Improved Loop Execution Modeling in the Clang Static Analyzer

Name1 [*]
Affiliation1
Email1

Name2 Name3 [†]
Affiliation2/3
Email2/3

Abstract

The LLVM Clang Static Analyzer is a source code analysis tool to find bugs in C, C++, and Objective-C programs using symbolic execution, i.e. it simulates the possible execution paths of the code. Currently the simulation of the loops is somewhat naive (but efficient), unrolling the loops for a predefined constant time. However, this approach can result in loss of coverage in various cases. This study aims to introduce two alternative approaches which can extend the current one and can be applied simultaneously: (1) with applied heuristics we determine loops worth to fully unroll (2) we use a widening mechanism to simulate arbitrary step of iteration. These methods were evaluated on numerous open source projects, and were proved to increase coverage in most of the cases. This work also laid the infrastructure for future loop modeling improvements.

Keywords keyword1, keyword2 TODO

1. Introduction

The Clang Static Analyzer finds bugs by performing a symbolic execution on the code. During symbolic execution, the program is being interpreted, on a function-by-function basis, without any knowledge about the runtime environment. It builds up and traverses an inner model of the execution paths, called ExplodedGraph, for every analyzed function. A node of this graph (called ExplodedNode) contains a ProgramPoint (which determines the location) and a State (which contains the known information at that point). Its paths from the root to the leaves are modeling the different execution paths of the analyzed function. Whenever the execution encounters a branch, a corresponding branch will be created in the ExplodedGraph during the simulated interpretation. Hence, branches lead to an exponential number of ExplodedNodes. This combinatorial explosion is handled in the Static Analyzer by stopping the analysis when given conditions are fulfilled. Ceasing the analyzation process may cause loss of potential true positive results, but it is indispensable for maintaining a reasonable resource consumption regarding the memory and CPU usage. These conditions are modeled by the concept of budget. The budget is a collection of limitations on the shape of the ExplodedGraph. These limitations include:

1. The maximum number of traversed nodes in the ExplodedGraph. If this number is reached then the analysis of the simulated function stops.
2. The size of the simulated call stack. When a function call is reached then the analysis continues in its body as if it was inlined to the place of call (interprocedural). There are several heuristics that may control the behavior of inlining process. For example the too large functions are not inlined at all, and the really short functions are not counted in the size of call stack.
3. The number of times a function is inlined. The idea behind this constraint is that the more a function is analyzed, the less likely it is that a bug will appear in it. If this number is reached then that function will not be inlined again in this ExplodedGraph.
4. The number of times a basic block is processed during the analysis. This constraint limits the number of loop iterations. When this threshold is reached the currently analyzed execution path will be aborted. The budget expression can be used in two ways. Sometimes it means the collection of the limitations above, sometimes it refers to one of these limitations. This will always be distinguishable from the context.

[*] with optional author note

[†] with optional author note

2. Motivation

As already mentioned in the introduction, the analyzer handles loops quite simply in the current state. More precisely, it unrolls them 4 times by default and then cuts the analysis of the path where the loop would have been unrolled more than 4 times.

Loss in code coverage is one of the problems with this approach to loop modeling. Specifically, in cases where the loop is statically known to make more than 4 steps, the analyzer do not analyze the code following the loop. Thus, the naive loop handling (described above) could lead to entirely unchecked code. Listing 1 shows a small example for that.

```
void foo() {
  int arr[6];
  for (int i = 0; i < 6; i++){
    arr[i] = i;
  }
  /*rest of the function*/
}
```

Listing 1. Since the loop condition is known at every iteration, the analyzer will not check the 'rest of the function' part in the current state.

According to the budget rule concerning the basic block visit number, the analysis of the loop stops in the fourth iteration even if the loop condition is simple enough to see that unrolling the whole loop would not be too much extra work relatively. Running out of the budget implies (in this case) that the rest of the function body remains unanalyzed, which may lead to not finding potential bugs. Another problem can be seen on Listing 2:

```
int num();
void foo()
{
  int n = 0;
  for (int i = 0; i < num(); ++i) {
    ++n;
  }
  /*rest of the function, n < 4 */
}
```

Listing 2. The loop condition is unknown but the analyzer will not generate a simulation path where $n \geq 4$ (which can result coverage loss).

This code fragment results an analysis which keep track the values of `n` and `i` variables (this information is stored in the State). In every iteration of the loop the values are updated accordingly. Note that updating the State means a new node insertion in the ExplodedGraph with the new values. Since the body of the `num()` function is not known, the analyzer can not find out the its return value. Thus, it is considered as unknown. This circumstance makes the graph split to two branches. The first one belongs to the symbolic execution of the loop body assuming that the loop condition is true. The other one simulates the case where the condition is false and the execution continues after the loop. This process is done for every loop iteration, however, the 4th time assuming the condition is true, the path will be cut short according to the budget rule. Although the analyzer generates paths to simulate the code after the loop in the above described case, yet the value of variable `n` will be always less than 4 on these paths and the rest of the function will only be checked assuming this constraint. This can result in coverage loss as well, since the analyzer will ignore the paths where n is more than 4.

3. Proposed Solution

In this section we present two solutions to resolve the previously mentioned limitations on symbolic execution of loops in the Clang Static Analyzer. It is important to note that these enhancements are incremental in the sense that on examples which are too complex to handle at the moment, we fall back to the original method. For sake of simplicity in the following examples a "division by zero" will illustrate the bug we intend to find.

