Master Thesis Software Engineering

Empirical Evaluation of Automatic Fault Localization based on Program Invariants

Lars de Ridder

Supervisor: Jan van Eijck

Publication status: Under review

Universiteit van Amsterdam

Centrum Wiskunde & Informatica

August 20, 2009

Abstract

Whenever a failure occurs in a program, the program has to be debugged. A large part of debugging consists of localizing the defect in the code. This can be very difficult and time consuming, due to failures being observed much later than their associated defects have been introduced, or because of a very large code base.

Techniques that automate the localization of a defect are called fault localization techniques. This thesis concentrates on evaluating an automatic fault localization technique based on invariants. This technique works by using invariants to build a model of what "normal" state and behaviour is. Violations of these invariants are considered abnormalities and should point in the direction of the defect.

We have implemented the technique in a prototype tool called Kitsune. This prototype can formulate a number of invariants for C programs and report violations of these invariants. We have evaluated the tool using the evaluation framework of Renieris and Reiss based on the Siemens suite of programs. The results have been compared with the results reported by other fault localization techniques.

We found that Kitsune performed better than expected according to suggestions that program invariants aren't suitable for fault localization [14]. Using program invariants for fault localization certainly has potential, and an idea to significantly improve the results is presented as a combination of our system with Tarantula's theory. Some other interesting ideas for future work in this area are also proposed.

Contents

Chapter 1

Introduction

Whenever a program doesn't do what it is supposed to do and this faulty behaviour is observable, a failure has occurred. A failure is caused by a defect[1] in the code of the program, which has to be corrected to make the program behave correctly again. The process of locating a defect and correcting it is called debugging.

The process of localizing the defect is usually the most time consuming part of software debugging, and thus in turn a large part of software development. This is because defects can result in failures much later than when they were introduced in the code, making it unclear which change caused it. In these situations, the debugger is often also confronted with an enormous code base, making it even harder.

Utilizing computing power to automate part of the debugging process would be very rewarding. Computing power has increased drastically over the years but is hardly used to increase the reliability of software. Automating part of the debugging process, which consists of a lot of tedious work, would be a large step in the right direction to increase software reliability.

Techniques that automate the localization of a defect are called fault localization techniques. These techniques aim at providing a (human) user with information to help find the location of a defect in the code. Possibly, this information can also help in correcting the defect. A large number of fault localization techniques have been proposed, including static methods that analyze the source code and dynamic methods that look at coverage information or other information obtained from executions of a program.

This thesis focuses on automatic fault localization using program invariants. The use of program invariants is inspired by Hangal and Lam, who reported positive results with their DIDUCE implementation for Java programs [8]. The theory behind this technique is that program invariants can capture what "normal" state and/or behaviour is at certain points in the program. Using this information, it is possible to determine if and when a program shows abnormal state and/or behaviour. Information about these abnormalities is then thought to point in the direction of the cause of a failure, and thus to the defect.

The goal of this research is to evaluate the effectiveness of automatic fault localization based on program invariants. Positive results using invariants have been reported by Hangal and Lam, as mentioned above, but Pytlik et al. reported negative results, and consequently suggested that invariants might not be suitable for fault localization at all [14]. This research will hopefully give a definitive answer to the question of the potential of invariants for finding faults.

To perform this evaluation we have implemented the technique in a suite of programs called Kitsune[2]. We have used the evaluation framework designed by Renieris and Reiss [16] and the Siemens test suite of C programs [9] to compare our technique to other fault localization techniques that used the same evaluation framework and programs.

[1] We will use the terms fault and defect interchangeably in this paper.

[2] Kitsune is Japanese for fox, often used to refer to foxes in Japanese folklore.

The rest of this thesis is structured as follows. In the following chapter, we will discuss some related work. After that, we will introduce our notion of program invariants, the actual invariants used and the program points instrumented. Then we will discuss our research method, detailing our implementation and providing some more information on the programs used for evaluation. We then present the results of our experiments, after which we finalize with a discussion of the results and some interesting ideas for future work.

Chapter 2

Related work

Multiple techniques have been proposed for automatic fault localization. These techniques can be subdivided in static and dynamic techniques. Static techniques exclusively use the source code of a program to localize defects, while dynamic techniques use information from actual executions of the program as well. As the technique discussed in this paper is dynamic, we will focus on dynamic techniques in this chapter.

This thesis was inspired by the DIDUCE system, presented by Hangal and Lam [8]. This system formulates hypotheses of invariants during the execution of a program with the purpose of localizing faults by detecting abnormalities. They have evaluated their technique on a number of Java programs. They reported positive results, but further analysis by Abreu et al. revealed that the faults in the programs that were used for evaluation demonstrated behaviour that is very easy to detect using their invariants [1].

Invariants have also been used by Pytlik et al. to detect faults [14]. They implemented a prototype called Carrot which used the notion of invariants developed by Ernst et al. [6], which in turn is implemented in the Daikon system. Unfortunately, their results turned out to be negative, and they even suggested that program invariants might not be suitable for fault localization at all.

Abreu et al. have used program invariants to flag runs as correct or faulty, after which a spectrum-based fault localization technique was used to find the actual location of the fault [1]. They found that the accuracy of program invariants to correctly flag runs as correct or faulty is high, comparable with knowing beforehand which runs will pass and which will fail.

The above techniques emphasized invariants in their research. However, other fault localization techniques have been proposed that don't use invariants at all. A number of those techniques change a failing run to make it behave correctly. Zhang et al. [21] and Wang et al. [19] forcefully change the outcomes of predicates in a failing run to turn it into a passing run. Jeffrey et al. alter values in a failing run for the same purpose [10].

Various statistical methods have been proposed as well. Renieris and Reiss [16] try to find a passing run that looks the most like a failing run by using several metrics. They then compare the statements executed in both runs to find suspicious statements. Jones and Harrold [11] developed the Tarantula system, which works by assigning a suspiciousness score to each statement. This score is higher for statements that are executed more often in failing test runs and less often in passing test runs. Statements with the highest suspiciousness score have the highest chance of being faulty.

Other techniques to localize faults include applications of the Delta Debugging technique by Zeller et al., that find state differences between passing and failing runs [20][4]. There exist also a number of techniques that use coverage information to build sets of statements that correspond to passing and failing runs, which are then used to find suspicious statements by performing various set operations on them (such as unions and intersections) [5][7].

Chapter 3

Program invariants

The purpose of program invariants in this research is to to track state and/or behaviour at a number of program points. We first "calculate"[1] these invariants based on values encountered in all passing runs. Then we look at values encountered in non-passing runs (i.e. failing runs) and determine whether the encountered values conform to the invariants or not. If they do not, the program invariants are violated and we expect these violations to point to the location of the defect in the code.

In this chapter we will first introduce program invariants and give a definition for them. After that, we will discuss the program points where the program invariants will be calculated. We will then detail the invariants used for our fault localization system, a section that ends with an example to illustrate the working of the invariants.

3.1 Definitions

Program invariants can be defined as conditions that have to be met by the state of the program for it to be correct [1]. This definition is a bit rigorous for our purposes, as this would mean no false positives would be allowed. We prefer not to encounter false positives either, but we are willing to tolerate them to prevent over-learning. Thus we relax this definition to the following:

> *A program invariant is a condition that is always met at a certain program point during all passing runs of a program*

This definition also requires definitions for "condition" and "passing run". A condition is a predicate that is formed based on values encountered at the associated program point during executions of the program. A passing run is an execution (or run) of a program whose output is equal to a certain predefined output. This in turn introduces questions about the correctness of the predefined output, but we will assume these to be correct.

