Siemens set.

Not all the faulty versions could be used. Versions 4 and 6 of print_tokens were discarded because the fault is data dependent [2], version 10 of print_tokens2 was discarded because the fault causes an infinite loop that will generate several gigabytes of log files, slowing down the experiments by several days. Version 9 from schedule2 and version 32 from replace were discarded as well, as their set of failed runs is empty.

<table>
<thead>
<tr>
<th>Program</th>
<th>Faulty Versions</th>
<th>SLOC</th>
<th>Test Cases</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>7</td>
<td>342</td>
<td>4130</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>10</td>
<td>355</td>
<td>4115</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>schedule2</td>
<td>9</td>
<td>292</td>
<td>2650</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>schedule2</td>
<td>10</td>
<td>262</td>
<td>2710</td>
<td>priority scheduler</td>
</tr>
<tr>
<td>replace</td>
<td>32</td>
<td>512</td>
<td>5542</td>
<td>pattern recognition</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>135</td>
<td>1608</td>
<td>altitude separation</td>
</tr>
<tr>
<td>tot_info</td>
<td>23</td>
<td>273</td>
<td>1052</td>
<td>information measure</td>
</tr>
</tbody>
</table>

Table 5.1: Description of the Siemens Suite. SLOC value generated using David A. Wheeler’s ‘SLOCCount’.
5.2 Data Acquisition

A series of makefiles were created to carry out the experiments and process the output logs. The processing was done using AWK and Ruby scripts that processed the output files and produced, for each version: a false positive file in 5% steps of training, a false negative file in 5% steps of training, a diagnostic file for use by the fault locator at every 5% point, a spectrum file with all the spectra of every test case combined, and several gnuplot files for generating graphs.

In order to evaluate the results of our error detectors, we need to have a reference error detector to compare with. Every program in the Siemens set comes with a correct version, so we will use the output of these correct versions, comparing them with the output from the faulty ones. If the output of a given test case and faulty version differs from the output from the control version, the run will be marked as ‘failed’. Otherwise it will be marked as ‘passed’. This approach is flawed, as we will show in the experiment results, but for now it is the best we have.

5.3 Evaluation Metrics

5.3.1 Error detection metrics

**True positives** Our goal is to produce a warning when an error is detected by the error detection algorithms. Thus, a true positive is a test case that exhibits the error and consequently produces a warning caused by a violation of one of the program invariants.

**True negatives** Test cases that do not exercise the fault and therefore do not cause an error are true negatives. There is no error so the alarm should not go off. In theory, any test case in the set should not produce any violations when run for a second time, i.e. after all the invariants have been relaxed to fit the case behaviour.

**False positives** If we stopped the training after a percent of the training set test cases have been used, there will still be some of them that will need some invariant relaxations. This means that they will be reported by the checking phase as positive even when they are not. Therefore, we call them false positives.

We will define \( f_p(n) \) as the number of false positives after the \( n\% \) of the training set has been used \( (F_p(n)) \), divided by the size of the training set \( (N_t) \). It can be seen as just how many of the training cases would cause violations if the training set was used as checking set after \( n\% \) of the training. This should be a decreasing function.

\[
f_p(n) = \frac{F_p(n)}{N_t}
\]  

(5.1)

**False negatives** On the other hand, there might be some test cases that should be reported as erroneous but they are not. Those cases are the false negatives.
5.3 Evaluation Metrics

In this case we will define $f_n(n)$ as the number of false negatives present after the $n\%$ of the training set has been used ($F_n(n)$), divided by the size of the checking set ($N_c$). As the training progresses, the invariants are more and more relaxed, therefore allowing some errors to be considered as normal behaviour. This means that the number of false negatives will increase with the training.

$$f_n(n) = \frac{F_n(n)}{N_c} \tag{5.2}$$

<table>
<thead>
<tr>
<th>Error</th>
<th>No error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warning</td>
<td>True positive</td>
</tr>
<tr>
<td>No warning</td>
<td>False negative</td>
</tr>
<tr>
<td></td>
<td>True negative</td>
</tr>
</tbody>
</table>

Table 5.2: Relationship table between the presence or absence of an error and the presence and absence of a warning for that error.

5.3.2 Fault localization metrics

Using spectrum-based fault localization we will create a ranking of suspicious blocks. As in our experiments we will be using the Siemens set, we will be able to know which one is the faulty block. Therefore, we can calculate how many blocks we would need to inspect before finding the fault. In case two or more blocks have the same $s_j$, the average ranking position will be used.

Let $d \in \{1, \ldots, N\}$ be the index of the faulty block. For all $j \in \{1, \ldots, N\}$, let $s_j$ denote the similarity coefficient for block $j$. The ranking position of the block is given by:

$$\tau = \frac{|\{j | s_j < s_d\}| + |\{j | s_j \geq s_d\}| - 1}{2} \tag{5.3}$$

The quality of the diagnosis, i.e. the percentage of blocks that don't need to be considered when searching for the fault by traversing the ranking is defined as:

$$q_d = (1 - \frac{\tau}{N - 1}) \cdot 100\% \tag{5.4}$$

On the quality of the error detector: $q_e$

It is reasonable to think that the quality of the fault localization $q_d$ will be somehow dependent on the quality of the error detection, $q_e$. On one hand, the set of runs in which an error is detected will be a subset of the set of runs where the error is present (false negatives). On the other hand, the quality of the error detector is also affected by the number of times there is no error and the error detector signals one (false positives).

In previous work, the following formula was used [21]:

$$q_e = \frac{a_{11}(d)}{a_{11}(d) + a_{10}(d)} \cdot 100\% \tag{5.5}$$
in which $a_{11}(d)$ counts the number of times the fault has been executed, it has caused an error and it has been consequently detected. $a_{10}(d)$ on the other hand counts the number of times the fault has been executed and no error has been detected (no matter whether there was a real error or not). However, those experiments where performed in total absence of false positives. In our current experiments there is a possibility that an error is reported when, in fact, no error exists, no matter whether the faulty line has been executed or not. This situation definitely affects the quality of the error detection and should be taken into account. For this reason we will not be able to use the previous formula. Even though we have some possible candidate formulas, we will leave the selection and evaluation for future work.
Chapter 6

Experiment Results

In this section we will present and comment the results of all the experiments we have performed. Our initial methodology will involve training both program invariants with the training set test cases (those that do not cause a failure), and then see how the trained invariants behave in the cases in which the fault is exercised. Further on we will examine the effect of different degrees of training in the amount of false positives and false negatives. Our last experiments will study the way in which the false positive and false negative rates affect the quality of the fault diagnosis.

6.1 First experiments

In this section we will present the results of the experiments conducted with the first version of Delftgrind. A total of four experiments were conducted where just the final false negative rate was evaluated.

Our first goal was to monitor just memory writes and reads. Being Delftgrind a binary instrumenter, this means that not only variable writes are included, but also function arguments and return values, as they have to be pushed into the stack.

As a first test, we ran the experiment monitoring only memory reads with bitmask invariants. For the second experiment we enabled the read monitoring. Range invariants were introduced in the third experiment. The fourth and last experiment tried to show the combined power of both invariant types and whether there was a big gain in using both methods at the same time. The results of the four experiments can be seen in Figures 6.1 to 6.7 For each program there is a graph that shows the performance of each version under the four different screening conditions. The first and the second bars show the performance of the bitmask invariants when only writes are being monitored and when both reads and writes are being monitored respectively. The third bar represents the performance of the range invariants for both reads and writes. Finally, the last bar is the combined performance of both invariants.

As can be seen in the graphs, the performance of the only-writes bitmask invariants was not very good, whereas the introduction of memory reads and range invariants provided a remarkable improvement in versions 3 and 7 of print_tokens, version 2, 3, 5 and 8
6.1 First experiments

Experiment Results

of print\_tokens\textsuperscript{2}. On the other hand, there are no cases where the combined powers of both invariants provide a significant improvement in detection accuracy over their separate performances. There are, however, some strange cases in where the combined performance is less than the separate. For example versions 1, 4, 6 or 9 of print\_tokens\textsuperscript{2}. The reason for this can be found in what we will call the pointer problem, where a change in the base address of the program stack or heap blocks can cause unpredicted invariant violations (see Section 6.2.1).
Experiment Results

6.1 First experiments

**Figure 6.5:** schedule2

**Figure 6.6:** tcas

**Figure 6.7:** tot.info
6.1 First experiments

**Experiment Results**

**Memory reads are not enough**  Instrumenting memory writes alone was not enough to get a good performance. The following code from the correct version of `print_tokens` shows why:

```c
277    while(check_delimiter(ch)==FALSE)
278   {
279      if(token_ind >= 80) break; /* Added protection - hf */
280      token_str[token_ind++]=ch=get_char(tstream_ptr->ch_stream);
281   }
```

The buffer `token_str` is being filled up to a maximum length of 80 characters. If it’s longer, it won’t be accepted as a valid token. However, faulty version number 7 contains this code instead:

```c
277    while(check_delimiter(ch)==FALSE)
278   {
279      if(token_ind >= 10) break; /* Added protection - hf */
280      token_str[token_ind++]=ch=get_char(tstream_ptr->ch_stream);
281   }
```

In this case only tokens shorter than 10 characters will be accepted because of the changed code on line 279. No memory write takes place in there, just a read. The only way to detect that error is enabling the memory read screening.

