Fault Localization & Relevance Analysis

Numair Mansur

August 6, 2016

1 Introduction

The most tedious task in a programmer's routine is to spend time on debugging
and to determine the cause of the error. Debugging can be made much more
simpler if we can determine automatically what program segments are actually
responsible for the error. This is called **"Fault Localization"**.
Fault Localization encompasses the task of identification of the program state-
ments that are **relevant** for the error trace and determining the variables whose
values should be tracked in order to understand the cause of the error.
There can be many notions of error relevancy. During my master praktikum,
I worked with and developed algorithm for two new relevancy criterion, which
we called Flow-Sensitive and non Flow-Sensitive error relevancy. A part of this
work will be to formalize these two new types of error relevancy. But first we
need a basic formal definition to "relevancy", that is, what does it mean that
a statement is relevant for an error. Surprisingly enough, I was unable to find
a formal and a generally accepted definition of error relevancy in the literature.
But this also makes sense because relevancy is not really a well established con-
cept and is a hard thing to define, let alone formally. Therefore, another aim of
this project is to try to give a formal definition to error relevancy.

1.1 Relevant Work

In the paper Flow-Sensitive Fault Localization [1], the authors take into account
the flow of the program while determining the statements that are relevant for
the error. They do it by a new flow-sensitive encoding of error traces into trace
formulas. After which, by identifying the irrelevant portions of the new trace
formula, we can isolate the possible cause of the error. The result of a flow-
sensitive fault localization not only explains why the error occurred, but also
justifies why the statements leading to the error were executed[1].
In the Error invariants paper [2] ..
Explaining inconsistent code paper ..

1

2   Goals/Approach

2.1   Formalizations

During my Master Praktikum, I worked on a fault localization algorithm that find relevant statements in an error with respect to two error relevance criterion namely Flow Sensitive and Non-Flow Sensitive relevance criterion. I want to formalize these two notions of error relevance. There are also two sub categories in a relevance criteria which also need a formal definition.

(1) Predetermined input (which we called the UC case)
(2) Non-deterministic input (havoc).

Another task in this step would be to formalize our already implemented fault localization algorithm .

2.2   Security Bugs

We also discovered during my Praktikum that we may be able to perform a security analysis via our fault localization algorithm and distinguish security bugs in a program from all other bugs. I want to formalize what it means that a bug is a security bug and use the algorithm to precisely find them. I expect some modifications in our algorithm so that it can correctly classify an error as a security/non-security error. Following steps are expected to complete this task:
1) Analyse examples containing security bugs.
2) Formalize the definition of a security bug that satisfies all the examples.
3) Check if the formal definition fits correctly with the examples.
4) Modify our fault localization algorithm so that it models the definition correctly.

2.3   Verification of the results

I also want to make a mechanism to verify that what we are computing is correct that includes verifying the result of our fault localization algorithm and security bug analysis. For this task i plan to separately implement checks and compare it with our algorithms.

2.4   Broader analysis of the program

Currently we are doing the analysis only on the error trace (or counter example), i also want investigate how we can broaden our perspective from an error trace to a whole program. This may also give us some new definitions for relevance for example, statements that are directly responsible, statements that cause the execution of the directly responsible statements. As an example, consider the code below:

```
1
2   foo ()
3   {
4      int z;
5      int x;
6      if (x==0)
7      {
8         if (z==0)
9         {
10           y := 1;
11        }
12        else
13        {
14           while (true)
15           {
16
17           }
18        }
19        assert (false);
20     }
21  }
```

π in this example is:

$$assume(x == 0); assume(z == 0); y = 1)$$

and our algorithm shows that none of the statements is relevant. But if we look at the program from a broader perspective then an error trace, we will notice that the value of z is relevant because if $z \neq 0$ then we would be stuck in a never ending loop and would not be able to reach the error. So we cannot say that $assume(z = 0)$ is irrelevant for the error.

3   Time Frame

This is a very rough estimate but i plan to give one month to each task, but it is also possible to give a lower amount of time to one task and give that extra time to the other task.
Step 1: End of August.
Step 2: End of August.
Step 3: End of September.
Step 4: End of October.

4 Formalization of Relevancy Criterion

4.1 Error Relevancy

In a program containing a set of statements ST, let $\pi \in ST$ be a sequence of statements. Let ϕ be an assertion condition, such that if π is executed, the assertion condition ϕ is violated. We call this an error. It is possible that ϕ will not be violated after all possible executions of π. It will only be violated if we start the execution of π from a specific state which belongs to a set of states, which we call Error-Precondition EP.

A set of statements $REL \in P(\pi)$ is called relevant iff REL is a set which have a minimum size in $P(\pi)$ and the following property holds:

"If all the assignments in REL from π are replaced with a havoc statement to get a new path π' and if we now execute π' , starting from a state which is in EP, it is not guaranteed that the assertion condition ϕ is violated any more. i.e $\forall \psi \in EP$, the execution of π' is not guaranteed to violate ϕ".

[!!!!!] This definition only holds for our flow sensitive case, not for the non-flowsensitive case because for some non-flow relevant statements, changing them with a havoc still guarantees that we end up violating the assertion condition in the end

[[[REL must also represent a trace that is actually feasible [Define feasible trace here !]]]].

Consider the code below:

```
1
2  foo ()
3  {
4     x  :=  1;
5     x--;
6     x++;
7     assert (x==0);
8  }
```

In the above example:
$\psi = True$
$\pi = \{x = 1, x - -, x + +\}$
$\phi \Rightarrow assume(x == 0)$

$$P(\pi) = \{\{x = 1\}, \{x--\}, \{x++\}, \{x = 1, x--\}, \{x = 1, x++\}, \{x--, x++\}, \{x = 1, x--, x++\}\}$$

$Rel = \{x++\}$ because if the statement in REL is replaced by a havoc statement in π to get a new path $\pi' = \{x = 1, x - -, havoc(x)\}$ then there is no guarantee that the assertion condition ϕ is violated any more after the execution of π'.