Our notion of program invariants is not a very common one. When invariants are mentioned, one usually thinks of for example loop invariants or class invariants. To clarify our notion of program invariants, it helps to make a comparison between our program invariants and the more common loop invariants.

Our notion has two major differences with loop invariants. The first is that a loop invariant is associated with a certain loop, and is true everywhere in and during every execution of that loop. In the case of our program invariants, each program invariant is associated to a certain point in the program. The invariant is true during every execution of the program, but only *at that associated point*. This means that every invariant is only true at a certain execution state of the program, but whenever an execution of the program reaches this execution state it will be true.

[1] We will explain what we mean with this in the next section.

The second major difference is that loop invariants are valid during various executions of loops, while our program invariants are valid during various executions of an *entire program*. However, one could view the whole program as a loop that is executed multiple times (with different input parameters each time). From that perspective, program invariants and loop invariants are very much alike, although the first major difference still applies of course.

In this thesis, we will mention invariants being "calculated" multiple times, sometimes in relation to a program point. With this we mean that the values encountered at a certain program point are used to "calculate" (or formulate, determine) what the predicate will be against which other values can be tested. For example, in the case of the range invariant discussed later, values from failing runs can be tested against a predicate formed by the computed ranges. We call this process "calculating" invariants, for lack of a better word.

3.2 Program points

A program point is simply what the name implies; a point in the program. In the context of this thesis, program points are the locations in the code where program invariants will be calculated. In this section, we will discuss which program points we selected for this.

The choice of where program invariants will be calculated has been heavily influenced by the framework used to instrument the test programs to gather data, which is the LLVM Compiler Infrastructure [12]. More details of this will be discussed later in the chapter detailing the research method. For now it suffices to say that LLVM compiles a program into its own intermediate bitcode representation, which can then be manipulated by using a library provided by LLVM.

The bitcode representation of LLVM works at a fairly low level of abstraction. Four of the main operations that deal with variables have been used as program points where the instrumentation will take place, and thus where program invariants are calculated. These operations are the following.

- LOAD - Loads a value from memory.
- STORE - Stores a value to memory.
- CALL - Calls a function (possibly with arguments) and optionally returns a value.
- CMP - Compares two values.

To illustrate, consider the following statement:

```
x = y + 2
```

In this statement, the value of the variable y is loaded from memory (LOAD operation in LLVM), the value 2 is added to this loaded value, and the new value is stored to memory as variable x (STORE operation in LLVM). This means that in this statement, program invariants are calculated over all loaded values of y as well as all stored values of x. More details about where instrumentation takes place (and thus where invariants are calculated) can be found in the example subsection of the chapter detailing research method.

The LOAD and STORE operations are the two most essential operations in the LLVM language, being used everywhere where values are involved (this makes intuïtive sense; a value has to be loaded from memory to be used, and stored for it to be used later). As a result, selecting these as program points to calculate our invariants ensures that we will calculate invariants on all points where value based (or data flow) abnormalities could occur. The two other operations (CALL and CMP) have been added later as potentially interesting points for a user as control flow abnormalities are thought to occur here.

This selection has some drawbacks though. As the LOAD and STORE operations occur very frequently, a very large number of invariant violations can be reported to the user in some cases, possibly with

duplicates. This also has a negative impact on the execution time, as a large number of invariants have to be calculated in any given program due to the frequent occurrence of the LOAD and STORE operations. The first drawback can largely be negated by implementing some intelligence in composing the fault reports, for example by filtering out duplicates. The second drawback is outside the scope of this research but might require attention in the future. Both drawbacks are mainly usability issues however and have little to no impact on the effectiveness of our technique.

Not all invariants will be calculated at each type of program point. For each invariant discussed in the next subsection we will mention at which program points and for which data types they will be calculated.

3.3 Invariants

Six types of program invariants have been designed for our fault localization system. Each type of invariant has a number of variable components, allowing it to be changed in a number of ways. For example, the range invariant (detailed later) has a variable number of ranges that can be used, allowing the designer some freedom in their implementation. For simplicity, we will continue referring to these types of program invariants as invariants or program invariants.

We divide the designed invariants in two groups: generic invariants and situational invariants. Generic invariants are invariants that are designed to localize most types of faults and are thus the main force behind our fault localization system. However, generic invariants are unable to localize some types of faults. This is either because these faults occur at variables with a relatively unusual type (such as pointers), or because the type of fault is very domain specific.

For this purpose, we introduce situational invariants, designed to localize a small number of specific types of faults. They are not expected to find a lot of faults, but when the types of faults for which they are designed exist in the program under analysis, they are supposed to perform very well.

We will first discuss the three generic invariants after which the three situational invariants will be detailed. At the end of this section, the workings of the invariants will be illustrated by a code example.

3.3.1 Generic invariants

This section will discuss the generic invariants designed for our prototype. These are the following:

- Bitmask invariant
- Sign-and-magnitude bitmask invariant
- Range invariant

Bitmask invariant This invariant was used by Hangal et al. in their fault localization system DIDUCE [8]. It is composed of two fields: the first observed value ($first$) and a bitmask ($bmsk$). A bitmask is a sequence of ones and zero's, where a zero represents an "activated" bit. Initially, all bits in the bitmask are set to 1.

Note that this invariant always considers the binary representation of values, not the actual decimal values. Every decimal value that is encountered is first transformed into its binary representation, which is then used in the invariant. In this section, for simplicity, we use 8 bits to represent the binary values. In reality, this can be any number of bits (for example 32 for integer values).

When a new value (new) is added to the bitmask, we determine whether or not a violation has occurred using the following equation:

$$violationOccurred = ((new \oplus first) \wedge bmsk) \neq 0 \tag{3.1}$$

Checked	First	Bitmask	New	Difference
0	-	-	00001010 (10)	-
1	00001010	11111111	00001000 (8)	00000010
2	00001010	11111101	01001010 (74)	01000000
3	00001010	10111101	00111000 (56)	00110000
4	00001010	10001101	00010010 (18)	00010000
5	00001010	10001101	00001011 (11)	00000001

Table 3.1: Example life cycle of the bitmask invariant. In this example, if we imagine the first five values (check 0 to 4) to be from passing runs, the last value (in check 5) would violate the invariant and make the program report a violation to the user.

where $\oplus$ and $\wedge$ are the bitwise *xor* and *and*, respectively. If the outcome of this equation is false, the bitmask will accept the value and remain unchanged.

If, however, the outcome is true, a violation is encountered. If the value belongs to a failing run, a violation is reported to the user for this value. If it belongs to a passing run, the bitmask will be relaxed according to the following equation:

$$bmsk = \neg(new \oplus first) \wedge bmsk \tag{3.2}$$

Table 3.1 shows the life cycle of an example bitmask invariant. In this table (and all other bitmask life cycle tables in this paper), the column "difference" is the result of $new \oplus first$. Whenever the bit pattern in this column has a 1 on the same location where the bitmask has a 1, equation 3.1 will be true and thus a violation occurs. Following this, we see that in check 4 in this table a value is added for which equation 3.1 returns false, causing the bitmask to be unchanged in check 5 relative to check 4. In all other cases, equation 3.1 returns true and thus the bitmask is relaxed following equation 3.2.

The bitmask invariant is used to track the normal state space of values encountered at a certain point in the program. It learns relatively quickly and can track a number of interesting properties, such as values that are always even or uneven (the rightmost bit of the bitmask will always be 1) and values that are always positive or negative (the leftmost bit of the bitmask will be 1).

Another advantage of this invariant is the existence of a useful confidence formula, to help prioritize users between reported violations. This is of limited use to us though as the other invariants used don't have anything comparable, which means we can only use it to prioritize between bitmask invariants. For completeness, we will explain it here briefly.

