Part (I) Building an Automated Bug Detection Tool

(a) Inferring Likely Invariants for Bug Detection

**Description of the code:**

1. Read the input and set the threshold according to to inputs.
2. Parse the call graph and save the graph information into HashMap and HashSet.

**HashSet<Node> Nodes:** save all functions name.

**HashMap<Node, HashSet<Node>> callMap:** key: all the caller in the call graph

Value: callees called by key

**HashMap<Node, Integer> NodeSupport:** key: function name value: the support value of key

**HashMap<Pair, HashSet<Node>> pairMap:** key: all the pairs that occurred in the call map value: functions that called the pair

1. Choose the pair support value that satisfy the threshold and save the value in to HashMap.

**HashMap<Pair, Integer> pairSupport:** key:pair name value: the support value of key

1. Parse each pair and find bugs.
2. Print the bugs.

(b) Finding and Explaining False Positives

**Reasons for false positives:**

1. **We only considered intra-procedural analysis in (a). Since a callee may be a caller for another function, we should also consider the content in callees (inter-procedural).**

For example, if A() calls A1(), A2(), A3(), B() and B() calls B1(), B2(), B3(). And if there is a bug: A1 in A(), pair <A1,B1> it will be a false positive.

1. **Even if the support of a pair is very large, two functions in this pair may have no or low correlation.**

In some programs, two functions are sometimes having no correlation with each other, but coincidently always appear together. We should also consider the semantic of the program and then find these false positives.

1. **The threshold for SUPPORT and CONFIDENCE**. Compared to both result for T\_SUPPORT=3 T\_CONFIDENCE=65 and T\_SUPPORT=10 T\_CONFIDENCE=80, we can find there are few false positives when T\_SUPPORT=10 T\_CONFIDENCE=80.

**Examples for false positives:**

**bug:** apr\_array\_push in ap\_file\_walk, pair: (apr\_array\_make, apr\_array\_push), support: 40, confidence: 80.00%

**bug:** ap\_merge\_per\_dir\_configs in ap\_fixup\_virtual\_hosts, pair: (ap\_merge\_per\_dir\_configs, apr\_array\_push), support: 3, confidence: 75.00%

The first bug belongs to reason 1, in 'ap\_file\_walk', it directly calls 'apr\_array\_push' but not directly calls 'apr\_array\_make'. When considering inter-procedural analysis, we can see 'ap\_file\_walk' calls 'prep\_walk\_cache' and 'apr\_array\_make' is called in 'prep\_walk\_cache'. That is to say, 'ap\_file\_walk' calls both 'apr\_array\_make' and 'apr\_array\_push'. So the first bug is false positive.

The second bug belongs to reason2, 'ap\_fixup\_virtual\_hosts' is used to setup all virtual hosts. 'ap\_merge\_per\_dir\_configs' is the function that run all of the modules merge per dir config functions. 'apr\_array\_push' is to add a new element to an array. 'apr\_array\_push' is not correlated with function 'ap\_merge\_per\_dir\_configs'. So the second bug is also false positive.

(c) Inter-Procedural Analysis.

1. Ways to inter-procedure analysis

After we stored the callMap which keeps caller function and its corresponding callee function set, we do a further step—using inter-procedure analysis to expand the original callMap to expandedCallMap ,which keeps all record in callMap and also added all callsets invoked by callee function sets in callMap.At this point,we came up with two solutions.

**Solution1**:callee function which is stored in callMap and invoke other functions will still be kept in the expandedCallMap with its callee set.

1. fetch every KEY in callMap whose data structure is a HashMap,then try to find its corresponding VALUE which is a callee function set,say C1,go to (2)
2. take every function in C1 out ,mark it as V1 and find its corresponding callee function set C2 in callMap,if C2 can be found as a KEY in callMap,go to (3)
3. creating a new set(S1) that stores C1 and C1’s callee set C2,and go to (4)
4. save every KEY in callMap and its corresponding new set S1 as a <KEY,VALUE> pair into the expandedCallMap

E.g:Originally we have Scope1() and Scope2() in CallMap,which can be expressed by Scope1(){A,B,Scope2()} ; Scope2(){C}.Then what we will store into expandedCallMap is Scope1(){A,B,scope2(),C} and Scope2(){}

**Solution2:**the main process is the same with Solution 2 ,the difference lies in that after we expand a callee(say B) function in callMap which called by caller(for example:A) ,we will delete B from VALUE set in expandedCallMap

1. fetch every KEY in callMap whose data structure is a HashMap,try to find its corresponding VALUE that is a callee function set,say C1,go to (2)
2. take every function in C1 out,mark it as V1 and find its corresponding callee function set C2 in callMap,if C2 can be found as a KEY in callMap,go to (3)
3. creating a new set(S1) that stores C1 and C2 but remove V1,function that invokes expanded callee Set C2
4. save every KEY in callMap and its corresponding new set S1 as a <KEY,VALUE> pair into the expandedCallMap

E.g:originally,we have Scope1() and Scope2() in the CallMap, Scope1(){A,B,Scope2()} Scope2(){C}.Then what we will store into expandedCallMap are Scope1(){A,B,C} and Scope2(){C},we will not store the function which invokes “C” into Scope1().

