## 

Project – Part C

Software Engineering II

Spring 2017

**Student:**   
Ragnar Mikael Halldórsson

**Instructors:**Mohammad Hamdaqa  
Bjarni Kristján Leifsson

## The CallGraph class

The call graph is represented in code as the CallGraph class, which is a directed, weighted graph implemented as an adjacency matrix. The matrix is represented with a HashMap whose keys are caller nodes, and whose values is another HashMap whose keys are that caller’s callees. In turn, the values of the callee HashMap is the weight of this edge, which represents how many times this caller calls the callee. To illustrate, consider this call graph:



Where the weights of the edges represent the number of times a node (caller) calls another node (callee). Note that no nodes call N1, and N5 does not call any nodes. This graph would be represented as a CallGraph in the following way:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N2 | N3 | N4 | N5 |
| N1 | 1 | 2 |  |  |
| N2 |  |  | 2 |  |
| N3 | 1 |  | 3 |  |
| N4 |  |  |  | 2 |

Where each row represents a single caller, calling multiple callees. Empty fields can be represented as null; the caller HashMap contains no key for the callee, and thus the value is null.

Representing our call graph as an adjacency matrix using Hash Maps has the advantage that calculating the support for a single function is very fast (on the order of O(n) where n is the number of callers) as we only need to iterate through each caller and count the number of found keys for a callee. Calculating the support for a pair of functions, or indeed any n-tuple of functions, is also on the order of O(n), as we only need to check each caller if both (or all) callee keys are present.

Having the graph weighted also gives us the possibility of counting how many times a callee is called in total, or how many times a single caller calls a callee. This becomes important when we start doing inter-procedural analysis.

## Intra-procedural analysis

Our algorithm for intra-procedural analysis is quite simple. We start by calculating the support for each callee, and storing the result if the support for the function is greater than or equal to our support threshold. This is important for performance, because in this way we can limit the amount of function pairs we need to generate, and thus limit the number of function pairs we have to calculate the support for (if a single function is called n times, then it is impossible for the function to be called n+1 times as part of a pair). We then generate all pairs of functions from the set of functions with support greater than the threshold, calculate the support for each of those pairs and store the results.

After the support calculations are done, we iterate all the pairs of functions generated and calculate the confidence of that pair with each of its elements. If the confidence for either of the elements is over the threshold, we can assume that there is a bug for that pair. The next step is to find which function the bug occurs in, and for that we iterate over all the callers (scopes) and check if one element but not the other exists in the scope. If we find that one of the pairs is present but not the other, we have found a bug.

## Inter-procedural analysis

Due to the way the Intra-procedural analysis algorithm and the call graph is designed, extending it to include inter-procedural analysis should be trivial. Instead of reporting the bug as soon as one of the pair is not present in a scope, we get all the callees of that scope (caller) and iterate those, looking for the missing function call. Note that we must differentiate if the scope calls the missing function as part of the same pair that we detected had a bug; then most likely the bug does not lie in this scope and we must continue searching.

If we search all callees up to the specified depth and do not find the single missing function call, we can report this as a likely bug. However, if we do find the missing function call, we can pair the found element in the original node with the new found function call and can assume that this is not a bug.

## Finding and Explaining False Positives

There are many reasons why a bug finder will detect a “bug” that is indeed not a bug. These can be hard to avoid, as their causes are often unavoidable or very resource dependent to detect. Most notable of these reasons are when certain parameters go out of scope, such as a mutex being locked and passed on to another function, this would be detected as a bug since the mutex was not unlocked in the same scope as it was locked.

Another big reason for false positives is random pattern matching. In the httpd code we looked at we noticed a trend of these, a few hook registers and debug macros were called together in various functions and others they were not, in one case a mutex lock and lseek() was called as a pair and resulted in a reported bug, even though they have no relation to each other.

**The first false positive pair we found was (apr\_array\_make, apr\_array\_push).**

The bug that we are going to look closer at :

*bug: apr\_array\_push in ap\_copy\_method\_list, pair: (apr\_array\_make, apr\_array\_push), support: 40, confidence: 80.00%*

Ap\_copy\_method\_list is a function that takes in two lists: src and dest, then copies the contents of src into dest. Since the function takes in the src list, no new list is made with apr\_array\_make and only apr\_array\_push is used to copy between the two lists.

It’s safe to assume that this is a scope issue, the array will have been made elsewere. Therefore it’s a false positive.

There are in total 16 bugs connected to this pair.

**Second false positive pair we found was (apr\_array\_make, apr\_hook\_debug\_show)**

The bug that we are going to look closer into is

*bug: apr\_hook\_debug\_show in ap\_hook\_default\_port, pair: (apr\_array\_make, apr\_hook\_debug\_show), support: 28, confidence: 87.50%*

This is a classic example of coincidence false positive, there is no apparent connection between creating a new array (apr\_array\_make) and calling a debugger hook (apr\_hook\_debug\_show). Several other similar occurrences were found, all false positives.

There are in total 8 bugs connected to this pair.