This confidence level can be calculated as follows. Let k be the number of times an expression has been evaluated and let 2^n be the number of values the invariants accepts, with n being the number of zero's in the binary representation of the invariant. The confidence is then calculated with the following formula:

$$confidence = \frac{k}{2^n} \tag{3.3}$$

This invariant has some shortcomings however. First of all, it has limited support for floating point values. We do think it may be possible to design a bitmask invariant that can deal with floating point values, however we haven't done this due to the complexity of the binary representation of floating point values and the limited time we have available for this research. Another shortcoming is that the bitmask invariant has limited capability for showing ranges due to its upper bound being far from tight. Finally, it has limited support for variables that can take on positive as well as negative values.

Most other papers only mention this last shortcoming without much elaboration. We will describe it in more detail, as it will be relevant for our discussion of the next invariant. To do this, however, we will first have to explain the binary number representation used for negative numbers for this bitmask invariant, which is the standard two's complement representation.

The two's complement representation is as follows. To obtain the binary representation for a certain negative number, let's say -9, consider the binary representation of its positive version (9), which is

00001001. To then get the binary representation for a negative number, we have to invert this bit pattern (that is, turn all zero's to ones and vice versa) and then add 1 to this result. Following this process, inverting the bit pattern of 9 gives us 11110110, after which we get the correct two's complement binary representation for -9 by adding 1 to this, resulting in 11110111. This means that the value -1, the closest negative integer to zero, has as binary representation 11111111, which is the result of inverting and adding one to 00000001, the binary representation of 1. Another notable value in this case is -128, the furthest negative integer from zero that can be represented in 8 bits, which has as binary representation 10000000.

Now, the limited support for negative values can be encountered in two situations. For both of these situations, we will first show an example where the situation is encountered, after which we will give a short explanation as to why and when this happens.

The first situation is illustrated in table 3.2. What we see here is a random positive value (42) being encountered together with a random negative value (-128), which violates the bitmask according to equation 3.1 and 3.2. (as you would expect). After that we encounter another random negative value (-86), however in this case the bitmask invariant is not violated. At first sight, it seems as if there's no relation between this value and the previous two values, making it unexpected that no violation is encountered. However, at a closer look, we see that 42 and -86 have almost the same binary representation; the only difference is the leftmost bit.

Checked	First	Bitmask	New	Difference
0	-	-	00101010 (42)	-
1	00101010	11111111	10000000 (-128)	10101010
2	00101010	01010101	10101010 (-86)	10000000

Table 3.2: Example life cycle of a bitmask invariant, illustrating its first shortcoming when encountering positive values together with negative values.

The above happens because of the nature of the two's complement binary representation. Due to the inverting of the bit pattern, there are numbers that have almost the same binary representation while having no obvious relation with each other in the decimal system, such as the 42 and -86 in the above example. To be precise, if we have a positive value $posVal$ and we consider b to be the number of bits used for the binary representation of values (8 in the examples, 32 for integers), the negative value $-2^{b-1} + posVal$ has the same binary representation as $posVal$, but with a 1 as leftmost bit. As a result, this negative value will be accepted by the bitmask invariant whenever $posVal$ and any negative value are encountered by the invariant.

For the second situation, consider table 3.3. What we see here is that our bitmask invariant encounters a very small positive value (0) and a negative value that is very near to zero (-1). After encountering these values, we see that the bitmask becomes all zero's. When the bitmask is all zero's it means that it will accept every value (including the 42 seen in the example), as equation 3.1 will never be true in that case. So even though two values are encountered that are very near to each other in the decimal system, they are seen as completely different in the binary system, and thus by the bitmask invariant.

Checked	First	Bitmask	New	Difference
0	-	-	00000000 (0)	-
1	00000000	11111111	11111111 (-1)	11111111
2	00000000	00000000	00101010 (42)	00101010

Table 3.3: Example life cycle of a bitmask invariant, illustrating its second shortcoming when encountering positive values together with negative values.

This effect is also caused by the inverting process of the two's complement binary representation. This situation occurs every time a variable can take on values that have a large amount of ones together with values that have a large amount of zero's in their bitmask representation, so either very large positive

numbers together with negative values that are very far from zero (such as 127 and −128), or small positive numbers together with negative numbers that are near to zero (such as 0 and −1, as in our example).

The bitmask invariant is calculated for every integer value at all LOAD, STORE and CALL program points. In case of the CALL operation, its arguments and return value are tracked.

Sign-and-magnitude bitmask invariant One of the drawbacks of the bitmask invariant is the limited support for negative numbers when both positive and negative numbers are encountered at a program point, as detailed above. As it is not uncommon to encounter positive as well as negative numbers on the same program point, this was a bigger drawback than we wanted to accept for this invariant. So, after recalling old lessons about binary number representations, we came up with the sign-and-magnitude bitmask invariant.

This invariant is in all ways the same as the normal bitmask invariant, except when dealing with negative numbers. When a negative value is added to the invariant (following equations 3.1 and 3.2), it doesn't transform the value to the standard two's complement binary representation as the normal bitmask invariant does, but the sign-and-magnitude binary representation.

The sign-and-magnitude binary representation states that the leftmost bit is used as the sign (equivalent to the − used for decimal numbers), while the rest of the bits signify the magnitude of the value. To obtain the binary representation of a negative value, you take the bit pattern of the positive version of the value and change the leftmost bit from a 0 to a 1. So for 8-bit values, the binary representation of the value −1 would be 10000001 and that of the value −127 would be 11111111, as opposed to 11111111 and 10000001 respectively in the two's complement representation.

The advantage of this representation is that whenever negative and positive values are near to each other or look like each other in the decimal system, this will be preserved in their binary representation. For example, the values 0 and −1 have 00000000 and 10000001 as their respective binary representations now. These representations are very much alike, representing the fact that these two values are near to each other in the decimal system.

However, the first problematic situation of the normal bitmask invariant still applies, although it makes more sense now. Consider table 3.4. Here we see that a positive value is encountered (42), together with some negative value (−1) and the negative version of the earlier encountered positive value (−42). This last value is accepted by the sign-and-magnitude bitmask invariant, even though this might not be expected behaviour. However, it is easier to see why this happens, as their similarity in the decimal system is now reflected in their binary representations.

Checked	First	Bitmask	New	Difference
0	-	-	00101010 (42)	-
1	00101010	11111111	10000001 (-1)	10101011
2	00101010	01010100	10101010 (-42)	10000000

Table 3.4: Example life cycle of a bitmask invariant, illustrating its first shortcoming when encountering positive values together with negative values.

We can't really theoretically determine which one of the bitmask invariants is superior, as there are an equal amount of situations where one performs better than the other. We think that situations where a variable takes on small positive numbers as well as negative numbers near to zero are among the most likely situations when both positive and negative numbers are encountered. If this is true, the sign-and-magnitude bitmask invariant would be superior. However, as there is no way to say whether this is true or not, our experiments have to show which one performs better.

The sign-and-magnitude invariant is calculated for every integer value at all LOAD, STORE and CALL program points. In case of the CALL operation, its arguments and return value are tracked.

Range invariant To compensate for the shortcomings of the bitmask invariants, we use the range invariant as well, introduced by Racunas et al. [15]. The working of this invariant is pretty straightforward. It reads all encountered values in passing runs and then partitions these values in a number of ranges (or clusters) of values. We have implemented the data clustering algorithm K-means [13] for this purpose, which we run after we collected all data. Gathering data first and processing it later is called offline sampling. In a fault localization tool using online sampling, which processes data the moment it is gathered, the ranges would dynamically be determined at runtime.