**Imprecision of bitmask boundaries**  Although in the second run memory reads were enabled, the error was still not being detected. The reason for this is in the very nature of bitmask invariants. Exact range boundaries cannot be represented with a bitmask invariant. Moreover, the lower bound is always zero, and the upper bound will be on the worst case a power of two (See table 6.1). In the case of `print_tokens v7`, the invariant will accept values up to 15, even if 11-15 are incorrect. Range invariants provide much tighter boundaries for the admitted values of a program point. After enabling the range invariant screener, the error was caught.

<table>
<thead>
<tr>
<th>First</th>
<th>Mask</th>
<th>Seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>0101</td>
</tr>
<tr>
<td>0101</td>
<td>1111</td>
<td>0110</td>
</tr>
<tr>
<td>0101</td>
<td>1100</td>
<td>0111</td>
</tr>
<tr>
<td>0101</td>
<td>1100</td>
<td>1000</td>
</tr>
<tr>
<td>0101</td>
<td>0000</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.1: Evolution of a 4-bitmask invariant. Although only values between 5 and 8 are seen, after the training the bitmask will accept any 4-bit value.
**Floating point and bitmasks** There is one more problem related to the nature of the bitmask invariants: floating point values. The bits of a 32 or 64 bit floating point number have different meanings that depend on whether they are in the exponent, the mantissa, or the sign. Moreover, the same mantissa represents a different number if the value of the exponent is different. The bitmask keeps very little information about the nature of the floating point values that have been observed. Schedule and schedule2 use floating point variables, and many of the seeded faults are floating point related. This is one of the possible causes of the low performance of the method for these two programs.

**Range invariant’s learning speed** The range invariants were not free from problems. The fault in print_tokens version 10 involves an infinite loop that causes a segmentation fault in the end.

```c
380 while (*str!='\0') { /* until meet the token end sign */
381   if(*str-i]=='"')
382     return(TRUE); /* meet the second '"' */
383   else
384     i++;
385 }
```

For every iteration of this loop, the variable i will be incremented, producing a violation of the range invariant in line 384. When i is big enough, it will fall outside the allowed memory space and cause a segfault. Before the program is terminated, hundreds of thousands of violations are reported, producing more than 5Gb of log files.

Counters are a worst case scenario for range invariants in learning speed. Every increment of the counter will cause a violation until the upper limit is reached, which means a number of violations in the order of $O(2^k)$, where k is the number of bits used to represent the boundaries. On the other hand, bitmask invariants have a much lower number of violations. Their worst case scenario is one where each violation will unset just one bit. This will cause a number of violations in the order of $O(k)$.

### 6.2 Second experiments

The second round of experiments, this time using the second version of Delftgrind, consisted on evaluating the learning speed of the invariants and the speed at which they become too general and start missing errors in the program. At every 5% point in the training phase, the invariant accuracy was tested for both the bitmask and the range invariants using the checking set.

As a secondary objective, we wanted to check whether stopping the spectrum generation just after the error is first found has any beneficial effects in the fault diagnosis. A program spectrum generator was added to Delftgrind with this purpose. Stopping the program just after the first error is reported would theoretically provide an improvement in the quality of the fault diagnosis by eliminating all the statements that hadn’t been reached by the time the warning is raised.
As can be seen in Figures 6.8 to 6.14, the learning speed of the invariants is very high, falling under 5% of false positives after just a 5% of the training has been done. The false negative rate grows fast as well, but never as fast as the decrease speed of the false positives. This is important, because if we wanted to use this method as it is right now to detect errors, we would like the true positives to be visible. If there was a high false positive rate, the errors that were detected would be buried in a high number of false positives, making almost impossible to tell which ones are true and which ones are false.

The graphs show that the false positive rate for the bitmask invariants is always lower than for the range invariants, which confirms experimentally the difference in the learning speeds of both methods. In return for this slower learning speed, the higher precision of the ranges allows them to produce lower false negatives.

However, we are not looking for a method that allows us to find all errors with high accuracy, but to provide one of the inputs for the spectrum based fault localization, not as a fault localization mechanism by itself. It still has to be proved, but previous experiments have shown that the spectrum fault localization can afford a high number of false negatives without affecting the quality of the detection significantly. Our next objective will be to improve the value of $f_n$ while maintaining the value of $f_p$, and further evaluating the performance of our error detector as input of the program spectrum fault detector. In order to do that, some issues that were found during the experiments will have to be addressed.
Experiment Results

6.2 Second experiments

Figure 6.10: replace

Figure 6.11: schedule

Figure 6.12: schedule2

Figure 6.13: tcas

Figure 6.14: tot_info
6.2.1 The pointer problem

Many of the variables in a program are not real values (integers, reals, characters), but memory addresses, i.e. pointers. Until now we have been handling them as if they had the same semantics as integers, but it is clear that they need a separate treatment. First of all, there is no point in watching the range of a pointer (unless it’s an array indexing pointer), or if it has been odd or even. The most important characteristic of a pointer is whether it is null or not and whether it contains a valid memory reference or it will cause a memory protection fault when dereferenced.

On addition, the same pointer can have different values in different executions. So far we have assumed that the same program under the same input should produce no invariant violations on a second execution. However, the region of memory where a program is stored varies from run to run, making the pointers violate some of their invariants, no matter how many times that test case has been trained.

Pointer violations can happen before the real fault has been executed, thus rendering the early spectrum obtained absolutely useless, as the faulty line would be marked as not run. Furthermore, some of the test cases that are not false negatives due to pointer violations that should not be there. This is specially relevant in totinfo (Figure 6.14). Therefore, we need to handle pointers in a different way to avoid all these complications.

Our first approach is to try to rule out of the instrumentation all the instructions that were handling pointers, but it is impossible in Valgrind, as pointers and integers are treated the same way. Our second thought was to look in the original binary of the program, decoding the original instruction that originated the Valgrind IR. This second approach also was not successful because the x86 instruction set also doesn’t make distinction between pointers and integers except for some cases. The last resort was to make use of the debugging symbols of the program. Unfortunately this didn’t work either. Debugging symbols are designed to provide a way to find the address of a variable given it’s name for any given point of program execution, and not the other way around.

6.3 Delftgrind’s Performance as Fault Localization Input

Although the instrumentation still had the serious aw of the pointer pointer, we selected a number of versions that did not exhibit this problem to do a quick test of the performance of our error detection information with different similarity coefficients.

In addition to producing the error diagnostic, we modified Delftgrind’s spectrum generator to produce early error detection spectra. Early error detection spectra are spectra whose generation was stopped just after the first error was detected (after the first invariant violation). We had the theory that stopping early will eliminate many lines from the fault candidates, contributing to the improvement of the value of \( q_d \). Table 6.3 shows the results for the selected versions.

As can be seen, even though print_tokens improves its results in good measure, the other program’s results are extremely poor with the exception of tcas v40 and tot_info v10. The reason for this is that the first violation happens before the faulty line is actually
executed, leaving a 0 in the spectrum line corresponding to it, making the fault locator ignore it.

The early invariant triggering may be caused by an unexpected input, precisely the input that will cause the faulty line to produce an error. As the program has not been trained with that input, some of the invariants might be tight enough to catch that unusual input before the execution reaches the faulty line.

Another reason for the triggering of an invariant before the faulty line is really executed is that the code path that leads to the faulty line has never been executed. This is yet another issue that we need to take into account if we are going to stop the program after the first error is found. It can be seen in the following piece of code.

```plaintext
001 if (flagVar == True) {
002     a = 10; /* Correct code */
003 } else {
004     a = 11; /* Incorrect code */
005 }
006
007 foo(a);
```

During the training only the correct code has been executed, which means that flagVar has always been true. For the faulty line to execute, flagVar has to be false, which will cause an invariant violation in line 001, before the real fault is executed. This means that line 004 will never be touched, and the fault detector will not consider it as a possible candidate.

