Automatic Bug Detection

Software Engineering II

Spring 2017

Students:

Ragnar Mikael Halldórsson

Berglind Lilja Björnsdóttir

Baldur Már Pétursson

Njáll Hilmar Hilmarsson

Instructors:

Mohammad Hamdaqa

Bjarni Kristján Leifsson

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. For example a mutex being locked and passed on to another function, which then unlocks it. This would be detected as a bug since the mutex was not unlocked in the same scope as it was locked. These sorts of false positives can often be mitigated by Inter-Procedural analysis.

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. In some cases, the program might have to lock a mutex to access a shared resource and call lseek() on it, but the these functions being called without the other should not necessarily be regarded as a bug since in most cases this is the normal behavior.

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%

The function ap_copy_method_list 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 elsewhere. 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

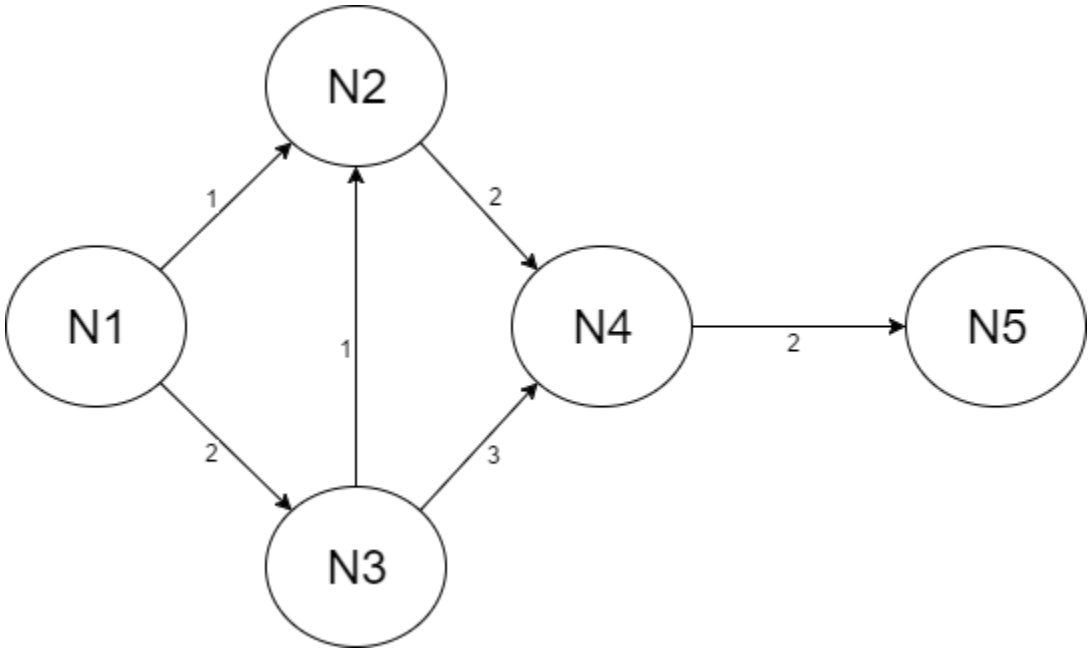
bug: apr_hook_debug_show in ap_hook_default_port, pair: (apr_array_make, apr_hook_debug_show), support: 28, confidence: 87.50%

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.

The CallGraph class

To understand our algorithms, you must understand the way we represent the call graph in our code. It is represented as the CallGraph class, which is a directed, weighted graph implemented as an adjacency matrix. The matrix is represented with a HashMap (<String, Hashmap>) whose keys are caller nodes and whose values is another HashMap (<String, Integer>). The inner HashMap's keys are that caller's callees and its values are the number of times the 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 an adjacency matrix 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. If a caller does not call a callee, then the inner HashMap will not contain the callee's key, and we can represent this field in the matrix as null. If a caller does call a callee, then the callee is present as a key in the inner HashMap, and the value for that key is the number of times the caller calls the callee.

Representing our call graph as an adjacency matrix using Hash Maps has the advantage that calculating the support for any n-tuple of function pairs is equally 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 callers that have all the keys we are looking for.

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 (part A)

We start by calculating the support for each callee (the number of callers that call this callee), and only 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.