We have used three ranges (or clusters) in the calculation of our range invariant. We have tried calculating the number of ranges dynamically, but as the data we can encounter is completely arbitrary, we see no way of determining which approach of doing this would be the best. Our opinion is that two ranges should be the minimum to allow for variables that take on a range of normal values and one invalid but accepted value, for example as initialization. Also, we thought we shouldn't use too many ranges, as this would result in a large amount of false positives. This made us decide to use three ranges.

This invariant doesn't have the disadvantages that the bitmask invariant has, but it doesn't show some of the extra interesting properties that the bitmask invariant can show either (for example even/uneven). Also, it learns a lot slower which should result in a lot more violations. This makes it valuable to use both invariants.

The range invariant is calculated for both integer as well as floating point values at all LOAD, STORE and CALL program points. In case of the CALL operation, its arguments and return value are tracked.

3.3.2 Situational invariants

This section will discuss the situational invariants designed for our prototype. These are the following:

- Compare invariant
- Null pointer invariant
- Not executed invariant

Compare invariant This invariant is inspired by the predicate switching techniques [19][21]. These techniques localize faults by forcefully changing the outcomes of predicates and thus altering the behaviour of the program.

We will be less rigorous and simply track the outcomes of every comparison of values in the code. The invariant will track if the outcome of a comparison is always true or always false in passing runs, or whether it can be both. If it turns out it can be both, we conclude that both outcomes can make the program behave correctly, and thus that neither outcome violates the invariant. In other cases it might prove to be an interesting indicator of abnormal behaviour.

The compare invariant is calculated for all COMPARE program points.

Null pointer invariant This invariant is very similar to the compare invariant. This invariant tracks whether a pointer is always null or never null in passing runs. Again, neither outcome violates the invariant if it can be both.

The null pointer invariant is calculated for pointer values at all LOAD, STORE and CALL program points. In case of the CALL operation, its arguments and return value are tracked.

Not executed invariant This is a special type of invariant in that it does not depend on values from passing runs. This invariant is violated when a program point is reached in a failing run that is never reached in any passing run. This means that this invariant can indicate parts of the program that are only executed in failing runs.

```
1  read (int x, int y) {
2    int z, a;
3    a = x + y; /* should be x - y */
4
5    if (x < y) {
6      z = a;
7    } else {
8      z = a + 1;
9    }
10   return z;
11 }
```

Listing 3.1: Code snippet with a fault on line 3.

Case	Input	Expected output	Actual output	Result
A	(x,y) = (0,0)	1	1	PASS
B	(x,y) = (1,0)	2	2	PASS
C	(x,y) = (-1,0)	-1	-1	PASS
D	(x,y) = (1,1)	1	3	FAIL
E	(x,y) = (0,-1)	2	0	FAIL

Table 3.5: Test suite for the faulty function *read* in code listing 3.1.

The not executed invariant is calculated for all types of values at all above mentioned program points.

3.3.3 Example

To further illustrate the working of the invariants, we will use a small code example. Note that the invariants used here are, whenever needed, simplified to easier illustrate the concepts.

The faulty function *read* shown in listing 3.1 (obtained from [10]) will be used in this example. The defect is located on line 3, where the addition operator (+) should be a subtraction operator (−). Table 3.5 shows a test suite for this function. The error manifests itself when the variable y takes on a non-zero value, so we would expect that a fault report points to this variable.

For simplicity, the range invariant will track only one range in this example, and the bitmask invariants will assume the values are 4 bit values. We will focus on invariants calculated for the variables on line 3, where the defect is located. Note that invariants are also formulated for various other program points, but we won't consider these now. To see where other invariants are calculated, see the example section at the end of the next chapter.

For the range invariant, the process goes as follows. First, we compute the range of valid values for each variable by looking at all values encountered for that variable in passing runs. Following table 3.5, for x this would be -1 to 1, for y we get a range of 0 to 0, and for a it would be -1 to 1. Then, for each variable, the values recorded during failing runs are checked against its calculated range.

In table 3.5, we see that value 1 and 0 are encountered for x in the two failing runs. These values fall within the valid range of values calculated for x, so nothing is done. However, for y, the values 1 and -1 are encountered, both falling outside the valid range. Also, variable a takes on the value 2 in case D of table 3.5, which falls outside its valid range as well. The result is that two violations are reported to the user. One of these will point at variable y, reporting the values 1 and -1, and one will point at a, where the value 2 is reported.

For the bitmask invariant and the sign-and-magnitude bitmask invariant, we will focus on the values of variable a on line 3, as these are the most interesting in this case. The life cycle for this variable of both bitmask invariants are shown in table 3.6. Here the shortcoming of the normal bitmask invariant encountering positive as well as negative values is highlighted, as encountering the value -1 in check 2 sets the bitmask to all zero's, resulting in it not detecting the violation in check 3 where the value 2 is

encountered and thus reporting nothing to the user.

Checked	First	Bitmask	New	Difference	SaM bitmask	SaM new	SaM difference
0	-	-	0000 (0)	-	-	0000 (0)	-
1	0000	1111	0001 (1)	0001	1111	0001 (1)	0001
2	0000	1110	1111 (-1)	1111	1110	1001 (-1)	1001
3	0000	0000	0010 (2)	0010	0110	0010 (2)	0010

Table 3.6: Life cycle of both bitmask invariants for the test suite in table 3.5 for the code in listing 3.1. The variable tracked is a on line 3. Check 3 encounters the value from the failing run (2). The sign-and-magnitude bitmask encounters a violation in this check, but the normal bitmask does not.

The sign-and-magnitude bitmask invariant does detect the violation, as encountering -1 has less of an influence on the bitmask. This violation is then reported to the user. This report contains the first encountered value (in this case 0), its bitmask representation (0000) the bitmask at the time of the violation (0110), the decimal representation of the faulty value (2), the binary representation of the encountered faulty value (0010) and the difference between the bitmask and the binary representation of the faulty value (0010).

The situational invariants do not play a significant role in this example. As no pointers are used in the faulty function, the null pointer invariant of course doesn't do anything. Also, the passing tests already result in full code coverage, so the not executed invariant is not used either. Finally, the if-statement on line 5 evaluates to true in case C and to false in case A and B. The compare invariant will thus state that both true as well as false is an accepted result of this comparison, which makes it not report a violation regardless of how it is evaluated in failing runs.

In this example, the range invariant and both bitmask invariants point directly to the faulty statement on line 3, all three reporting variable y[2] as taking on abnormal values (note that these invariants would also point at y on line 5). In addition, the range invariant and the sign-and-magnitude bitmask invariant will indicate variable a on line 3 to be abnormal. Figure 4.2 in the example at the end of the section on the implementation in the next chapter will show how these violations are reported to a user.

[2]We didn't discuss y in our discussion of the bitmask invariants, but as the only valid value is 0, it should be fairly easy to see that any other value would trigger a violation for these invariants.

Chapter 4

Research method

In the previous chapter, we have introduced our notion of invariants together with which invariants are to be used in our fault localization system. Now is a good time to step back and remember our goal: we want to evaluate automatic fault localization based on program invariants. To realize this goal, we have implemented a prototype called Kitsune and we have evaluated this prototype. This chapter will discuss how these steps were performed.

We will first discuss the set of programs we have used for our evaluation. Then, we will describe how we implemented the various part of our fault localization system, together with an example to illustrate its working in practice. Finally, we will detail how we have evaluated the effectiveness of our system.

4.1 Benchmark set

We used a set of programs called the *Siemens set* [9], as maintained by the Galileo Software-artifact Infrastructure Repository (SIR) [17]. This set consists of seven C programs, that each have one correct version and a number of faulty versions, totalling up to 132 faulty versions. Each faulty version has exactly one defect, but each defect may span multiple statements or functions. Each program has a set of inputs that ensures full code coverage. Table 4.1 provides some more information about the programs in the set.