### Table 6.2: Delfgrind’s early error detection (D) performance, compared to the failure error detector (F).

<table>
<thead>
<tr>
<th>Program</th>
<th>Version</th>
<th>Tarantula F</th>
<th>Tarantula D</th>
<th>Jaccard F</th>
<th>Jaccard D</th>
<th>Zoltar F</th>
<th>Zoltar D</th>
<th>Ochiai F</th>
<th>Ochiai D</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>v1</td>
<td>18</td>
<td>18</td>
<td>18</td>
<td>15</td>
<td>2</td>
<td>37</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>print_tokens</td>
<td>v2</td>
<td>16</td>
<td>9</td>
<td>15</td>
<td>8</td>
<td>7</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>print_tokens</td>
<td>v3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>print_tokens</td>
<td>v7</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>53</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>print_tokens2</td>
<td>v2</td>
<td>1</td>
<td>507</td>
<td>1</td>
<td>507</td>
<td>1</td>
<td>507</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>print_tokens2</td>
<td>v3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>print_tokens2</td>
<td>v5</td>
<td>0</td>
<td>515</td>
<td>0</td>
<td>515</td>
<td>0</td>
<td>515</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>v1</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>v5</td>
<td>4</td>
<td>108</td>
<td>4</td>
<td>108</td>
<td>4</td>
<td>133</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>v6</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td>109</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>tcas</td>
<td>v36</td>
<td>0</td>
<td>126</td>
<td>0</td>
<td>126</td>
<td>0</td>
<td>126</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>tcas</td>
<td>v40</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>10</td>
<td>16</td>
<td>6</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>tot_info</td>
<td>v10</td>
<td>77</td>
<td>51</td>
<td>77</td>
<td>51</td>
<td>67</td>
<td>46</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>tot_info</td>
<td>v11</td>
<td>2</td>
<td>102</td>
<td>2</td>
<td>102</td>
<td>2</td>
<td>102</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Experiment Results  
6.3 Delfgrind’s Performance as Fault Localization Input
This is interesting, as it means that maybe stopping the spectrum generation after just
the first error is not always the right approach and that we should wait until more violations
are reported. In the previous example, until the invariant for the variable $a$ in line 007 is
checked.

6.4 Third Experiments

Our third round of experiments were conducted using the new Ninjin instrumentation. Without
the effects of false pointer violations, the true performance of the dynamic error detec-
tion method and the cases when it does work and the cases when it doesn’t can be finally be
seen. Figures 6.15 to 6.22 show the results for the experiments, and an averaged

Without no more pointer issues, the false positives decrease even faster, reaching zero
after a 50% of the training set has been used. However, the false negative rate hasn’t im-
proved, furthermore, it seems that in some cases it has worsened.

Figure 6.15: print_tokens

Figure 6.16: print_tokens2

Figure 6.17: replace

Figure 6.18: schedule
Experiment Results

6.4 Third Experiments

Figure 6.19: schedule2

Figure 6.20: tcas

Figure 6.21: tot_info

Figure 6.22: average
**Contaminated training set**  A very important issue concerning schedule2, which had a very poor performance, has to be noted. Looking at the code of version 1 for example, shows that the kind of seeded fault that has been inserted is detected in other programs (print_tokens2 has a very similar fault) but not in this case.

```c
129 int upgrade_prio(prio, ratio)
130 int prio;
131 float ratio;
132 {
133     int status;
134     struct process * job;
135     /* if(prio < 1 || prio > MAXLOPRIO) return(BADPRIO); */
136     if((status = get_process(prio, ratio, &job)) <= 0)
137         return(status);
138     /* We found a job in that queue. Upgrade it */
139     job->priority = prio + 1;
140     return(enqueue(prio + 1, job));
141 }
```

All the cases that do not cause an error (when `prio` is inside bounds) should train the invariants for `prio` in lines 136 to 140 to not accept values lower than 1 and higher than `MAXLOPRIO`. The moment a case from the testing set tries use an out of bounds value for `prio`, all the alarms should go off.

However, this is not happening, due to the way the cases were separated into the training and testing sets (See Section 5.2 and [1, 2]). An error on the program does not mean that the output will be incorrect. If the output of the program is very simple (in schedule2 it is just a number), there is a big chance that even with an error the program will produce a correct output, i.e. there won't be a failure.

This is precisely the case of schedule2's training set. Some of the training cases contain the error but don't fail, thus making the invariant in lines 136-139 accept erroneous values.

**Incorrect single values**  In many versions, the return value of a function is mutated for some of the faulty cases. If the value is still returned for some of the correct inputs, the invariant will not be able to detect it. Moreover, if the value returned is different but happens to fall in the set of accepted values, it will not be caught either.

In the following piece of code the return value in line 6 will not violate any invariant because it is a valid return value used in line 10.

```c
001 char esc(char *s, int *i) {
002     char         result;
003     if (s[*i] != ESCAPE)
004         result = s[*i];
005     else if (s[*i + 1] == ENDSTR) {
006         result = NEWLINE; /* constant mutation ESCAPE; */
```
Relationships between variables

The performance of the method for replace is not good either. After an examination of the source code of some of its versions, the reason for this lack of effectivity is clearer, and it’s deeply related with the biggest limitation of the method so far: the invariants cannot represent relationships between variables.

Most of the versions that perform well have invariant violations that involve constant values in the source code. The problem is that there are many other versions where the error is not in the value itself, but in the relationship between two or more variables. For example there might be a part of the program where two variables are equal throughout all the training and suddenly during the checking they are not. Our invariants as they are now are absolutely unable of recording this property.

In the following piece of code from replace version 3, the check on line 494 has been removed. If we assume that during training, each time lines 495 and 496 are executed the values of lastm and m are different, a hypothetic invariant that checks the relationship between lastm and m (like the ones used in DAIKON [6]) will catch the error when the lines are executed and lastm and m have the same value. Unfortunately, our invariants are incapable of reflecting this property and the error will go unnoticed.

```
493 m = amatch(lin, i, pat, 0);
494 if ((m >= 0) /* && (lastm != m) */) {
495   putsub(lin, i, m, sub);
496   lastm = m;
497 }
498 if ((m == -1) || (m == i)) {
499   fputc(lin[i], stdout);
500   i = i + 1;
501 } else
502   i = m;
```

Loop variants

Sometimes the error is not in the value of the variable, but in the way it is modified from pass to pass. We can have a loop with a counter that it is always incremented by 1, but that due to an error sometimes it is incremented by 2. Version 13 of replace has this precise problem.
This error can be solved implementing the full version of the the invariants, adding the bitmask or ranges for the difference between the previous and current values. Still we would have to solve the problem of what to do each time the function is called, even across different runs of the program. If we compute the difference between the value of \( i \) on line 491 in the last iteration of the last invocation, and the first value of \( i \) in the first iteration of the next call to the loop, it will make no sense because we are in a different context and the delta will start accepting large values that are not the real loop variant.

**Locality of errors** In Section 1.3.1 we stated that an error will propagate causing more errors across component boundaries [14]. If errors propagated and replicated, the probability of one of the invariants being violated should be high. Even if the original error caused by the fault is not detectable by our current means, some of the derived errors might be detectable.

Even if this might be true for big systems, our test suite does not seem to exhibit this kind of behaviour. When processing a stream of text as replace does, faults cause the input to be skipped or not be accepted as if it was an error in the input. Thus, the program just skips the “incorrect” input and continues as if nothing had happened.

The previous example for loop variants also serves as example for this kind of problems. Sometimes one character of the input is skipped, which makes the input not to be accepted. Unless the training set contained no test cases with incorrect inputs, the model of the system will see perfectly normal for the input to be incorrect.

### 6.5 Ninjin’s Performance as Fault Localization Input

In this section we will present the most important results of the Thesis. In the previous section we have seen that there are many cases where the unary invariants are not very good performing and have false negative rates of almost 1. However, we have already stated that
we are not looking for a good error detector, but for an error detector that produces a good enough input for the fault locators. This means that even if the false negative rate is high, it will be good for us if the results of the fault locators are good.

### 6.5.1 Performance without False Positives

Previous works [2] have shown that error detectors with a high false negative rate, obtained by manually removing part of the working lines from the fault detection matrix, don’t have a very big impact on $q_d$.

Our first fault location experiment will consist on evaluating $q_d$ for every version, now with the Ninjin error detector after full training instead of the failure based error detector. As we will train the detector with the complete training set, there will be no false positives.

The results of this experiment can be seen in Figures 6.23 to 6.29 and in further detail in Tables A.1 to A.6 including the false negative rate. The versions where no error was detected, and thus $f_n$ was 1 and no fault location could be performed, are not included in order to save space.

The error detection input to calculate these values is the combined output of both bitmask and range invariants. In the next section the results are separated into bitmask and range invariants.

As can be seen, the relative value of $q_d$ doesn’t change considerably in most of the cases for Zoltar and Ochiai, the most advanced similarity coefficients, even when the false negative rate is very high. For example, despite version 6 of `print_tokens2` has a $f_p(100) = 0.99$, the value of $q_d$ remains unchanged. There are even some cases in which the presence of false negatives improves the value of $q_d$, such as for `schedule v7`.