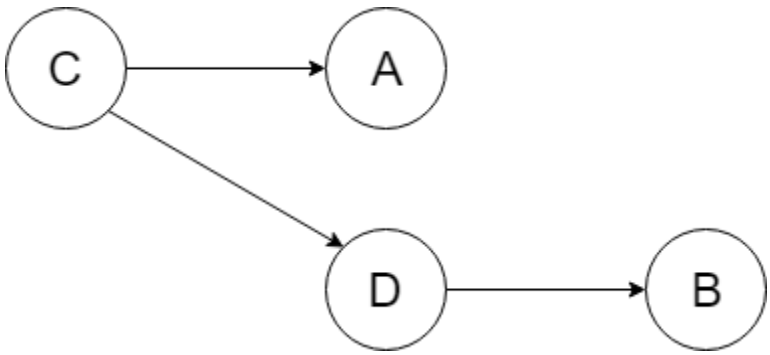
After the support calculations are done, we iterate all the pairs of functions generated and calculate the confidence of that pair with each of its elements. If the confidence for either of the elements is over the threshold, we can assume that there is a bug for that pair. The next step is to find which 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 (part C)

Due to the way the Intra-procedural analysis algorithm and the call graph is designed, extending it to include inter-procedural analysis is quite trivial. Instead of reporting the bug as soon as one of the pair is not present in a scope, we search for the missing function of the pair using breadth-first-search (it is most likely that the pair is found at shallow depths). If the missing function is found, we can pair it with the other function of the pair and eliminate the false positive. If the missing function is not found after searching the call graph to the specified depth, we can report this as a possible bug.

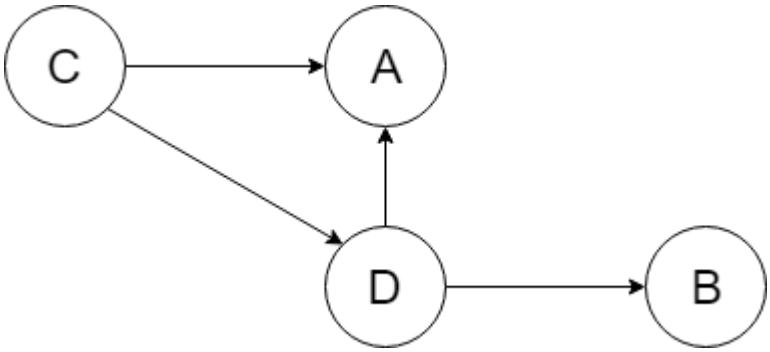
The only changes we made to the original algorithm was adding breadth-first-search with a specific goal to the reporting routine. If the BFS algorithm returns a node (which is the node that contains the missing function), we do not report this as a bug. However, if the BFS algorithm returns null, it did not find the missing function in the graph and we can report this as a possible bug.

The goal of the BFS algorithm is checking if the current node calls the target (the missing function) node more often than the other (the single function of the pair) node. For example, consider the following simple call graph



Here, C calls A and D, and D calls B. In this case, ({A,B}, {A}) is reported as a possible bug in scope C, since A is called without its partner B. We now check scope D for calls to B, the missing element. Since D calls B more times than A, we have found A's partner and can pair them together, and we identify this report as a false positive and don't report it.

However, suppose we change the call graph slightly by making D call A as well as B.



In this case, C calling A without B will be reported as a bug. Since D calls A and B as a pair, we cannot pair the single A called in C with any B, and we can report this as a bug. The BFS algorithm goal will detect this, since in scope D, B is not called more often than A (the number of calls is equal).

What we found

After running the modified bug detector, we were able to reduce the number of detected bugs in test case 3 from 205 to 157, eliminating 48 false positives. With depth at only 1, we get 168 bugs reported, with depth at 2, there are 159 bugs reported and at depth 3 we get to 157. All of these tests were done with default support and confidence thresholds (3 and 65)

By tweaking the support and confidence thresholds and the inter-procedural depth we were able to filter out more false positives. Raising the confidence level to 90% with support 3 yielded 6 reported bugs, all of them concerning the pair *(apr_thread_mutex_lock, apr_thread_mutex_unlock)*, which do look like likely candidates for bugs.