Efficient use of memory by reducing size of AST dumps in cross file analysis by clang static analyzer

Clang SA works well with function call within a translation unit. When execution reaches a function implemented in another TU, analyzer skips analysis of called function definition. For handling cross file bugs, the CTU analysis feature was developed (Mostly by Ericsson people)[2]. The CTU model consists of two passes. The first pass dumps AST for all translation unit, creates a function map to corresponding AST. In the second pass when TU external function is reached during the analysis, the location of the definition of that function is looked up in the function definition index and the definition is imported from the containing AST binary into the caller's context using the ASTImporter class. During the analysis, we need to store the dumped ASTs temporarily. For a large code base this can be a problem and we have seen it practically where the code analysis stops due to memory shortage. Not only in CTU analysis but also in general case clang SA analysis reducing size of ASTs can also lead to scaling of clang SA to larger code bases. We are basically using two methods:-

1) Using Outlining method[3] on the source code to find out AST that share common factors or sub trees. We throw away those ASTs that won't match any other AST, thereby reducing number of ASTs dumped in memory.

2) Tree prunning technique to keep only those parts of tree necessary for cross translation unit analysis and eliminating the rest to decrease the size of tree. Finding necessary part of tree can be done by finding the dependency path from the exploded graph where instructions dependent on the function call/execution will be present. A thing to note here is that prunning of only those branches whose no child is a function call should be done.

The CTU model can be given by this:-

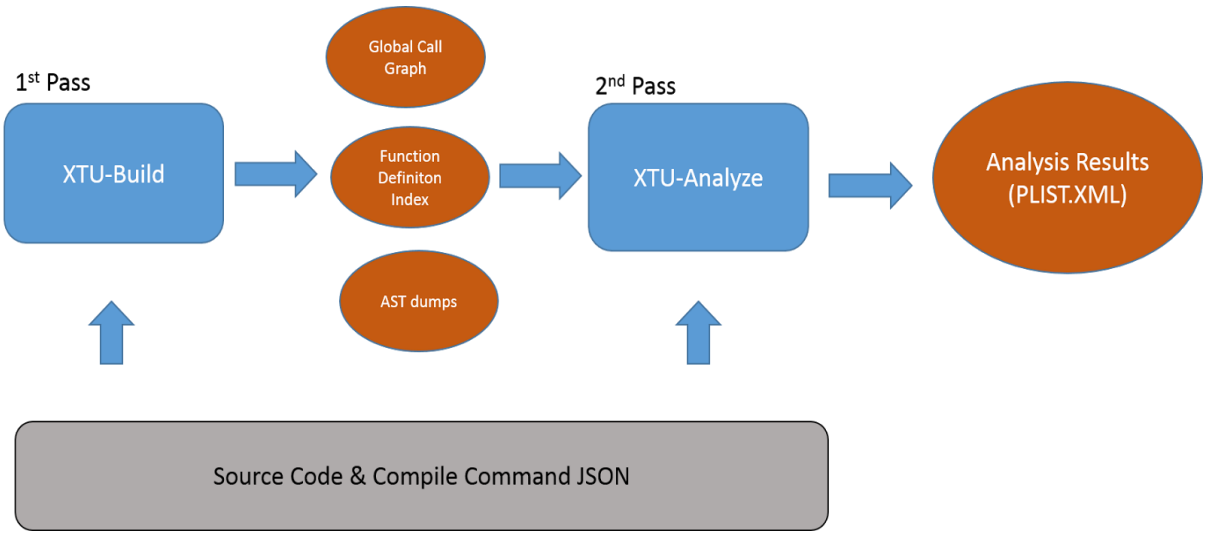

Fig.1 [1]

In this model if u look at Fig.1 in the first pass while dumping AST in memory the outlining algorithm can be applied to reduce the memory occupied by AST dumps. The outlining algorithm can be summarized by the following steps:-

To reduce the size of the tree, we can eliminate ASTs that won't match anything else in a first pass (that is, if you don't care about matching sub trees anyway). A hashing scheme that would store pointers to trees. Two trees would be in the same bucket if they could possibly match. They would be in different buckets if they definitely cannot match (like a bloom filter kind of setup). Then we can flatten the trees in each bucket, use the outlining technique there, and then end up with a factorization that way.

(1) Construct every AST.
(2) Say two ASTs "could be equal" if they are isomorphic to each other.
(3) Bucket each ASTs based off the "could be equal" scheme.
(4) For each bucket with more than one entry, flatten out the ASTs and run the outlining technique on the trees. At the end of each iteration, throw out the suffix tree built to handle the bucket.

Here the main point to note here is that we are eliminating AST which won't match anything, this reduces a large number of ASTs in memory. We are using a fast subtree isomorphism algorithm for matching ASTs which takes O((k^(1.5)/logk) n) , where k and n are the numbers of nodes in two ASTs.

For Tree pruning we are using the exploded graph concept to find the execution path when an externally defined function is called, focussing only on the variables or instructions which are affected as a result of that function call. We find like these all paths where an external function call is there, we keep these paths/branches in AST and eliminate all other branches of AST thereby reducing size of AST.

References

[1]http://llvm.org/devmtg/2017-03//assets/slides/cross_translation_unit_analysis_in_clang_static_analyzer.pdf

[2] https://www.youtube.com/watch?v=7AWgaqvFsgs

[3] https://www.youtube.com/watch?v=yorld-WSOeU&t=1060s