![Figure 6.23: print_tokens](image1)

![Figure 6.24: print_tokens2](image2)
6.5 Ninjin’s Performance as Fault Localization Input

Experiment Results

Figure 6.25: replace

Figure 6.26: schedule

Figure 6.27: schedule2

Figure 6.28: tcas

Figure 6.29: totinfo
6.5.2 Performance with False Positives

The results from the previous section are far too optimistic. In a real situation, the training is not going to be able to cover every possible correct run, and thus, the presence of some false positives will be unavoidable. To replicate this situation, we will train the invariants in steps of 5 percent points and then use the remaining training cases and the checking cases to obtain an error diagnostic input file for the fault locator. When no errors are reported, and therefore no ranking can be constructed, we assume that the faults are uniformly distributed and assign \( q_d \) an expected value of 0.5.

The results of this experiment can be seen in Figures 6.30 to 6.37. More detailed results are available in the Appendix, in Tables A.8 to A.15.

The first conclusion that can be extracted from the graphs is that the range invariants consistently outperform the bitmask invariants. Despite their slower learning speed and thus higher false positive rate, their lower false negative rate seems to play a more relevant role when it comes to fault localization.

We can extract a second conclusion regarding the performance of the similarity coefficients. Even though the Zoltar coefficient had the best overall performance when no false positives were present, i.e. after the training is 100% completed, it seems to be the most affected by all their presence. This effect is strong enough in some cases, such as in schedule, to make the Zoltar coefficient for ranges the worst performer of all, even worse than Zoltar for bitmasks, which has a lower false positive rate.

All the graphs, including the average, show a drop in the quality of the fault detection after 20% of the training. The explanation for this is the rapid increase in false negatives in the first stages of training (in fact, all after a 20% of training, the false negatives increase very slowly).

\texttt{replace} shows a decreasing trend, but suddenly that trend reverses and starts increasing. This is caused by our decision of assigning an expected value of 0.5 to \( q_d \) when no fault localization can be performed because there is a 100% of false negatives. The value of \( q_d \) for a given version can be very low, as long as the value of \( f_n \) for that version stays under 1. However, when the value of \( f_n \) is equal to one, the average \( q_d \) will be pulled upwards by the expected values that we inserted.

The case of \texttt{print_tokens} and \texttt{schedule} are somehow special due to its really low false negative rates. In these programs we can see how false positives affect the quality of the fault diagnosis. All coefficients exhibit a growing trend throughout the training that proves the adverse effects of false positives in similarity coefficients. Once again we see the bad performance of the Zoltar coefficient in the presence of false positives, and its quick improvement in the moment they disappear.
6.5 Ninjin’s Performance as Fault Localization Input

Experiment Results

Figure 6.30: print_tokens

Figure 6.31: print_tokens2

Figure 6.32: replace

Figure 6.33: schedule

Figure 6.34: schedule2

Figure 6.35: tcas
Experiment Results

6.5 Ninjin’s Performance as Fault Localization Input

Figure 6.36: tot_info

Figure 6.37: average
Chapter 7

Related Work

On this chapter we will present the different approaches that have been used until now regarding dynamic invariants. Dynamic program invariants have been previously used and published \[8, 6, 25\], but most of this previous work was focused directly in fault diagnosis \[25\]. Our research on the contrary, pursued a different objective: detecting the presence of an error in the execution of the program and feed it (just a yes/no value) to the fault diagnoser.

7.1 Invariant related work

Previous work on this area tried to use the errors reported by the invariant violations to try to directly find the cause of the error (i.e. the fault) \[6, 8, 25\]. Our approach is slightly different, using the information about the presence or absence of error as an input for a new method based on program spectra \[31, 1\].

7.1.1 DIDUCE

Our work with bitmask invariants is inspired by the DIDUCE dynamic invariant error detector that was developed by Sudheendra Hangal of Sun Microsystems and Monica Lam from Stanford University \[8\]. Furthermore, our experiments have been based on one of DIDUCE’s proposed usage models: debugging programs that fail on some inputs.

DIDUCE consists in a dynamic invariant detector plus an invariant checking engine for Java programs. The instrumentation is done at bytecode level, so the source code of the program is not required, but due to Java’s bytecode high level nature, very few information is lost. In addition, some of the errors that are present in programs of the Siemens set are impossible to have in Java code, such as buffer overruns, because array accesses in Java are bound checked.

The expressions tracked by DIDUCE are reads and writes of object variables, reads and writes of static variables, and procedure call arguments and return values. For each of the tracked expressions, a bitmask invariant is associated. In the case of reference types (objects) instead of recording their value, the invariants track if it has been null or non-null.
DIDUCE was tested on four real-world applications with successful results. However, in all of the cases the error was caused by a variable that had a constant value throughout the training phase and that in the faulty cases had a different value. In our opinion, those test cases are not enough to evaluate the method’s performance, which has been one of the motivations for this research.

Their bitmask invariants include a bitmask for the variation between consecutive values, but in the kind of practical examples they provide, the delta invariant is not relevant. Furthermore, we believe the delta needs some refinement to correctly work in loops that are run more than once, when the delta would be the last value of the previous execution subtracted from the first value of the current execution of the loop.

### 7.1.2 Racunas’ screener

In a paper presented in the IEEE 13th International Symposium on High Performance Computer Architecture in 2007 [26], they present a study about the performance of different screeners when detecting hardware faults instead of software faults.

Their random fault seeding mechanism cannot be used for our study, as it just randomly alters the results of the instructions of normal programs such as GCC, bzip2 or the SpecInt benchmark.

Five different screeners are presented in the report:

- An extended history screener that stores 64 unique observed values and 64 unique deltas between consecutive values. Instructions that cause violations in more than one percent of the times they are executed are filtered. This method has a very fine detail if the set of values is small, however, the method will perform poorly when the set of acceptable values grows too much. There’s no word about the replacement strategy used for the values.

- A dynamic range screener in which we based our range invariants. The version presented in their paper is more sophisticated as it has multiple adjustable intervals and stride calculation. When all the range points are full, new values cause a rearrangement that will add the less possible extra values to the interval.

- A bitmask screener similar to the one in DIDUCE, including the delta bitmask.

- A TLB miss screener. Every time a page fault is triggered it is considered a fault.

- A Bloom filter based screener that hashes the instruction address and result several times. The hashes are used as addresses into a table that stores if a particular instruction-value pair has already been observed.

### 7.1.3 Daikon

The first two previous works have one important characteristic in common which our experiments share as well: they only extract unary invariants, i.e. no relationships between program variables are taken into account. This approach, although being the most lightweight,
is very limited as many faults cause errors that involve relationships between variables. However, relationships between pairs or triplets of variables produce a combinatorial explosion that is not affordable within an embedded environment.

Daikon [6, 7] follows the mentioned approach of extracting invariants over relationships between variables in program code in either C, C++, Java or Perl. It can check for 75 different invariant types, which makes it very powerful but also quite slow.

Daikon can produce invariants over call parameters and return values, pre-state values, global variables, object or struct fields, results of calls to side-effect-free methods and 25 different derived values. All these can be combined creating compound variables.

Daikon instruments procedure entries and exits, which means the invariants produced will correspond to preconditions and postconditions of functions. Object or class invariants are also extracted from what is called object point. The object point extracts object wide invariants that are observed at entry and exit points of public methods of a class, and that are passed into or returned from methods of other classes, or that are stored in object fields.

### 7.1.4 Carrot

Carrot [25] is a similar approach to Daikon’s vision, but in a more lightweight fashion using a smaller set of invariants. In fact it uses Daikon’s instrumentation engine for the extraction of the execution traces.

In the cited work, a methodology is presented that is related to our own. First the model is trained using a set of known good runs until it converges to a steady state. Then a set of known bad runs is checked and the results are contrasted to see if the difference can provide any clue about the origin of the error. The Siemens set [9] is also used to perform their experiments.

They conclude that the approach is not good enough to catch all the errors in the Siemens suite possibly due to flaws in the set of invariants that are implemented in Carrot, or because potential invariants are not suitable for debugging.

Our approach differs from Carrot in two aspects. First of all, our invariants are unary meanwhile Carrot uses relationships between variables for its invariants. And most importantly, our approach does not use the error information (violations in the invariant model) to locate the fault directly but as a input to another fault location technique.

### 7.2 Other dynamic error detectors

So far we have presented related works in error detection that are based on program invariants, which are data and control flow related, and therefore are close to the problem domain. However, some errors are less related to how the problem is interpreted but to how the solution is built. Memory errors such as buffer overflows and memory leaks, as well as synchronization issues such as race conditions and deadlocks cause programs to fail. There has been previous work in error detection techniques that help detect these kind of errors, some of which are well known and reputed.

