Fault Localization & Relevance Analysis

Numair Mansur

June 22, 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.

1.1 Relevancy

In a program with a set of statements ST, let π be a sequence of statements from ST whose execution produced an error. By error, we mean that there is an assertion condition ϕ that is violated if we execute π. It is possible that ϕ will not be violated in all cases. It will only be violated if we start the execution of π from a specific state. Infact, there is a whole set of states, which we call Error-Precondition EP, such that if we start the execution of π from a state $\psi \in EP$, we will always violate ϕ.
A set of statements $REL \in \pi$ are called relevant, if removing all the statements in REL from π will give us a new path π' and if we now execute π' , starting from a state which is in EP, the assertion condition ϕ is not violated any more. i.e $\vee\psi \in EP$, the execution of π' will not violate ϕ.

Figure 1: relevancy examples

Consider the code below:

```
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:
$\pi \Rightarrow \{x = 1; y = 1; y = 2; x = 0\}$
$\phi \Rightarrow assume[x == 1]$
$REL \Rightarrow \{y = 2, x = 0\}$
$\pi' \Rightarrow \{x = 1; y = 1\}$

2 Goals

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) Unsatisfiable core
(2) Non-deterministic input (havoc).

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.
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.
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 }
```

The error trace is

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

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 Approach

Following are the major steps (in order) that i plan to complete in the master project. It is possible that each of these steps also might have further sub-steps.

- Formal definitions of relevance, flow sensitive faults, non-flow sensitive faults.
- Formal definition of Security bugs.
- Modification of our fault localization algorithm to also identify security/non-security bugs.
- Verification of the results.
- Broader analysis of the program from just from an error trace.