Consider another example:

```
1
2  foo ()
3  {
4     x  :=  1;
5     y  :=  1;
6     y  :=  2;
```

```
7    If ( y == 2 )
8    {
9      x := 0;
10   }
11   assert (x==1);
12 }
```

In this example:
$\pi = \{x = 1; y = 1; y = 2; x = 0\}$
$\phi = assume[x == 1]$
$REL = \{x = 0\}$
$\pi' = \{x = 1, y = 1, y = 2, havoc(x)\}$

4.2 Non-Flow Sensitive Relevance criteria

4.2.1 Preliminary definitions

Let ST be a set containing all the statements in a program. $\pi \in ST$ is called an Error Trace iff there is an assertion condition ϕ, which is violated if we execute π, starting from a state which belongs to a set of states called Error Precondition EP. We call the violation of π an error.

For some given program statement S and some postcondition R there is a (possibly empty) set of program states such that if execution of S is initiated from one of these states then S is guaranteed to terminate in a state for which R is true. The weakest precondition of S with respect to R, normally written $WP(S, R)$ is a predicate that characterizes this set of states.

4.2.2 Definition

One possible definition of relevancy could be that in an error trace π, an assignment statement $\pi[i]$ is relevant if after replacing that statement with havoc (non-deterministic input), then we can't be certain that the assertion condition ϕ will be violated any more.

For example:

```
1  foo ()
2  {
3    x := 1;
4    y := 2;
5    If ( x == 2 )
6    {
7      y++;
8    }
9    assert (y !=3 );
10 }
```

Error Trace π will be:

```
1  x := 1;
2  y := 2;
3  assume ( x==1 )
4  y := 3;
5  assume ( y==3 )
```

If we replace $y := 3$ by $hacov(y)$, then y can non-deterministically be assigned any value and we can't be certain any more that the assertion condition in the end $\phi = (y! = 3)$ will be violated.

4.2.3 Algorithm

Let π be an error trace of length n, $WP()$ the weakest pre-condition operator, $\pi[i]$ the ith statement of π and $\pi[i, j]$ be the sub-trace $\pi[i]...\pi[j-1]$.

Let ϕ be a post-condition and $\phi = WP(False, \pi[i+1, n])$.
Let Ψ be pre-condition and $\Psi = \neg WP(False, \pi[i, n])$.
Where n is the length of the error trace. A statement $\pi[i]$ in an error trace π is relevant with respect to the non-flow sensitive criteria $(*)$ if the conjunction of the the three formulas $(\psi, \pi[i], \neg\phi)$ is unsatisfiable and $\pi[i]$ is in the unsatisfiable core.
$\pi[i]$ is considered relevant with respect to non-flow sensitive criteria $(@)$ if $(\psi, \pi[i], \neg\phi)$ is satisfiable. (In this case, $\pi[i]$ is a non-deterministic input statement $(Havoc)$).
For example consider the following code:

```
1  foo ()
2  {
3     var y:  int ;
4     var x:  int ;
5     var z:  int ;
6
7     y :=  0;
8     z :=  0;
9     if (y==0)
10    {
11       x :=  1;
12       if ( z==0)
13       {
14          z :=  1;
15       }
16    }
17    else
18    {
19       y :=  2;
20       y :=  3;
21       y :=  4;
22    }
23    assert (x == 0);
24 }
```

For the above code we will get the following error trace:

```
1  y :=  0;  [*]
2  z :=  0;  [*]
3  assume  (y == 0);
4  x :=  1;  [*]
5  assume  (z == 0);
6  z :=  1;
7  assume  !(x == 0);
```

In the above trace, for $i = 4$,

$\psi = \neg WP(False, \pi[4,7]) \implies z = 0$
$\pi[4] \implies x = 1$
$\phi = WP(False, \pi[5,7]) \implies x = 0 \lor \neg(z = 0)$

The formula $(\psi, \pi[4], \neg\phi)$ is unsatisfiable and $\pi[4]$ is in the unsatisfiable core. Hence $\pi[4]$ is relevant with respect to Non-Flow Sensitive relevance criterion and we therefore label it with a $[*]$.

4.3   Flow Sensitive Relevance criteria

The definition of relevancy is actually similar to the non-flow sensitive case. The only thing different is that now we take branches in the trace into account. If there is a branch in the error trace, the relevancy of the statements inside the error trace will only be calculated if the branch as a whole is relevant to the error. We see that we can get slightly different results from the Non-Flow Sensitive case because it is possible that a statement might be relevant with respect to non-flow sensitive case but it lies inside a branch which is not relevant.
When we say that a branch $\pi[i,j]$ must be relevant to the error, we mean that the branch is reduced to a transition formula which we call a "$Markhor Formula$" and it's relevancy is calculated in the same way as we did for a statement in the non-flow sensitive case (by checking if the conjunction of $(\psi, \pi[i], \neg\phi)$ is unsatisfiable and the statement is in the unsatisfiable core).

4.3.1   Markhor Formula

References

[1] J. Christ, E. Ermis, M. Schaf, and T. Wies.Flow-sensitive fault localization. In VMCAI, volume 7737, pages 189–208, Berlin, Heidelberg, 2013. Springer

[2] E.Ermis, M. Schaf, and T. Wies. Error Invariants. In FM'12, pages 338–353. Springer, 2012.