Valgrind, the instrumentation tool we have used on the first stages of the research, comes with a memory error screener, Memcheck, that can report every leaked memory block in a
7.2 Other dynamic error detectors

Valgrind was also used as the base for an experimental bounds checker [22], but as its author admits, it didn't perform very well as Valgrind’s binary level instrumentation is missing many vital source level information about what is a pointer and what not. We faced this same problem and it is what moved us away from Valgrind in favour of LLVM.

Monica Lam worked in yet another error detector, a dynamic memory overflow detector called CRED [28] that used source level information. CRED determines pointer bounds by finding their referent object by means of an object table, a runtime structure that collects all the base address and size information of all static, heap and stack objects. Their work was based in a previous GCC extension by Richard Jones [13].

In the field of synchronization errors, Savage et al. published in 1997 Eraser [29], a tool that implements the Lockset algorithm for race detection. The algorithm enforces a simple locking discipline in which every shared variable has to be protected by a lock. As there is no way of knowing which locks protect which variables, this information has to be inferred from the execution history.

Deadlock detection has been a field of study for long time. Bensalek and Havelund have recently presented a tool [4] that detects potential deadlocks in running programs from their execution trace by building a lock graph and examining its cycles. Although their tool is Java oriented, the algorithm is general enough to be applied to programs written in C or C++.
Chapter 8

Conclusions and Future Work

This chapter gives an overview of the project’s contributions. In addition, this overview, we will reflect on the results obtained in this Master’s Thesis and draw some conclusions. Finally, some ideas for future work will be discussed.

8.1 Contributions

During the time span of this Master Thesis, two instrumentation engines have been provided to further develop and study dynamic error detectors. The first one was a Valgrind-based tool [24], and as such it was binary based and had a series of flaws, some of them impossible to overcome. However, even if flawed, binary instrumentation could be the only possible way to proceed if, for example, the source code of the program was not available and therefore performing a source-based instrumentation would be impossible.

The second instrumentation engine that has been contributed uses LLVM [16]. It solved many of the issues associated with the previous binary instrumentation, while remaining source code independent by means of an intermediate representation with a higher level of abstraction than binary code. Despite the instrumenter has only been tested with ANSI C code, it should be possible to use the current instrumenter with C++ and Objective C code, as well as more sophisticated C projects, without a single modification. This second instrumenter has the great advantage that it is able to produce instrumented binaries for many architectures or even C source code, providing a great portability as well as a performance boost over the Valgrind approach.

Our work with the instrumenters has allowed us to experiment with two types of dynamic invariants: bitmask invariants and range invariants. Our experiments have shown the strengths and weaknesses of both types of invariants, and even any other type that falls in the family of unary invariants.

Finally, we have presented the full picture of a completely automated error detection and fault diagnosis method that works without any domain specific information and allows us to ignore between a 60% and 80% of the program when searching for a fault. Our most important contribution in fault location has been comparing the effects of false positives in
the different similarity coefficients that have been used by the department’s previous work [1], something that hadn’t been done before.

8.2 Conclusions

8.2.1 Instrumentation

Binary instrumentation has some serious drawbacks that cannot be overcome easily. The compilation process of a program removes almost all the high level information from it, the most important of all being the typing information. Without typing information, the ability to distinguish between signed and unsigned values, or values from memory addresses is lost in many cases. The first is needed for the comparisons in range invariants to correctly work, while the latter is needed to avoid the pointer problem.

Regarding the pointer problem, one may think that if we could force the operating system or runtime monitor to load the program in the same area of memory, and completely avoid the use of dynamic memory allocation, the problem would be solved. However, this is not true. The stack is also a dynamic structure that grows and shrinks. The values of pointers to automatic variables depend on the concrete situation of the stack, and can change very easily. The only improvement we would get imposing all these limitations is that a program under the same input conditions would not cause any violations when run for a second time.

The ideal level at which the instrumentation should be done is the original source level, before any compilation is done. However, source-level instrumentation has its own drawbacks as well. Source-level instrumentation requires a working parser of the language. For some languages with simple, unambiguous grammars this is not a problem. However, C’s grammar, for example, is neither simple nor unambiguous. Moreover, even though there is a standard reference grammar, every compiler vendor has its own extensions and interpretations of some aspects of it, which makes achieving universal support almost unfeasible.

Our second way of tackling the problem of instrumentation provides a good level of compromise between source and binary-level instrumentation. By instrumenting an intermediate representation with a lower level of abstraction, we lose some information such as control-flow statements (which are converted to jumps) but we gain a simplified code that is much easier to instrument. Bytecode instrumentation is not exclusive of LLVM. Java’s Virtual Machine bytecodes can be instrumented as well with successful results as shown by DIDUCE [8]. As multiple high level languages can share the same intermediate representation (C# and Visual Basic.Net, for example), all of them will benefit from the work done.

8.2.2 Error Detection and Fault Diagnosis

Even though both types of invariants learn extremely fast, they are far from perfect, even from good, and will very likely never be good enough due to their biggest limitation: they are incapable of reflecting relationships between variables. Many of the errors present in the Siemens set are equality or inequality relationships that will go unnoticed.
However, introducing more sophisticated invariants such as the ones Daikon has will require way more storage and processing power than an embedded system can provide. On one hand, unary invariants are very easy to represent and check, and take very little storage space (around 76 bits each, 64 bits if we removed the usage count). A single program can contain a number of instrumentation points in the order of hundreds or thousands, which means we could need around 64Kb of storage. Considering relationships between variables needs, in the first place, a more complex representation that will take more space will be needed. Supposing that in every program point there was an average of 5 variables in scope, this would give us an average of 10 possible of combinations to be checked per program point. For each combination there are 6 situations (equal, different, strictly greater, strictly lower, greater or equal, lower or equal) that will need to be checked. Supposing we need 128 bits to represent one invariant, the total cost in storage space would be 7680 bits per program point, in other words, almost one megabyte. This is a cost in storage not all embedded systems can afford.

The speed of the learning process seems to have an influence on the accuracy of the invariants. A highly accurate invariant will need more training to achieve a low false positive rate. Bitmap invariants are very fast learners because every time we have a violation we unset at least one bit of the bitmask, which in turn means that the number of accepted values will be doubled. This is the reason behind the speed of the bitmask’s learning process and the main reason for their inaccuracy. On the other hand, range invariants can add single values to their set of accepted values if they are consecutive, this means that they will have a greater accuracy, but at the price of a higher number of false positives.

Every approximate method is imperfect. Therefore a balance point between training (false positives) and detection power (false negatives) has to be established. In the case of dynamic invariants, after a 20% of the training has been completed, the false negative rate is quite stable, whereas the false positives need to reach at least the 50% mark. Looking at the fault location graphs, after the 50% there’s a clear decreasing trend, (except for those programs that have errors that can be more easily detected by our invariants), which suggests that in that point is where the best balance is achieved. Of course, if we had some kind of information about what is the nature of the faults we are looking for (if we are looking for errors like those present in print_tokens), we could choose to increase the training in order to improve the quality of the detection.

A very important note about our experiments is that the false negative and false positive rates’ values are not precise. The reason for this is twofold. First of all, the error detector used as reference is not perfect and some of the cases used during the training may actually contain the error. As our method is so sensitive to noise in the training, the ideal results may be altered. And secondly, when one of the invariants is violated during the checking, we don’t have an easy way to know if that violation is due to an error that has been caught or to insufficient training in other parts of the program (leaving unexplored code paths and unexpected inputs aside).

As we have already stated, ranges have a superior performance in terms of false negatives due to their higher accuracy with respect to bitmask invariants and this comes at the price of a higher false positive rate. This effect is something that has to be taken into account when choosing the similarity coefficient for the fault diagnosis part. If our training
set is fairly complete and we are confident the false positive rate will be low enough after the training has been completed, we can use the Zoltar coefficient. However, if the training set is not complete enough, we should avoid Zoltar and opt for Ochiai instead, as it seems to tolerate better the presence of false positives.

8.3 Future work

Bitmask invariants and range invariants are just some of the possible invariants our screener can use. They can be extended, for example adding a refined version of the the delta invariant that the original versions of the invariants were using [8, 26]. Some of the other invariants that can be found in [26] could be implemented, such as the Bloom filter invariants, the history invariants or the multiple range invariants.

Every instrumentation adds an overhead in terms of time and space. As the method is designed to be used in an embedded system where both resources could be a real concern, further studies should be carried out to find out what would be the effect on the technique of a limitation on the number of program points being instrumented.

The issue of the sensitivity to noise should be addressed. A way of finding out the relevance of a violation, in order to decide if an error has occurred or it’s just a relaxation for training. Bitmasks already provide a confidence level formula that could be used in this process, and we have already proposed (although not tested) a similar formula for range invariants.

