Fault Localization & Relevance Analysis

(For the latest version: http://numairmansur.github.io/Error-Localization/project.pdf)

Numair Mansur

August 4, 2016

1 Introduction

The most time consuming part in a programmer's routine is to spend time on debugging and to determine the cause of the error and what statements are actually responsible for the error. This is called **"Fault Localization"** .
Fault Localization encompasses the task of identification of the program statements 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 relevancy. During my master's laboratory, i worked with two, which we called Flow-Sensitive and non Flow-Sensitive relevancy criterion. Both of these types will be explained in detail, but we found out that for the same error, statements that are relevant with respect to one relevancy criteria might not be relevant at all with respect to the other one.But first we need a basic formal definition to "relevancy", that is, what does it mean that a statement is relevant for an error.

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 very important task in this step would be to formalize our 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 two kinds of bugs called Security and non-Security 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) Verify 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;
```

```
  If ( y == 2 )
  {
    x := 0;
  }
  assert(x==1);
}
```

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

Let π be an error trace of length n, $WP()$ the weakest pre-condition operator, $\pi[i]$ the i^{th} 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:

```
foo()
{
  var y: int;
  var x: int;
  var z: int;

  y := 0;
  z := 0;
  if(y==0)
  {
    x := 1;
    if(z==0)
    {
      z := 1;
    }
  }
  else
  {
    y := 2;
    y := 3;
    y := 4;
  }
  assert(x == 0);
}
```

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 $i = 4$,

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

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 "$MarkhorFormula$" 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).

To Do:
-TALK A LITTLE MORE ABOUT THE GENERAL IDEA.
- Define what is markhor formula.
- Then define a recursive definition of relevancy w.r.t the flow sensitive relevancy criteria.

Now we also have to take into account the relevancy of branches in the trace. The relevancy of branch $\pi[i, j]$ can still be calculated in the same way as a relevancy of statement in an error trace $\pi[i]$.
Ofcourse we need a transition formula for the branch which we call a "$Markhor\ formula$".

We check if the branch is relevant or not. If it is relevant then we go into the branch and then check for the relevant statmetnts. It might be possible that some statmetns might be relevant if we see them in the context of the whole program and dont take branches into account. But now here we take the branch into account, check if the branch is relevant or not. IF IT IS RELEVANT then and only then we go into the branch and check for the relevant statements. That is the ONLY FUNCTIONAL DIFFERENCE between flow sensitive and non-flow senesitive analysis.
The magic happens when we have to determine how we calculate the relevancy of the branch. We calculate the MARKHOR FORMULA of the branch and check if that Markhor formula is un-satisfiable along with the PRE-CONDTION and POST.

4.3.1 Markhor Formula

Check the exact form of Markhor formula from the code.

5 References