The Siemens set is used because it is commonly used by other fault localization techniques. It also forms the basis of the evaluation framework proposed by Renieris and Reiss to compare fault localization techniques with each other [16]. The set is not assembled with evaluating fault localization techniques as its purpose, however.

We haven't used all 132 faulty versions in our evaluation. A number of faulty versions caused segmentation faults, which in general is not a problem as our implementation logs the values at runtime, and as

Program	Faulty Versions	LOC	Test Cases	Description
print_tokens	7	539	4130	Lexical analyser
print_tokens2	10	489	4115	Lexical analyser
schedule	9	397	2650	Priority scheduler
schedule2	10	299	2710	Priority scheduler
replace	32	507	5542	Pattern recognition
tcas	41	174	1608	Altitude separation
tot_info	23	398	1052	Information measure

Table 4.1: The Siemens set of programs.

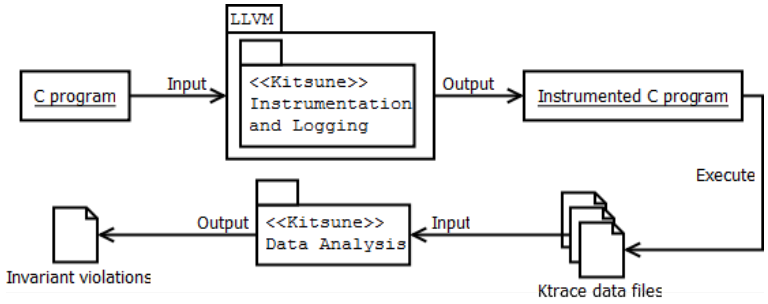

Figure 4.1: Simplified workflow of our prototype, Kitsune.

such does not rely on a clean exit of the program. However, for version 10 of printtokens2 and version 8 of schedule2, our implementation couldn't log sufficient data to run the invariant analysis so these have been discarded. In addition, version 32 of replace and version 9 of schedule2 have been discarded as they had no failing test cases, so the fault was never manifested.

This leaves us with 128 faulty versions that we used for the evaluation. In contrast, the nearest neighbour method of Renieris and Reiss used 109 faulty versions, and the Tarantula system of Jones and Harrold used 122 faulty versions.

4.2 Implementation

The technique was implemented in a prototype called Kitsune. The prototype consists of three components; an instrumentation and logging component, a data analysis component, and a controller component that ties the first two together. Figure 4.1 shows the (slightly simplified) workflow between the components of the prototype[1]. We will discuss each of these components in some more detail below, after which we will use the code example from listing 3.1 again to illustrate the process.

4.2.1 Instrumentation and logging

The instrumentation and logging component has as purpose to gather data from executions of programs. As mentioned at the start of the chapter on program invariants, calculating program invariants requires looking at all values encountered at certain program points. This component does just that; it instruments a program on a number of points and then logs all values encountered at these points to data files. This section details how this is realized.

As stated earlier, our goal is to analyse the Siemens suite of programs. These programs are written in the C programming language, so of course our instrumentation component should be able to support C. For this purpose, we used the LLVM Compiler Infrastructure [12]. LLVM transforms programs to an intermediate bitcode representation, much like the Java compiler. This bitcode can then be manipulated by writing an LLVM Pass using the framework provided by LLVM, written in C++. These manipulations are meant for optimizations, but it suits our instrumentation purposes very well.

The largest advantage that LLVM's intermediate bitcode representation has over other intermediate representations (such as the one Valgrind offers) is that it retains typing information. This was essential for our analysis purposes, which is why LLVM was the preferred solution.

We have also considered source code transformation tools (such as the Meta-environment [18]), to save the trouble of translating to an intermediate representation and thus retaining all source-level information. Instrumentation using these kind of tools however would require a disproportionate amount of time due

[1]The controller component is not explicitly shown in figure 4.1, as its purpose is to realize the transitions between the components. One could think of the controller as the collection of arrows.

to the ambiguity of the C language, with the added risk of not covering all cases. LLVM is simply more suitable for this purpose.

Our LLVM pass instruments a C program to perform calls to helper functions on relevant program points. For each combination of data type (such as integer and pointer) and program point (such as load and store) a different helper function is defined, that each accept a unique identifier for the instrumented point and the encountered value as arguments. The unique identifier is generated by incrementing a counter during instrumentation for every inserted call to a helper function[2].

Every helper function now has access to a unique identifier, the value that is encountered and the data type of the value. In addition, we also provide some debug information for each program point, namely the line of code in the original source where the value is encountered and (if possible) the name of the variable. When the instrumented program is executed, the helper functions log all available information to data files (which we call ktrace files), which will then be used as input to the data analysis component.

4.2.2 Data analysis

The purpose of our data analysis component is as follows. First, it will have to parse the ktrace data files from passing runs (created by the instrumentation and logging component) and calculate invariants based on the contents of these ktrace files. Then, the ktrace data files from failing runs have to be parsed, after which the contents of these ktrace files have to be tested against the calculated invariants. If invariant violations occur, they will have to be reported to the user.

Our data analysis tool is implemented in Java. As large amounts of data need to be parsed, this might not have been the most ideal choice. However, we are more experienced with Java and other languages wouldn't make enough of a difference to go through the trouble (relatively speaking) of programming in them, while not gaining very significant advantage in the end.

The output of this component is a fault report containing a list of program points where invariant violations occurred. A program point consists of the line number where the violation occurred and the associated variable name (if available). A violation reports which invariant was violated, the state of the invariant (for example, in case of the range invariant, the valid ranges) and the encountered value(s) that triggered the violation.

4.2.3 Controller

To tie the previous components together, we have written a Shell script. This script first instruments the specified faulty C program by using the instrumentation component. It then runs the program using every input and stores the ktrace files generated by running the program. It also determines whether the run was a passing run or not by comparing the output of the program to the output of the correct version. Finally, it runs our data analysis component with the ktrace files from passing and failing runs as arguments and stores its output to a file.

4.2.4 Example

To illustrate the working of the above components, we once again use the example of listing 3.1 (although slightly altered, as can be seen in listing 4.1). Note that we will show the instrumentation process on C-level. In reality, this of course happens in the bitcode representation of LLVM, but introducing this representation is outside the scope of this thesis. We will omit debug information provided to the helper functions (such as variable names) for simplicity.

As a first step, our instrumentation and logging component would instrument the code in listing 4.1

[2]In reality, this identifier is not completely unique. We found that the same program point could be used on different lines by the compiler, probably due to the reuse of basic blocks. We uniquely identify a program point now as the combination of the mentioned counter and the line where it is encountered.

```
1  read (int x, int y) {
2    int z, a;
3    a = x + y; /* should be x - y */
4
5    tmpresult = x < y;
6    if (tmpresult) {
7      z = a;
8    } else {
9      z = a + 1;
10   }
11   return z;
12 }
```

Listing 4.1: Slightly altered version of the code example in listing 3.1.

```
1  read (int x, int y) {
2    int z, a;
3    __kitsune_load_int(1, x);
4    __kitsune_load_int(2, y);
5    a = x + y; /* should be x - y */
6    __kitsune_store_int(3, a);
7
8    __kitsune_load_int(4, x);
9    __kitsune_load_int(5, y);
10   tmpresult = x < y;
11   __kitsune_compare(6, tmpresult);
12   if (tmpresult) {
13     __kitsune_load_int(7, a);
14     z = a;
15     __kitsune_store_int(8, z);
16   } else {
17     __kitsune_load_int(9, a);
18     z = a + 1;
19     __kitsune_store_int(10, z);
20   }
21   __kitsune_load_int(11, z);
22   return z;
23 }
```

Listing 4.2: Instrumented version of the code in listing 4.1.

with calls to helper functions for every program point where invariants will be calculated. The resulting instrumented code is shown in listing 4.2. The calls to helper functions start with *__kitsune* and their first argument are the unique identifiers for the program points[3]. When the code in listing 4.2 is executed, the helper functions are called that log the relevant information to the ktrace files.