The current code for the instrumenter is not very good quality. There are many repetitive functions that can very possibly be refactored. LLVM optimization passes are implemented in C++, therefore coming up with an extensible architecture that allows the addition of different program point as plugins shouldn’t be very difficult. The code for the screener was ported from inside Valgrind and is implemented in C. A new version of it implemented as well in C++ could allow even more code reuse and a more refined and scalable architecture. The screener could include its own spectra generator and extensible fault location algorithms.

From a more formal point of view, dynamic invariants are nothing else than a machine learning technique used as classifier. There’s already work in that area [20], with evaluation and classification criteria. It would be desirable to study the role of dynamic invariants inside the context of machine learning techniques and their relationships with other techniques. In other words, elaborate a formalization of these techniques.

On Section 5.3.2 we left open the issue of finding a suitable formula for measuring the quality of the error detector \( q_e \). Such formula should include information from both the false positives and false negatives. The false positive and false negative rate are related to the set of passed runs and to the set of failed runs respectively. \( q_e \) on the other hand should be related to the checking set, which includes both passed and failed runs.
Bibliography


Appendix A

Result tables
Table A.1: Fault location for print tokens at full training, comparing the failure error detector (F) with the nijnin error detector (N).

<table>
<thead>
<tr>
<th>Version</th>
<th>Tarantulla</th>
<th>Jaccard</th>
<th>Zoltar</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_n$</td>
<td>F</td>
<td>N</td>
<td>F</td>
</tr>
<tr>
<td>v1</td>
<td>0.00</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>v2</td>
<td>0.95</td>
<td>0.95</td>
<td>0.75</td>
<td>0.96</td>
</tr>
<tr>
<td>v3</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v5</td>
<td>0.84</td>
<td>0.85</td>
<td>0.83</td>
<td>0.91</td>
</tr>
<tr>
<td>v7</td>
<td>0.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table A.2: Fault location for print tokens2 at full training, comparing the failure error detector (F) with the nijnin error detector (N).

<table>
<thead>
<tr>
<th>Version</th>
<th>Tarantulla</th>
<th>Jaccard</th>
<th>Zoltar</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_n$</td>
<td>F</td>
<td>N</td>
<td>F</td>
</tr>
<tr>
<td>v1</td>
<td>0.99</td>
<td>0.78</td>
<td>0.79</td>
<td>0.78</td>
</tr>
<tr>
<td>v2</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v3</td>
<td>0.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v4</td>
<td>0.99</td>
<td>0.93</td>
<td>0.86</td>
<td>0.99</td>
</tr>
<tr>
<td>v5</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v6</td>
<td>0.99</td>
<td>1.00</td>
<td>0.99</td>
<td>1.00</td>
</tr>
<tr>
<td>v8</td>
<td>0.00</td>
<td>0.75</td>
<td>0.75</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table A.3: Fault location for schedule at full training, comparing the failure error detector (F) with the nijnin error detector (N).

<table>
<thead>
<tr>
<th>Version</th>
<th>Tarantulla</th>
<th>Jaccard</th>
<th>Zoltar</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_n$</td>
<td>F</td>
<td>N</td>
<td>F</td>
</tr>
<tr>
<td>v1</td>
<td>0.98</td>
<td>0.94</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>v6</td>
<td>0.71</td>
<td>0.91</td>
<td>0.87</td>
<td>0.87</td>
</tr>
<tr>
<td>v8</td>
<td>0.62</td>
<td>0.65</td>
<td>0.68</td>
<td>0.68</td>
</tr>
<tr>
<td>v10</td>
<td>0.91</td>
<td>0.63</td>
<td>0.63</td>
<td>0.63</td>
</tr>
</tbody>
</table>

Table A.4: Fault location for schedule2 at full training, comparing the failure error detector (F) with the nijnin error detector (N).
## Result tables

<table>
<thead>
<tr>
<th>Version</th>
<th>Tarantulla</th>
<th>Jaccard</th>
<th>Zoltar</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_n$</td>
<td>F</td>
<td>N</td>
<td>F</td>
</tr>
<tr>
<td>v3</td>
<td>0.96</td>
<td>0.82</td>
<td>0.78</td>
<td>0.83</td>
</tr>
<tr>
<td>v4</td>
<td>0.96</td>
<td>0.82</td>
<td>0.78</td>
<td>0.83</td>
</tr>
<tr>
<td>v5</td>
<td>0.96</td>
<td>0.94</td>
<td>0.92</td>
<td>0.97</td>
</tr>
<tr>
<td>v13</td>
<td>0.07</td>
<td>0.82</td>
<td>0.82</td>
<td>0.83</td>
</tr>
<tr>
<td>v14</td>
<td>0.98</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v15</td>
<td>0.00</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>v18</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v19</td>
<td>0.33</td>
<td>0.74</td>
<td>0.79</td>
<td>0.74</td>
</tr>
<tr>
<td>v22</td>
<td>0.94</td>
<td>0.94</td>
<td>0.92</td>
<td>0.94</td>
</tr>
<tr>
<td>v25</td>
<td>0.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v26</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v27</td>
<td>0.00</td>
<td>0.86</td>
<td>0.86</td>
<td>0.87</td>
</tr>
<tr>
<td>v28</td>
<td>0.97</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v30</td>
<td>0.98</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v31</td>
<td>0.99</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table A.5: Fault location for replace at full training, comparing the failure error detector (F) with the nijnin error detector (N).

<table>
<thead>
<tr>
<th>Version</th>
<th>Tarantulla</th>
<th>Jaccard</th>
<th>Zoltar</th>
<th>Ochiai</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f_n$</td>
<td>F</td>
<td>N</td>
<td>F</td>
</tr>
<tr>
<td>v1</td>
<td>0.88</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v4</td>
<td>0.62</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v12</td>
<td>0.98</td>
<td>0.69</td>
<td>0.70</td>
<td>0.69</td>
</tr>
<tr>
<td>v16</td>
<td>0.90</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>v23</td>
<td>0.97</td>
<td>0.86</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>v24</td>
<td>0.71</td>
<td>0.86</td>
<td>0.86</td>
<td>0.86</td>
</tr>
<tr>
<td>v26</td>
<td>0.91</td>
<td>0.67</td>
<td>0.70</td>
<td>0.67</td>
</tr>
<tr>
<td>v28</td>
<td>0.97</td>
<td>0.90</td>
<td>0.86</td>
<td>0.90</td>
</tr>
<tr>
<td>v29</td>
<td>0.89</td>
<td>0.89</td>
<td>0.86</td>
<td>0.89</td>
</tr>
<tr>
<td>v35</td>
<td>0.97</td>
<td>0.90</td>
<td>0.86</td>
<td>0.90</td>
</tr>
<tr>
<td>v36</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v37</td>
<td>0.96</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v38</td>
<td>0.51</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v40</td>
<td>0.00</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
</tr>
<tr>
<td>v41</td>
<td>0.95</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Table A.6: Fault location for tcas at full training, comparing the failure error detector (F) with the nijnin error detector (N).
Table A.7: Fault location for tot_info at full training, comparing the failure error detector (F) with the nijnin error detector (N).