These two approaches can generate different expandedCallMap structure and lead to diverse bug result.For our team,we prefer the former one.we make this decision based on this situation:suppose Scope1() and Scope2() are marked as a pair based on former programming codes and we have a function,say Scope3(){Scope2(),A,B},if we expand our functions based on Solution2,then after the first expansion on Scope3(),we will get Scope3(){C,A,B},in the end, when we do bug detection,we will just consider pair match among A,B and C,which means we will ignore the pair<Scope1(),Scope2()>,this is truly worth and reasonably considering ,because for Scope3(){},it has a bug .It calls Scope2() without calling Scope1().To conclude,we think Solution2 will make us ignore some potential bugs.Thus,we choose Solution1 as our strategy.During the Bug detection period,first we will use CallMap to do the bug detection first and then we will double check,if some function in a pair is regarded as a bug in callMap ,but this pair exists in the expandedCallMap ,it will not regarded as a bug,by doing this,we call reduce false positive and it performs better than the situation when expanding level equals to 0.

2.Result of inter-procedure analysis

Since our group prefers Solution1,we will mainly focus on results and analysis of this solution.

|  |  |  |
| --- | --- | --- |
| Expanding Level | Support:3  Confidence:65% | Support:10  Confidence:8% |
| 0 | 205 | 34 |
| 1 | 165 | 30 |
| 2 | 156 | 30 |
| 3 | 156 | 30 |
| 4 | 156 | 30 |

As we can see from the table,as expanding level increases,the bug we detected becomes less,and when the expanding level increases to certain number,in this case ,it equals to 2,the bug number will become stable.

(d) Improving the Solutions.

**Motivation**：In this part considered both reducing the false positive and finding more bugs. We take the **order** of functions into consideration.

This is an example that could occur in the testing code.

|  |  |
| --- | --- |
| Caller | Callee |
| Scope1() | A() B() C() D() |
| Scope2() | A() B() D() |
| Scope3() | A() B() |
| Scope4() | B() D() A() |

In this case, function A() and function B() are both called in these four callers. There will be no bug for pair (A, B) if we use the automatic testing tool in part1(a). But as we can see in this case, A() is always called before B() except for Scope4(). We can consider A() as a function of opening the file and B() as a function of close the file. Now the sequence of two functions becomes more important. If the close-file function appears before open-file function then the mistake will be made and the program maybe crashed. So **more bugs** will be found when we consider the sequence of the functions.

Moreover, when we treat (A, B) and (B, A) differently, in the case above, the support value of (A, B) is 3, but the support value of (B, A) is only 1. Since we made T\_SUPPORT as 3 we will not consider pair (B, A). This will in some way help us **reduce the false positives**.

Another point we want to add is that: It is possible that lock and unlock function are appeared in different method. This should not be considered as a bug. But the implementation in part1 (a) is not taking care of this issue. Since it is being stated as a bug but in real it might be considered as a false positive. To minimize these false positives, what we can implement is that we can check if two functions have the same support, then they won't be considered as a bug. So for example, there is a lock function and there is an unlock function. If the support of lock function is equal to the support of the unlock function, this would mean that if one of them occurs alone, it will not be a bug. This will also help us reduce some false positives.

**Steps of solution are as follows:**

1. Read the input and set the threshold according to to inputs.
2. Parse the call graph and save the graph information into HashMap and HashSet.

In callMap, key is every caller and the values are the functions that called by caller.

In orderCallMap: key is every call and the values are callees sorted by the sequence that called by caller.

1. Save pairs that considered order to orderPairMap and save calleeMap.

The key of calleeMap saves all the callee and the values refer to callers.

1. Select pairs that satisfied the threshold. Put those pair as keys in orderPairSupport and put the support value as values in the same HashMap.
2. Calculate confidence and findbugs according to calleeMap and orderPairMap.

**Result and compare:**

We use test case 3 to compare two implementations in part 1A and this part. As for part 1A, when the T\_SUPPORT=3 and T\_CONFIDENCE=65 the total number of bugs is 205. In this part when we only consider the order of functions, the total number of bugs is 243. As we run verify.sh the result shows like this.



We can see from this, when considering order, we do find more bugs and reduce some false positives compared to the sample result of part 1A.

When we also consider whether the support value of two functions are equal, the total number of bugs is 191.