Now, every time this instrumented code is run, we obtain a ktrace file that contains all encountered values for every executed program point in that run. By running this code with all passing inputs and all failing inputs we obtain two sets of ktrace files, which we then pass to our data analysis component. This component will calculate invariants based on the values in the passing files, after which it will check if any value in the failing files violate one of these invariants. This process is described in the example section at the end of the chapter on program invariants.

The result of the data analysis component is, as mentioned earlier, a fault report containing a list of program points where violations occur. Figure 4.2 shows a part of such a fault report, listing a violation of the range invariant and the sign-and-magnitude bitmask invariant. The program point referred to is variable a on line 3 of listing 4.1, and the values are from the example at the end of the chapter on program invariants.

[3]In this example, the identifiers ascend reading the code top to bottom. In reality, this is much more arbitrary.

```
At AnalysisPoint number 3
  Line number: 3
  Variable name: a
  Operation type: STORE
  Type of values: INT
The following invariant violation(s) occurred:
- RangeInvariant violated.
  Message: Value "2" did not fall in one of the valid ranges:
    Range: <-1,1>.
- SignAndMagnitudeBitmaskInvariant violated.
  Message: Value "2" with binary representation "0010" was not contained by
  the bitmask. First int value: "0", in binary: "0000". Bitmask binary: "0110".
  Difference violation and bitmask: "0010".
```

Figure 4.2: Report of a violation of a range invariant and a sign-and-magnitude bitmask invariant. The values are taken from the example in the chapter on program invariants.

4.3 Evaluation

The list of program points with corresponding invariant violations is our fault report, with every program point being an entry in this report. This fault report should help the user locate the defect in the code. To be able to compare the quality of our reports with the quality of the reports of other techniques, we will need to measure this quality.

As part of their evaluation framework, Renieris and Reiss proposed a score based on the program dependence graph of a program. We will now give a relatively brief description of how this score is calculated. More details can be found in their paper [16].

To calculate the score for a given fault report R, we do the following. First, we need to find the entry in the fault report that points to a node in the program dependence graph with the shortest path to a node where (an occurrence of) the defect exists. We will call the length of this path (with every edge having length 1) k. Then, we find all nodes we can reach by performing k steps from every program point indicated by our fault report. We call this set of nodes the dependency sphere k of the fault report, or $DS_k(R)$.

The score of the given fault report can now be calculated as follows:

$$score = 1 - \frac{|DS_k(R)|}{|PDG|} \tag{4.1}$$

where PDG is the complete program dependence graph. This score can be calculated for every fault report and is a measure for the percentage of the program that can be ignored by a user who is looking for a defect using that fault report.

Depending on the fault localization method under evaluation, a different mapping between entry in a fault report and node in the program dependence graph is required. For example, for many fault localization techniques the entries in the fault report are simply pointers to lines of code in a program (possibly with a certain score attached to it). The nodes in the program dependence graph that correspond to each entry are thus all nodes on that line of code.

Our method based on invariants has more expressive power in our fault reports than many other techniques however, providing us with advantages over other techniques. The first advantage is that the entries in the fault reports can actually point to specific variables on lines of code in many cases, so that part of a line can be ignored. Also, it can report the value(s) encountered that made a run fail, together with the "normal" values on that point.

It would be desirable to take these advantages into account when assigning scores, as the advantages would then be reflected in the scoring. Unfortunately, it is not possible to take the second advantage into consideration without compromising the reliability of the results, as the way reported values are

interpreted depends greatly on the user's knowledge of the program logic and his debugging skills. However, the first advantage can be taken into consideration by selecting exactly those nodes that correspond not only to the line number, but to the variable name as well. Usually, this is the node that is the tightest fit around the reported variable.

For example, consider a line of code containing the following expression:

```
if (c == 0 && d <= 5)
```

If an entry in the fault report would point at this line of code together with the variable d, the node with as content $d <= 5$ would be selected, as this would be the tightest fit.

To generate the dependence graph we used CodeSurfer [3], a commercial C/C++ code browser. CodeSurfer is capable of exporting a program dependence graph, which we have used for our scoring process.

Chapter 5

Results

This chapter will present the results obtained by performing the evaluation discussed in the previous chapter on the Siemens set of programs. First, we will name the techniques that we selected for comparison with Kitsune, our prototype. Then, we present our results compared to the results of these other techniques in the form of a number of tables and a graph.

5.1 Techniques used for comparison

The following techniques have been selected for comparison with the Kitsune system. The reference at each technique points to the paper where the scores for the technique have been reported.

- The best performing Nearest-Neighbour technique of Renieris and Reiss, namely the one using permutation distancing (NN/Perm) [16]. This technique finds a passing run that is most similar to a given failing run and then compares these to find faults.
- The Tarantula system of Jones and Harrold [11]. This technique is the best performing technique of the ones we use for comparison. The theory behind it is that the suspiciousness of a statement increases if it is executed more often during faulty runs and less often during passing runs.
- All three Cause-Transition techniques of Cleve and Zeller [4]. The various Cause-Transition technique works by finding locations in the code where some variable ceases to be a failure cause and another variable begins. CT is the standard Cause-Transitions technique, CT/Relevant is this same technique exploiting relevance in the ranking technique and CT/Infected is again this same technique exploiting infection knowledge. Note that these two last versions of the technique require human involvement.

Note that there are many other techniques available for comparison. The above techniques have been selected as we expect Kitsune's results to be comparable to two of them (Nearest-Neighbour and Cause-Transition), and to show its relation to one of the best existing techniques (Tarantula). The results reported in this section can easily be used to compare Kitsune with other techniques that used the same evaluation framework however.

5.2 Results

Table 5.1 compares the results of the other fault localization techniques with Kitsune. The leftmost column shows the score segments, corresponding to the scores obtained by equation 4.1, the formula described in the section about evaluation in the previous chapter. These scores signify the percentage

Score	Kitsune	NN/Perm	Tarantula	CT	CT/Rel	CT/Inf
99 – 100%	7,81%	0,00%	13,93%	4,65%	5,43%	4,55%
90 – 99%	14,06%	16,51%	41,80%	21,71%	30,23%	26,36%
80 – 90%	11,72%	9,17%	5,74%	11,63%	6,20%	10,91%
70 – 80%	10,16%	11,93%	9,84%	13,18%	6,20%	13,64%
60 – 70%	6,25%	13,76%	8,20%	1,55%	9,30%	4,55%
50 – 60%	5,47%	19,27%	7,38%	6,98%	10,08%	6,36%
40 – 50%	3,13%	3,67%	0,82%	3,10%	3,88%	1,82%
30 – 40%	3,13%	6,42%	0,82%	7,75%	10,08%	3,64%
20 – 30%	7,03%	1,83%	4,10%	4,65%	3,10%	7,27%
10 – 20%	4,69%	0,00%	7,38%	6,98%	10,85%	0,00%
0 – 10%	26,56%	17,43%	0,00%	17,83%	4,65%	20,91%

Table 5.1: Percentage of faulty versions at each score segment.