3.1 Loop Unrolling Heuristics

Loop unrolling means we have identified heuristics and patterns (such as loops with small number of branches and small known static bound) in order to find specific loops which are worth to be completely unrolled. This idea is inspired by the following example:

```
void foo() {
  for (int i = 0; i < 6; i++){
    /*simple loop which does not
     change 'i' or split the state*/
  }
  int k = 0;
  int l = 2/k; // Division by zero
}
```

Listing 3. Complete unrolling of the loop makes possible to find the division by zero error.

In the current solution a loop has to fulfill the following conditions in order to be unrolled:

1. The loop condition should be arithmetically compare a variable – which is known at the beginning of the loop – to a literal (like: `i < 6` or `6` $\geq$ `i`)
2. The loop should only change once the loop variable in its body and the difference needs to be constant. (This way we can estimate the maximum number of steps.)
3. The estimated number of steps should be less than 128. (Still do not want to simulate loops which takes thousands of steps because they could single handedly exhaust the budget.)
4. The loop must not generate new branches or use `goto` statements.

Using this method we can successfully find the bug on the above example.

3.2 Loop Widening

The final aim of widening is quite the same as the unrolling, to increase the coverage of the analysis. However, it reaches it in a very different way. During widening the analyzer simulates the execution of an arbitrary number of iterations. There is already a solution which reaches this behavior by discarding all of the known information before the last step of the loop. So, the analyzer creates the paths for the first 3 steps and simulate them as usual, but the widening (means the invalidating) happens before the 4th step in order to not lose the first precise simulation branches. This way the coverage will be increased but can easily result in too much false positives. Consider the example on Listing 4.

```
int num();
void foo() {
  bool b = true;
  for (int i = 0; i < num(); ++i) {
    /*does not changes 'b'*/
  }
  int n = 0;
  if (b)
    n++;
  n = 1/n; // False positive:
           // Division by zero
  }
}
```

Listing 4. Invalidating every known information (even the ones which are not modified by the loop) can easily result false positives.

In this case the analyzer will create and check that impossible path where the variable `b` is false, so `n` is not incremented and lead into a division by zero error. Since this execution path would never be performed while running the analyzed program, it is considered a false positive. My aim was to give a more precise approach for widening.

The principles are that we try to continue the analysis after the block visiting budget is exhausted but invalidate the information only on the variables which are possibly modified by the loop. For this I developed a solution which checks every possible way in which a variable can be modified in the loop. Then it evaluates these cases and if it encounters a modified variable which cannot be handled by the invalidation process (e.g.: a pointer variable) then the loop will not be widened and we return to the conservative method. This mechanism ensures that we do not create nodes containing invalid states. This approach helps us to cover cases and find bugs like the one illustrated on Listing 5, and still not report false positives which are represented on Listing 4.

```
int num();
void foo() {
  int n = 0;
  for (int i = 0; i < num(); ++i) {
    ++n;
  }
  if (n > 4) {
    int k = 0;
    k = 1/k;  // Division by zero error
  }
}
```

Listing 5. Invalidating the information on only the possible changed variables can result higher coverage (while limiting the number of the found false positives).

We find the bug since we invalidate the known informations on variable `n` (and `i` as well). This cause the analyzer to create a branch where it checks the body of the `if` statement and finds the bug. However, this solution has its own limitations when dealing with nested loops. Consider the case on Listing 6.

```
int num();
void foo() {
  int n = 0;
  for (int i = 0; i < num(); ++i) {
   ++n;
   for (int j = 0; j < 4; ++j) {
     /*body that does not change n*/
   }
  }
  /*rest of the function, n <= 1 */
}
```

Listing 6. The naive widening method does not handle well the nested loops. In this example the outer loop will not be widened.

In this scenario, when the analyzer first step into the outer loop (so, assumes that `i < num()` is true) and encounter the inner loop, then it consumes its (own) block visiting budget. (This implies that it will be widened, although in this case it means that only the inner loop counter (`j`) information is discarded.) After that we move on to the next iteration, and assume that we are on the path where the outer loop condition is true again. Because we already exhausted the budget in the previous iteration, the next visit of the first basic block of the inner loop (the condition) means that this path will be completely cut off and not analyzed. This results that the outer loop will not reach the step number where it would been widened. Furthermore, the outer loop will not even reach the 3rd step and the 2nd is stopped at in its body as well (as described above). This causes the problem, that even though we use the loop widening method, we will analyze the rest of the function with the assumption `n <= 1`.

In order to deal with the above described nested loop problem, I have implemented a replay mechanism. This means that whenever we encounter an inner loop which already consumed its budget, we replay the analysis process of the current step of the outer loop but performing a widening first. This ensures the creation of a path which assumes that the condition is false and simulates the execution after the loop while the possibly changed information are discarded. This way the analyzer will not exclude some feasible path because of the simple loop handling which solves the problem.

Another note to the widening process that it makes sense to analyze the branch where the condition is true with the widened State as well. The example on Listing 7 shows a case where this is useful.

```
int num();
void foo() {
 int n = 0;
 int i;
 for (i = 0; i < num(); ++i) {
   if (i == 7) {
     break;
   }
   for (int j = 0; j < 4; ++j) {/* */}
 }
 int n = 1 / (7 - k); // Possible division
    by zero
}
```

Listing 7. The replay mechanism successfully helps us to find the possible error the outer loop.

This way the analyzer will produce a path where the value of `i` is known to be 7, so it will be able find the possible division by zero error.

4. Evaluation

The effect of the described loop modeling approaches were measured on various C/C++ open source project.

4.1 Loop Unrolling

4.2 Loop Widening

5. Conclusion

6. Future work

Acknowledgments

Acknowledgments, if needed.

References

P. Q. Smith, and X. Y. Jones. ...reference text...