Fault Localization & Relevance Analysis

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

Numair Mansur

September 5, 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 simpler if we can determine automatically what program segments are actually responsible for the error. This process of automatically determining error locations 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 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 a well established concept 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], the authors introduce a new concept in which for each position in an error trace they find a formula over program variables that over-approximates the reachable states at the given position while only

capturing the states that will still produce the error, if execution of the trace is continued from that position. We can find the relevant statements in the trace with the help of inductive error invariants. Inductive invariants are those error invariants that hold for consecutive positions in an error trace and hence characterize statements in the trace that are irrelevant for the trace [2].
In the paper Explaining inconsistent code [3] the authors use the idea presented in Error Invariants to find and explain inconsisitencies in a program using something called an "$ErrorInvariantAutomaton$". The error invariant automaton is an abstraction of the input code fragment that only mentions program statements and facts that are relevant for understanding the cause of the inconsistency.

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 Performance Analysis of the Algorithm

I want to test the algorithm on big error traces and analyse its performance and find out where can the performance be improved. This might also require some modifications in the algorithm.

2.2.1 Non-Flow Sensitive Analysis

...

2.2.2 Flow-Sensitive Analysis

...

2.3 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.

2.4 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.

2.5 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 Formalization of Relevancy Criterion

3.1 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:
$\pi = \{x = 1; y = 1; y = 2; x = 0\}$
$\phi = assume[x == 1]$
$REL = \{x = 0\}$
$\pi' = \{x = 1, y = 1, y = 2, havoc(x)\}$

3.2 Non-Flow Sensitive Relevance criteria

3.2.1 Preliminary definitions (new ones)

Introduction:

Proof based fault localization techniques rely on encoding of an error trace into a so called error trace formula. A ***error trace formula*** is an unsatisfiable logical formula (or is it ? isn't it unsatisfiable together with the post condition). An error is observable by executing an error trace starting from a state that satisfies an ***error pre-condition***.
We will encode the (error?)trace into a so called extended trace formula which consists the error precondition, correctness assertion and the trace formula of the trace. When the extended trace formula is unsatisfiable, we say that the trace in the extended trace formula is an error trace and we say that we are seeing an "error".

Formal Setting:

Let X be a fixed set of variables which we call "program variables", and let L be a set of labels. We denote by $Expr(X)$ the set of terms built from variables in X and we denote by $Preds(X)$ the set of all quantifier-free formulas with free variables in X.
A *program statement st* is either a conditional choice, a while loop, a sequence of statements, an assignment or an error label.

$$st := if\ \textbf{cond}\ then\ \textbf{st}\ else\ \textbf{st} \mid while\ \textbf{cond}\ do\ \textbf{st} \mid st; st \mid x := e \mid label\ error$$

Error labels have no operational meaning. They are only used to uniquely identify the error states at certain points during the execution of the program statement.
A program $P =< st, \Phi >$ consists of a program statement st, and an assertion map.

$$\Phi : error \to Preds(X)$$

which maps each error label *error* in st to a formula that should not hold at the point of execution of *st* determined by *error*.
We define the semantics of program statements and programs in terms of *atomic statements* which are defined by the following grammar.

$$st^a := assume(\textbf{cond}) \mid x := e \mid label\ error \mid havoc\ \textbf{cond}$$

A ***trace*** is a finite sequence of atomic statements. Let π be a trace of length n. A program state S is a function that assigns a value $S(x)$ to each program variable $x \in X$. Formulas $Preds(X)$ can also be called state formulas and we write $S \Rightarrow F$ to denote that a state S satisfies a state formula F.
The formulas $Preds(X \cup X')$ are called ***transition formulas***. We use transition formulas to represent the semantics of atomic statements in a trace, where the variables X' denote the values of the variables from X in the next state. We write $s, s' \Rightarrow T$ to denote that states s and s' satisfy the transition formula T.
An ***execution*** σ of a trace π of length n is a sequence of states $s_0....s_n$ such that for all $0 \leq i \leq n$, $s_i, s_{i+1} \Rightarrow T[\pi[i]]$. We denote by $Execs(\pi)$ the set of all executions of π. The trace formula of π is the formula:

$$TF(\pi) := T[\pi[0]]^{<0>} \wedge ..... \wedge T[\pi[n-1]]^{<n-1>}$$

A trace is called feasible if its trace formula is satisfiable, otherwise we call it infeasible.
A program $P =< st, \Phi >$ is safe if for every trace $\pi \in Trace(P)$ whose final statement is a label statement and every execution σ of π, the final state of σ satisfies $\Phi(l)$.
If a program is not safe, an error can be witnessed along a trace. The error corresponds to a set of executions that violate the assertion associated with the last label of the trace.
Error & Error Trace: For an unsafe program $P =< st, \Phi >$, with a trace $\pi = st_0^a; .....st_{n-2}^a; label\ error$, an error precondition Ψ, a state formula ϕ, such that $\Phi(error) = \phi$, we say that there is an ***error*** in the program P if the following condition hold :
$\Psi \wedge TF(\pi) \wedge \phi^{<n>}$ is satisfiable.
In this case π is called an ***error trace***. Hence for an error trace π, no execution of the trace π that starts in a state satisfying the ***error pre-condition*** Ψ ends in a state satisfying the postcondition ϕ.
Weakest Precondition Operator: For some given formula $R \in Preds(X)$, and a program statement *st*, the weakest precondition operator returns the weakest formula that must be true such that if we execute *st*, R holds.

$$wp(st, R) \wedge T[st] \Rightarrow R$$

.

3.2.2 Preliminary definitions (to be discarded)

Weakest Precondition Operator (Def 1): 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.
Def 2: For some given logical predicate R, and a program statement ST, $wp(R, ST)$ is defined to be the weakest logical predicate that must be true before executing S in order to prove that R is true afterwards. [4].
Let us assume variable x denotes the tuple of variables involved in statement ST. Then, a given Hoare Triple $\{P\}ST\{R\}$ is provable in Hoare Logic for total correctness if and only if the first-order predicate below holds:

$$\forall x, P \Rightarrow wp(ST, R)$$

.

3.2.3 Definition

If a relevant assignment statement in an error trace is removed or replaced by a non-deterministic input statement (havoc), then this can alter the observable error and we cannot guarantee any more that we will reach the error after executing the new error trace.
More formally, we say that in a error trace π, an assignment statement $\pi[i]$ is relevant w.r.t non-flow sensitive relevancy criteria if after replacing that statement in the trace with a havoc statement to get a new trace π' , there exists an execution σ' in $Execs(\pi')$, such that $\Psi \wedge TF(\pi') \wedge \phi^{<n>}$ is satisfiable.
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 there will be an execution for which condition in the end $\phi = (y! = 3)$ will not be violated and the formula $\Psi \wedge TF(\pi') \wedge \phi^{<n>}$ is satisfiable.

3.2.4 Algorithm

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:

```
1  foo()
2  {
3    int y;
4    int x;
5    int z;
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 $i = 4$,

$\psi = \neg WP(False, \pi[4, 7]) \Longrightarrow z = 0$
$\pi[4] \Longrightarrow x = 1$

$\phi = WP(False, \pi[5,7]) \Longrightarrow x = 0 \vee \neg(z = 0)$

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 $[*]$.

3.2.5 Example

...

3.3 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).

3.3.1 Markhor Formula

.
.
.

4 Performance Analysis

4.1 Non-Flow Sensitive Fault Localization:

I think our algorithm is more effected not by the size of the trace, but the size of the formulas in the trace. That's why, sometimes even for small traces, the non-flow sensitive analysis is fast but it takes very long for flow-sensitive analysis to run, not because of the size of the trace , but the complexity of the formulas in it.
Therefore there might not be much we can do to increase the performance of non-flow sensitive analysis because there are not many steps involved and the formulas we compute here (wp, pre) are our most basic requirement to compute the relevance of a statement and their calculation cannot be further simplified.

5 Security Errors

A very interesting problem that we might be able to solve via our fault localization algorithm is to distinguish security error from all other kinds of errors. According to our current definition an error is a security error iff an input statement (network/user) can cause the program execution to reach the error. Following are the (current) criterion for a security error.
1. There is some reachable location where the program reads input.
2. There is some input value, such that continuing from this location we definitely reach the error
3. There is some input that continuing from this location we do not reach the error.
From condition 2 and 3 we can deduce that the input is somehow relevant for the error.
For example:

```
1  foo()
2  {
3    int y;
4    int x;
5    x := 1;
6    y := user_input();
7    if(y==2)
8    {
9      y := y+1;
10   }
11   assert(y != 3);
12 }
```

In the above example, the input statement $y := user\_input()$ determines if we reach the error or not. Hence there is a security error in this program.

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.

[3] M. Schaf, D. Schawrtz, T. Wies. Explaining Inconsistent Code. In Joint meeting of the European Software Engineering conference and the Symposium on the Foundations of Software Engineering, ESEC/FSE'13, pages: 521 - 531, Saint Petersburg, Russian Federation. August 18-26,2013

[4] https://courses.cs.washington.edu/courses/cse503/06sp/correctness2.pdf