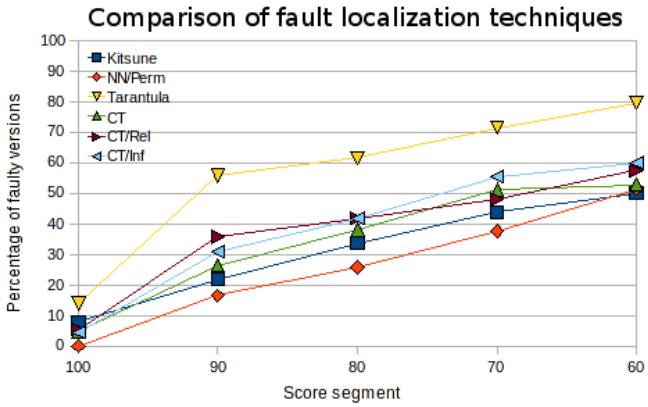

Figure 5.1: Percentages of faulty versions that are at or above each of the five highest score segments.

of the program that can be ignored by a debugger if he uses a certain technique. Each score segment is 10 percentage points except for the 99-100% range, following the convention introduced by [16]. The values in the other columns represent the percentage of faulty versions of which the fault report received a score in the corresponding score segment. Every column represents the results of a different technique. Simply said, the higher the percentages in the higher score segments, the better the technique performs.

Figure 5.1 contains the same information as table 5.1, but with a difference in presentation. Instead of showing the percentage of faulty versions that received a score in the corresponding segment as table 5.1 does, it shows the percentage of faulty versions that received a score in or higher than the corresponding segment. For example, we see in figure 5.1 that at score segment 80 the percentage of faulty versions for Kitsune is around 34%. This means that in 34% of the cases, the assigned score was 80% or higher, which corresponds with the results in table 5.1. It only shows this for the five highest score segments, for a better focus on the segments that matter most.

Table 5.2 shows the scores for Kitsune together with each of the used invariants separately, using the same format as the comparison of techniques in table 5.1. These results have been obtained by running our fault localization technique with only one invariant enabled, and then evaluating the results. This gives insight in how much each invariant contributed to the final result.

Table 5.3 shows the average score for Kitsune overall and each invariant separately, per program in the Siemens suite. The averages are calculated by taking the score for every faulty version and dividing it by the number of versions. Note that this results in every program having a different weight depending in the number of faulty versions that were considered. The situational invariants are included here for completeness only, as average scores do not carry much weight for situational invariants.

Score	Kitsune	Bitmask	SaM bitmsk	Range	NullPtr	Compare	NotExec
100%	7,81%	2,34%	2,34%	6,25%	0,00%	0,00%	2,34%
90 − 99%	14,06%	9,38%	10,16%	14,06%	0,78%	0,78%	2,34%
80 − 90%	11,72%	3,13%	3,91%	11,72%	0,78%	0,00%	1,56%
70 − 80%	10,16%	3,13%	3,13%	10,16%	0,00%	0,00%	0,78%
60 − 70%	6,25%	0,78%	0,78%	6,25%	0,00%	0,00%	0,78%
50 − 60%	5,47%	1,56%	0,78%	4,69%	0,00%	0,00%	0,00%
40 − 50%	3,13%	0,00%	1,56%	3,13%	0,00%	0,00%	0,00%
30 − 40%	3,13%	1,56%	1,56%	3,91%	0,00%	0,00%	0,00%
20 − 30%	7,03%	0,00%	0,78%	7,03%	0,00%	0,00%	0,00%
10 − 20%	4,69%	4,69%	4,69%	4,69%	0,00%	0,78%	0,78%
0 − 10%	26,56%	73,44%	70,31%	28,13%	98,44%	98,44%	91,41%

Table 5.2: Distribution of scores for each invariant separately.

Program	Kitsune	Bitmask	SaM bitmsk	Range	NullPtr	Compare	NotExec
print_tokens	90,21%	48,07%	48,07%	90,21%	0,00%	0,00%	12,57%
print_tokens2	88,56%	32,41%	32,41%	88,42%	0,00%	1,97%	21,02%
schedule	28,87%	11,10%	11,10%	28,87%	0,00%	0,00%	0,00%
schedule2	16,21%	4,20%	15,49%	16,21%	0,00%	0,00%	0,00%
replace	40,58%	10,45%	10,45%	38,93%	0,00%	0,00%	3,70%
tcas	47,56%	8,08%	11,85%	44,75%	0,00%	0,00%	4,34%
tot_info	64,41%	44,57%	44,05%	62,90%	7,77%	4,29%	15,13%
Overall	50,84%	19,08%	20,90%	49,26%	1,40%	0,90%	7,17%

Table 5.3: Average score for every program in the Siemens set and every invariant separately.

Finally, two other interesting observations have been made during the experimentation process, which couldn't easily be distilled into a table. Firstly, for nearly all ($> 95\%$) program points where a violation of a bitmask was reported, the range invariant was also violated. Secondly, there were some cases where the sign-and-magnitude bitmask invariant was violated and the normal bitmask invariant wasn't, but not vice versa.

Chapter 6

Discussion and conclusions

We have implemented and evaluated Kitsune, a fault localization system based on program invariants. According to table 5.1 and figure 5.1 it outperforms the Nearest Neighbour technique and performs nearly as well as the various Cause-Effect techniques. Tarantula still performs much better though, for a large part thanks to the fact that none of their scores fall in the lowest segment $(0-10\%)$, which often represents an empty fault report for Kitsune.

However, with more than 40% of the faulty versions reaching a score of 70% or higher, we can definitely say that fault localization based on program invariants has potential. This invalidates the suggestion of Pytlik et al. that invariants might not be suitable for fault localization at all [14]. Apparently, their notion of invariants was less suitable for this process, which is not that strange considering that the invariants they used weren't designed with fault localization as purpose.

In table 5.2 and 5.3 a breakdown is provided of the effectiveness of each invariant separately. Comparing the generic invariants, it is clear that the range invariant performs much better than the two bitmask invariants, almost performing as well as the overall performance of Kitsune. It seems that the bitmask invariants are not that well suited for fault localization after all, an observation that contrasts sharply with the findings of Hangal and Lam [8], who reported very positive results using only this invariant.

In this thesis, we have introduced the sign-and-magnitude bitmask invariant, a variation on the normal bitmask invariant that should be better suited to deal with variables that can take on both positive as well as negative numbers. From the results in tables 5.2 and 5.3, and from the observation that there were only cases where the sign-and-magnitude bitmask invariant was violated and the normal bitmask invariant wasn't and not vice versa, we can conclude that the sign-and-magnitude bitmask invariant performs better than the normal bitmask invariant, although not by a very large margin. This small margin is to be expected though as most of their functionality is the same, however it does show that in those cases where they differ, the sign-and-magnitude bitmask invariant outperforms the normal bitmask invariant.

We also introduced three situational invariants in this thesis. From the results in the previous section, most notably table 5.2, we observe that the not executed invariant performs quite well for those cases where it is violated. The compare invariant and null pointer invariant hardly ever got violated though, but in the case of the null pointer invariant it at least performed quite well in those cases.

Finally, table 5.3 also shows a breakdown of the effectiveness of our technique on every program in the Siemens suite separately. We see that Kitsune's performance fluctuates a bit between programs, suggesting that its effectiveness might be linked to certain programs or faults. Other techniques don't provide such a breakdown however, so we can't really say a lot about this at this point.

```
1  void initialize() {
2    Positive_RA_Alt_Thresh[0] = 400;
3    Positive_RA_Alt_Thresh[1] = 500;
4    Positive_RA_Alt_Thresh[2] = 640;
5    Positive_RA_Alt_Thresh[3] = 740+20; // Seeded fault: "+20".
6  }
7  int ALIM () {
8    return Positive_RA_Alt_Thresh[Alt_Layer_Value];
9  }
10 bool Non_Crossing_Biased_Climb() {
11   ...
12   result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >=
         ALIM())));
13   ...
14 }
```

Listing 6.1: Code snippet from version 19 of tcas. The faulty statement is on line 5.