<table>
<thead>
<tr>
<th>Version</th>
<th>$f_a$</th>
<th>F</th>
<th>N</th>
<th>F</th>
<th>N</th>
<th>F</th>
<th>N</th>
<th>F</th>
<th>N</th>
<th>F</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>0.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>v2</td>
<td>0.00</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.95</td>
<td>0.95</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
</tr>
<tr>
<td>v4</td>
<td>0.09</td>
<td>0.90</td>
<td>0.91</td>
<td>0.91</td>
<td>0.91</td>
<td>0.97</td>
<td>0.97</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>v5</td>
<td>0.34</td>
<td>0.63</td>
<td>0.63</td>
<td>0.63</td>
<td>0.63</td>
<td>0.78</td>
<td>0.70</td>
<td>0.65</td>
<td>0.66</td>
<td>0.65</td>
<td>0.66</td>
</tr>
<tr>
<td>v6</td>
<td>0.28</td>
<td>0.64</td>
<td>0.62</td>
<td>0.64</td>
<td>0.62</td>
<td>0.77</td>
<td>0.76</td>
<td>0.78</td>
<td>0.63</td>
<td>0.78</td>
<td>0.63</td>
</tr>
<tr>
<td>v7</td>
<td>0.75</td>
<td>0.90</td>
<td>0.95</td>
<td>0.92</td>
<td>0.95</td>
<td>0.96</td>
<td>0.96</td>
<td>0.92</td>
<td>0.96</td>
<td>0.92</td>
<td>0.96</td>
</tr>
<tr>
<td>v8</td>
<td>0.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v9</td>
<td>0.86</td>
<td>0.80</td>
<td>0.86</td>
<td>0.80</td>
<td>0.86</td>
<td>0.94</td>
<td>0.89</td>
<td>0.83</td>
<td>0.88</td>
<td>0.83</td>
<td>0.88</td>
</tr>
<tr>
<td>v10</td>
<td>0.00</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.75</td>
<td>0.75</td>
<td>0.73</td>
<td>0.73</td>
<td>0.73</td>
<td>0.73</td>
</tr>
<tr>
<td>v11</td>
<td>0.09</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v12</td>
<td>0.00</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
<td>0.94</td>
<td>0.94</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>v13</td>
<td>0.71</td>
<td>0.90</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.96</td>
<td>0.96</td>
<td>0.92</td>
<td>0.96</td>
<td>0.92</td>
<td>0.96</td>
</tr>
<tr>
<td>v15</td>
<td>0.09</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>v16</td>
<td>0.58</td>
<td>0.77</td>
<td>0.66</td>
<td>0.81</td>
<td>0.66</td>
<td>0.85</td>
<td>0.85</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
<td>0.84</td>
</tr>
<tr>
<td>v17</td>
<td>0.09</td>
<td>0.90</td>
<td>0.90</td>
<td>0.92</td>
<td>0.91</td>
<td>0.97</td>
<td>0.97</td>
<td>0.93</td>
<td>0.93</td>
<td>0.93</td>
<td>0.93</td>
</tr>
<tr>
<td>v18</td>
<td>0.00</td>
<td>0.82</td>
<td>0.82</td>
<td>0.87</td>
<td>0.87</td>
<td>0.95</td>
<td>0.95</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
<td>0.89</td>
</tr>
<tr>
<td>v19</td>
<td>0.71</td>
<td>0.94</td>
<td>0.90</td>
<td>0.95</td>
<td>0.90</td>
<td>0.96</td>
<td>0.96</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
<td>0.95</td>
</tr>
<tr>
<td>v20</td>
<td>0.00</td>
<td>0.84</td>
<td>0.84</td>
<td>0.85</td>
<td>0.85</td>
<td>0.93</td>
<td>0.93</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
<tr>
<td>v21</td>
<td>0.33</td>
<td>0.69</td>
<td>0.68</td>
<td>0.69</td>
<td>0.68</td>
<td>1.00</td>
<td>0.90</td>
<td>0.73</td>
<td>0.69</td>
<td>0.73</td>
<td>0.69</td>
</tr>
<tr>
<td>v23</td>
<td>0.00</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.97</td>
<td>0.97</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table A.8: Fault location for the Tarantula similarity coefficient using bitmasks.
### Tarantula with Ranges

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.87</td>
<td>0.87</td>
<td>0.87</td>
<td>0.88</td>
<td>0.89</td>
<td>0.90</td>
<td>0.89</td>
<td>0.90</td>
<td>0.91</td>
<td>0.90</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.76</td>
<td>0.71</td>
<td>0.71</td>
<td>0.71</td>
<td>0.78</td>
<td>0.77</td>
<td>0.57</td>
<td>0.70</td>
<td>0.67</td>
<td>0.82</td>
</tr>
<tr>
<td>schedule</td>
<td>0.78</td>
<td>0.72</td>
<td>0.76</td>
<td>0.82</td>
<td>0.82</td>
<td>0.80</td>
<td>0.83</td>
<td>0.81</td>
<td>0.87</td>
<td></td>
</tr>
<tr>
<td>schedule2</td>
<td>0.71</td>
<td>0.72</td>
<td>0.71</td>
<td>0.69</td>
<td>0.73</td>
<td>0.75</td>
<td>0.70</td>
<td>0.69</td>
<td>0.67</td>
<td>0.62</td>
</tr>
<tr>
<td>replace</td>
<td>0.83</td>
<td>0.80</td>
<td>0.78</td>
<td>0.80</td>
<td>0.82</td>
<td>0.82</td>
<td>0.84</td>
<td>0.72</td>
<td>0.67</td>
<td>0.70</td>
</tr>
<tr>
<td>tcas</td>
<td>0.78</td>
<td>0.79</td>
<td>0.78</td>
<td>0.77</td>
<td>0.79</td>
<td>0.78</td>
<td>0.80</td>
<td>0.77</td>
<td>0.73</td>
<td>0.65</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.79</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
<td>0.82</td>
<td>0.83</td>
<td>0.84</td>
<td>0.84</td>
<td>0.83</td>
<td>0.81</td>
</tr>
<tr>
<td>average</td>
<td>0.79</td>
<td>0.77</td>
<td>0.77</td>
<td>0.78</td>
<td>0.81</td>
<td>0.81</td>
<td>0.78</td>
<td>0.78</td>
<td>0.76</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table A.9: Fault location for the Tarantulla similarity coefficient using ranges.

### Jaccard with Bitmasks

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.73</td>
<td>0.41</td>
<td>0.47</td>
<td>0.44</td>
<td>0.47</td>
<td>0.56</td>
<td>0.57</td>
<td>0.57</td>
<td>0.57</td>
<td>0.92</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.78</td>
<td>0.68</td>
<td>0.69</td>
<td>0.55</td>
<td>0.65</td>
<td>0.65</td>
<td>0.75</td>
<td>0.77</td>
<td>0.77</td>
<td>0.72</td>
</tr>
<tr>
<td>schedule</td>
<td>0.75</td>
<td>0.74</td>
<td>0.81</td>
<td>0.87</td>
<td>0.86</td>
<td>0.86</td>
<td>0.83</td>
<td>0.82</td>
<td>0.76</td>
<td>0.77</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.70</td>
<td>0.63</td>
<td>0.65</td>
<td>0.58</td>
<td>0.58</td>
<td>0.58</td>
<td>0.59</td>
<td>0.59</td>
<td>0.58</td>
<td>0.56</td>
</tr>
<tr>
<td>replace</td>
<td>0.76</td>
<td>0.66</td>
<td>0.65</td>
<td>0.66</td>
<td>0.64</td>
<td>0.58</td>
<td>0.51</td>
<td>0.38</td>
<td>0.46</td>
<td>0.55</td>
</tr>
<tr>
<td>tcas</td>
<td>0.79</td>
<td>0.80</td>
<td>0.79</td>
<td>0.78</td>
<td>0.77</td>
<td>0.71</td>
<td>0.67</td>
<td>0.62</td>
<td>0.58</td>
<td>0.56</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.81</td>
<td>0.82</td>
<td>0.83</td>
<td>0.83</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.81</td>
<td>0.78</td>
<td>0.78</td>
</tr>
<tr>
<td>average</td>
<td>0.76</td>
<td>0.68</td>
<td>0.70</td>
<td>0.67</td>
<td>0.68</td>
<td>0.68</td>
<td>0.68</td>
<td>0.65</td>
<td>0.64</td>
<td>0.69</td>
</tr>
</tbody>
</table>

Table A.10: Fault location results for the Jaccard similarity coefficient using bitmasks.

### Jaccard with Ranges

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.74</td>
<td>0.76</td>
<td>0.81</td>
<td>0.83</td>
<td>0.86</td>
<td>0.90</td>
<td>0.90</td>
<td>0.91</td>
<td>0.91</td>
<td>0.90</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.74</td>
<td>0.71</td>
<td>0.71</td>
<td>0.71</td>
<td>0.78</td>
<td>0.77</td>
<td>0.58</td>
<td>0.70</td>
<td>0.67</td>
<td>0.82</td>
</tr>
<tr>
<td>schedule</td>
<td>0.75</td>
<td>0.72</td>
<td>0.75</td>
<td>0.81</td>
<td>0.82</td>
<td>0.81</td>
<td>0.80</td>
<td>0.83</td>
<td>0.81</td>
<td>0.87</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.73</td>
<td>0.71</td>
<td>0.71</td>
<td>0.70</td>
<td>0.73</td>
<td>0.75</td>
<td>0.70</td>
<td>0.69</td>
<td>0.67</td>
<td>0.62</td>
</tr>
<tr>
<td>replace</td>
<td>0.83</td>
<td>0.80</td>
<td>0.78</td>
<td>0.81</td>
<td>0.82</td>
<td>0.82</td>
<td>0.84</td>
<td>0.72</td>
<td>0.67</td>
<td>0.70</td>
</tr>
<tr>
<td>tcas</td>
<td>0.80</td>
<td>0.80</td>
<td>0.78</td>
<td>0.78</td>
<td>0.79</td>
<td>0.78</td>
<td>0.81</td>
<td>0.77</td>
<td>0.73</td>
<td>0.65</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.84</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.83</td>
<td>0.83</td>
<td>0.84</td>
<td>0.85</td>
<td>0.83</td>
<td>0.81</td>
</tr>
<tr>
<td>average</td>
<td>0.78</td>
<td>0.76</td>
<td>0.77</td>
<td>0.78</td>
<td>0.80</td>
<td>0.81</td>
<td>0.78</td>
<td>0.78</td>
<td>0.76</td>
<td>0.77</td>
</tr>
</tbody>
</table>

