Links:
stl map reference: http://www.cplusplus.com/reference/map/map/
stl map time complexity:
http://stackoverflow.com/questions/222658/multiset-map-and-hash-map-complexity
trie: http://en.wikipedia.org/wiki/Trie
radix tree: http://en.wikipedia.org/wiki/Radix_tree
use hash_set with strings in gcc:
http://www.moosechips.com/2008/10/using-gcc-c-hash-classes-with-strings/
hash_set deprecated: http://gcc.gnu.org/gcc-4.3/changes.html

Data Structures:

After having a brief look over the options, it looks like the best data structure to use in C++ is the map in the stl. Now there is a catch, being that the underlying implementation doesn't seem to be a hash table, instead it seems to be some sort of tree structure that gets O(log(n)) for its operations. However, the TA seem to think that anything that ended up making the overall algorithm run in less than n^3 time would be sufficient, and we can use this to start out and optimize if we really want to.

Algorithm:

I propose that we use a three-part algorithm as follows:
1. Parse input and build data structures
In this stage, the program will build the data structures necessary for the next steps.

- A map for retrieving integer keys for functions given the function name as a string. (As we go through, we can increment a counter to use as the ID of the next new function we find, and then store its id in the map. If we want to improve the efficiency of this data structure, we could build a custom trie or radix tree).
- A map for storing support values. We could use the ID of the function as the key, and an object containing the support value of that function and a hash or vector containing the support values for pairs (A,B) involving that function (we can always store the support value for a pair of functions in the object corresponding to the function with the lowest ID number, to avoid duplication).
- A vector for storing data on each top-level function, specifically which functions are called inside of it. We need to make sure to store the top-level function's name.

2. Filter for support and confidence

In this stage, we can find the function pairs (A, B) such that support ({A, B})/support ({A}) > T_CONFIDENCE and support({A,B}) > T_SUPPORT. We can build a list of functions B that should always be called whenever a function A is called. Processing this now will reduce the number of comparisons would be to make later on.

3. Find bug instances

In this stage, the program will look through the top-level function data structure built previously, examine each function call, and find missing function pairs (by first looking at the functions that we think must be called with each function, and eliminating ones that are actually in the same scope).

As long as we're careful with their data structures, and we don't end up having too many massive functions in the input (which the numbers don't suggest is likely) we should come out okay. It might be a good idea to generate a large sample test case so we can time our execution once a basic implementation is in place, and then see if there are any bottlenecks we'll need to fix.