6.1 Threats to validity

A number of threats to validity exist with regards to this research. One of the largest is the fact that results obtained from the evaluation of techniques based on the Siemens set of programs can not necessarily be generalized to arbitrary programs, such as larger programs or programs with multiple faults. However, this goes for most evaluations of fault localization techniques up till now. Until the degree of representativeness of the Siemens set of programs has been determined and the faults have been categorized, this will remain a problem.

Also, the evaluation procedure has some limitations. For the technique discussed in this thesis, the expressive power of fault reports that invariant violations provide is not fully reflected in the score. With this expressive power we mean the ability of the fault reports to report the value(s) accepted as valid and the value(s) that violated the invariants. Interpreting these values can have a large influence in the effectiveness of our technique.

To illustrate this, consider the code listing in listing 6.1, which is a code snippet from faulty version 19 of the program *tcas*. The fault is located on line 5. One of the three violations that Kitsune reported was a violation of a range invariant on line 12. The value of the variable *Down_Separation* was 743 in a failed run, while the valid ranges were -100 to 421, 447 to 741 and 799 to 930. Anyone that has some knowledge of this program will quickly look at line 5 as that is the only relevant value of *Positive_RA_Alt_Thresh* in this case and, if we'd assume a debugger identifies a fault the moment he sees it, would find the fault very quickly. However, using the scoring procedure, we received a score of only 17,59% for this report.

We admit that it does seem very difficult to automate taking all of these factors into account, especially between totally different and arbitrary techniques. A possibility would be to evaluate the various techniques using humans, measuring their performance in finding faults using different techniques. However, this of course introduces a wealth of threats to validity on its own.

Our conclusion is that it is probably best for our technique (and any other more complex technique) to try to meet the evaluation process halfway by providing some extra information to guide the evaluation process, for example by providing a ranking of entries in the fault report. A possibility for such a ranking for our technique is introduced in the next chapter on future work.

Chapter 7

Future work

Tarantula has proven to be a very good performer when it comes to fault localization. Apparently, the theory behind their system (statements that get executed relatively often during faulty runs are more suspicious) is quite effective. However, our technique has as advantages that it provides much more expressive power in fault reports and that it can also capture semantics. If the power from Tarantula could be combined with our technique, we could improve the performance of invariant based fault localization significantly without trading in these advantages.

We think that this could be done by using a metric similar to what Tarantula uses for ranking suspicious statements to rank invariant violations. Abreu et al. [2] found that out of three different similarity coefficients that can be used as metrics for this ranking (including Tarantula's metric), the Ochiai similarity coefficient, known from the biology domain, performs the best of the three. We expect that it would be very rewarding to use this coefficient as metric to rank invariant violations (and other statements, possibly based on their proximity to violations), adding some of the power of Tarantula's theory to invariant based fault localization. More details on these similarity coefficients / metrics can be found in [2].

Our goal with Kitsune was to design a generic fault localization system. However, due to the flexibility of program invariants as a technique, it is easily possible to implement invariants that are more domain specific. For example, one could gather information about encountered faults during a software system's development and then (possibly automatically) design invariants to detect and localize these specific faults.

Earlier in this thesis, we mentioned how Pytlik et al. achieved negative results by using the notion of invariants implemented in Daikon by Ernst et al. for fault localization. However, one cannot ignore the wealth of invariants that Daikon can detect. It might be worth it to combine part of Daikon's notion of invariants with the much simpler invariants that we use, as it is not unthinkable that Daikon's invariants can detect faults that simple invariants can not, and vice versa.

There's also still room left for tweaking on the invariants introduced in this thesis. For example, the compare invariant will never report a violation if both true and false have been encountered during passing runs. However, if for example not all but *almost* all encountered values were true, it might still be worth it to report a violation when false is encountered, just with a lower confidence. The same goes for the null pointer invariant. This might improve the performance of these invariants.

In addition, while writing this thesis, my supervisor suggested that the problems with the bitmask invariants encountering both positive as well as negative values could possibly be omitted by simply using two bitmasks per bitmask invariant, one for positive values and one for negative values. Positive and negative values then don't interfere with each other at all, while the special properties of the bitmask invariant will be retained (if implemented correctly). This will however make the bitmask invariant cost more in terms of memory and overhead, which might be an issue in for example embedded systems.

Also the range invariant, the best performing invariant, should still be subject to some tweaking. We have used three ranges, but perhaps it performs better when more or less ranges are used, or when the

ranges are determined dynamically based on the encountered values. In addition, a confidence value for the range invariant would be a good way to help prioritize between invariant violations, as this invariant is prone to encountering false positives.

Finally, as mentioned in the threats to validity section, an analysis of the degree of representativeness of the Siemens suite for the evaluation of fault localization techniques is essential if the results based on it are ever to be generalized to arbitrary programs. At the very least, the types of faults have to be categorized so that something can be said about the performance of techniques on certain types of faults.

Chapter 8

Acknowledgements

I would like to thank my supervisor, Jan van Eijck, for his feedback and advice whenever it was needed, as well as the staff of the SEN department of the CWI for giving me the opportunity and freedom to perform this research. I am also grateful to Paul Griffioen, for his always-present enthusiasm and readiness to discuss and advice. My thanks also go to Alberto Gonzalez Sanchez from the TU Delft for his assistance with the instrumentation component, and to Manos Renieris from Brown University / Google for his help on the evaluation part.

I would also like to thank all teachers and students of the Software Engineering course at the University of Amsterdam, for making this one of the most instructive and memorable years of my academic career. A special mention goes to Alex Hartog, Jeroen Bach and Michel de Graaf, for forming the most awesome group ever. Finally, special thanks go to my dear Sheila, for putting up with me this last year.

Bibliography

[1] R. Abreu, A. González Sánchez, P. Zoeteweij, and A.J.C. van Gemund. Automatic software fault localization using generic program invariants. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 712–717, 2008.

[2] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 89–98, 2007.

[3] P. Anderson and T. Teitelbaum. Software inspection using codesurfer. In *Proceedings of the first Workshop on Inspection in Software Engineering*, 2001.

[4] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, 2005.

[5] R.A. DeMillo, H. Pan, and E.H. Spafford. Critical slicing for software fault localization. *ACM SIGSOFT Software Engineering Notes*, pages 121–134, 1996.

[6] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 213–224, 1999.

[7] N. Gupta, X. Zhang H. He, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 263–272, 2005.

[8] S. Hangal and M.S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, 2002.

[9] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow– and controlflow–based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, 1994.

[10] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 167–178, 2008.

[11] J.A. Jones and M.J. Harrold. Empirical evaluation of the tarantula automatic fault–localization technique. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[12] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, 2004.

[13] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28:129–137, 1982.

[14] B. Pytlik, M. Renieris, S. Krishnamurthi, and S.P. Reiss. Automated fault localization using potential invariants. 2003.

[15] P. Racunas, K. Constantinides, S. Manne, and S.S. Mukherjee. Perturbation-based fault screening. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 169–180, 2007.

[16] M. Renieris and S.P. Reiss. Fault localization with nearest neighbor queries. *Automated Software Engineering*, pages 30–39, 2003.

[17] G. Rothermel, A. Kinneer S. Elbaum, and H. Do. Software-artifact infrastructure repository.

[18] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, L. Moonen P. Klint, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 365–370, 2001.

[19] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 347–351, 2005.

[20] A. Zeller. Isolating cause-effect chains from computer programs. *ACM SIGSOFT Software Engineering Notes*, 27(6):1–10, 2002.

[21] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 272–281, 2006.