Table A.11: Fault location for the Jaccard similarity coefficient using ranges.
### Zoltar with Bitmasks

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.59</td>
<td>0.36</td>
<td>0.40</td>
<td>0.45</td>
<td>0.45</td>
<td>0.54</td>
<td>0.54</td>
<td>0.60</td>
<td>0.60</td>
<td>0.95</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.65</td>
<td>0.59</td>
<td>0.60</td>
<td>0.53</td>
<td>0.63</td>
<td>0.63</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.72</td>
</tr>
<tr>
<td>schedule</td>
<td>0.68</td>
<td>0.67</td>
<td>0.72</td>
<td>0.75</td>
<td>0.76</td>
<td>0.78</td>
<td>0.77</td>
<td>0.79</td>
<td>0.76</td>
<td>0.77</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.80</td>
<td>0.71</td>
<td>0.69</td>
<td>0.61</td>
<td>0.59</td>
<td>0.60</td>
<td>0.60</td>
<td>0.59</td>
<td>0.58</td>
<td>0.57</td>
</tr>
<tr>
<td>replace</td>
<td>0.71</td>
<td>0.62</td>
<td>0.62</td>
<td>0.58</td>
<td>0.53</td>
<td>0.47</td>
<td>0.36</td>
<td>0.43</td>
<td>0.56</td>
<td></td>
</tr>
<tr>
<td>tcas</td>
<td>0.72</td>
<td>0.72</td>
<td>0.74</td>
<td>0.74</td>
<td>0.74</td>
<td>0.67</td>
<td>0.66</td>
<td>0.59</td>
<td>0.55</td>
<td>0.56</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.81</td>
<td>0.85</td>
<td>0.85</td>
<td>0.82</td>
<td>0.80</td>
<td>0.81</td>
<td>0.82</td>
<td>0.80</td>
<td>0.80</td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>0.71</td>
<td>0.65</td>
<td>0.66</td>
<td>0.65</td>
<td>0.65</td>
<td>0.66</td>
<td>0.65</td>
<td>0.64</td>
<td>0.64</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Table A.12: Fault location for the Zoltar similarity coefficient using bitmasks.

### Zoltar with Ranges

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.64</td>
<td>0.65</td>
<td>0.69</td>
<td>0.71</td>
<td>0.71</td>
<td>0.73</td>
<td>0.72</td>
<td>0.79</td>
<td>0.85</td>
<td>0.95</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.64</td>
<td>0.65</td>
<td>0.64</td>
<td>0.66</td>
<td>0.67</td>
<td>0.73</td>
<td>0.57</td>
<td>0.66</td>
<td>0.67</td>
<td>0.85</td>
</tr>
<tr>
<td>schedule</td>
<td>0.62</td>
<td>0.62</td>
<td>0.65</td>
<td>0.67</td>
<td>0.66</td>
<td>0.68</td>
<td>0.66</td>
<td>0.74</td>
<td>0.74</td>
<td>0.87</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.79</td>
<td>0.79</td>
<td>0.80</td>
<td>0.79</td>
<td>0.80</td>
<td>0.82</td>
<td>0.73</td>
<td>0.73</td>
<td>0.70</td>
<td>0.64</td>
</tr>
<tr>
<td>replace</td>
<td>0.73</td>
<td>0.73</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.74</td>
<td>0.66</td>
<td>0.63</td>
<td>0.72</td>
</tr>
<tr>
<td>tcas</td>
<td>0.74</td>
<td>0.74</td>
<td>0.74</td>
<td>0.74</td>
<td>0.75</td>
<td>0.77</td>
<td>0.78</td>
<td>0.78</td>
<td>0.73</td>
<td>0.65</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.84</td>
<td>0.84</td>
<td>0.85</td>
<td>0.84</td>
<td>0.85</td>
<td>0.87</td>
<td>0.88</td>
<td>0.86</td>
<td>0.86</td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>0.72</td>
<td>0.72</td>
<td>0.73</td>
<td>0.73</td>
<td>0.74</td>
<td>0.76</td>
<td>0.73</td>
<td>0.75</td>
<td>0.74</td>
<td>0.79</td>
</tr>
</tbody>
</table>

Table A.13: Fault location for the Zoltar similarity coefficient using ranges.

### Ochiai with Bitmasks

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.61</td>
<td>0.41</td>
<td>0.45</td>
<td>0.48</td>
<td>0.49</td>
<td>0.57</td>
<td>0.57</td>
<td>0.58</td>
<td>0.58</td>
<td>0.91</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.78</td>
<td>0.69</td>
<td>0.70</td>
<td>0.56</td>
<td>0.66</td>
<td>0.66</td>
<td>0.76</td>
<td>0.77</td>
<td>0.77</td>
<td>0.72</td>
</tr>
<tr>
<td>schedule</td>
<td>0.71</td>
<td>0.69</td>
<td>0.72</td>
<td>0.79</td>
<td>0.78</td>
<td>0.78</td>
<td>0.80</td>
<td>0.79</td>
<td>0.76</td>
<td>0.77</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.75</td>
<td>0.68</td>
<td>0.67</td>
<td>0.59</td>
<td>0.59</td>
<td>0.60</td>
<td>0.60</td>
<td>0.59</td>
<td>0.58</td>
<td>0.56</td>
</tr>
<tr>
<td>replace</td>
<td>0.75</td>
<td>0.66</td>
<td>0.65</td>
<td>0.65</td>
<td>0.62</td>
<td>0.56</td>
<td>0.49</td>
<td>0.38</td>
<td>0.45</td>
<td>0.56</td>
</tr>
<tr>
<td>tcas</td>
<td>0.77</td>
<td>0.77</td>
<td>0.77</td>
<td>0.75</td>
<td>0.75</td>
<td>0.69</td>
<td>0.65</td>
<td>0.59</td>
<td>0.55</td>
<td>0.56</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.84</td>
<td>0.85</td>
<td>0.86</td>
<td>0.85</td>
<td>0.83</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.79</td>
<td></td>
</tr>
<tr>
<td>average</td>
<td>0.75</td>
<td>0.68</td>
<td>0.69</td>
<td>0.67</td>
<td>0.67</td>
<td>0.67</td>
<td>0.67</td>
<td>0.65</td>
<td>0.64</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Table A.14: Fault location for the Ochiai similarity coefficient using bitmasks.
### Ochiai with Ranges

<table>
<thead>
<tr>
<th>Version/Training</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>0.74</td>
<td>0.78</td>
<td>0.82</td>
<td>0.85</td>
<td>0.84</td>
<td>0.92</td>
<td>0.91</td>
<td>0.92</td>
<td>0.93</td>
<td>0.92</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.72</td>
<td>0.73</td>
<td>0.79</td>
<td>0.59</td>
<td>0.68</td>
<td>0.68</td>
<td>0.83</td>
</tr>
<tr>
<td>schedule</td>
<td>0.66</td>
<td>0.64</td>
<td>0.67</td>
<td>0.70</td>
<td>0.71</td>
<td>0.73</td>
<td>0.75</td>
<td>0.81</td>
<td>0.81</td>
<td>0.87</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.80</td>
<td>0.77</td>
<td>0.77</td>
<td>0.76</td>
<td>0.78</td>
<td>0.80</td>
<td>0.71</td>
<td>0.72</td>
<td>0.69</td>
<td>0.64</td>
</tr>
<tr>
<td>replace</td>
<td>0.78</td>
<td>0.77</td>
<td>0.74</td>
<td>0.75</td>
<td>0.76</td>
<td>0.75</td>
<td>0.78</td>
<td>0.69</td>
<td>0.65</td>
<td>0.71</td>
</tr>
<tr>
<td>tcas</td>
<td>0.79</td>
<td>0.75</td>
<td>0.74</td>
<td>0.77</td>
<td>0.79</td>
<td>0.79</td>
<td>0.80</td>
<td>0.78</td>
<td>0.74</td>
<td>0.65</td>
</tr>
<tr>
<td>tot_info</td>
<td>0.86</td>
<td>0.86</td>
<td>0.86</td>
<td>0.86</td>
<td>0.87</td>
<td>0.88</td>
<td>0.88</td>
<td>0.88</td>
<td>0.86</td>
<td>0.84</td>
</tr>
<tr>
<td>average</td>
<td>0.77</td>
<td>0.76</td>
<td>0.76</td>
<td>0.77</td>
<td>0.78</td>
<td>0.81</td>
<td>0.77</td>
<td>0.78</td>
<td>0.77</td>
<td>0.78</td>
</tr>
</tbody>
</table>

Table A.15: Fault location for the Ochiai similarity coefficient using ranges.