Improved Loop Execution Modeling in Clang Static Analyzer

GSoC 2017 Final Report

Péter Szécsi

1. Motivation and aims

The Clang Static Analyzer is a source code analysis tool that finds bugs in C, C++, and Objective-C programs. In order to find bugs the tool performs a symbolic execution on the code. During symbolic execution, the program is being interpreted, without any knowledge about the runtime environment. It builds up and traverses an inner model of the executions paths, called ExplodedGraph. Its paths from the root to the leaves are modeling the different execution paths of the analyzed function. An ExplodedNode contains a ProgramPoint (which determines the location) and a State (which contains the known information at that point).
In the current state the analyzer handles loops quite simply. 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.

One (strong) motivation can be that there were already questions on this behavior in the cfe-dev mailing list and even requests on having the option to modify it. (eg. mail).
Loss in code coverage is one of the problems with this approach to loop modeling. Specifically, in cases where the loop bound is statically known to be greater than 4, the analyzer often did not analyze the code following the loop. Thus, the naive loop handling (described above) could lead to unchecked code. Here is a small example for that:

```
void f() {
  int a[6];
  for (int i = 0; i < 6; i++){
   a[i] = i+2;
  }
  //complex body
}
```

In the above example, because the loop bound is a statically known value, the analysis will be stopped at the 4th iteration of the loop and the “complex body” after the loop will not be analyzed.

2. The Working Process

My work on improving the loop modeling can be primarily divided into 2 tasks:

1. Loop Unrolling - Work up heuristics and AST 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. All patches for this work have been reviewed and committed to the clang repository which are the following:
    - The initial implementation: D34260
    - Changed the CFG of the clang compiler to represent loops more directly: D35668 and D35670
    - Update the feature to use the LoopExit element: D35684
    - Evaluated the option on different C/C++ open source projects, and measured performance as well as coverage to choose the best defaults.
    - New heuristics and fixes added in: D36962, D37103, and D37181
2. The Loop Unrolling work laid the infrastructure for future improvements in loop modeling. In the last part of the summer, I started investigating approaches to Loop Widening, where 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. My aim was to give a more precise solution/approach for widening. I submitted the following patches for review:
    - The initial patch which only invalidates the possibly changed variables but does not handle every possible case. In this scenario if we encounter a not handled statement we decide not to widen the loop. D36690
    - In order to handle nested loops better, the analyzer tries to reanalyze the loops with a widened state which execution was aborted due to visiting the same block too many times on a given path. D37108

2.1 Loop Unrolling:

An initial implementation of the unrolling was added in D34260 (landed).
One of the main challenges was that the Static Analyzer core was not prepared for handling loops more precisely. That means that the decisions were made based on purely the encountered CFG blocks and their terminator statements.
So in the initial patch I tracked the loop statement in the State. That means that for every maxBlockVisitOnPath counter check I had to collect the corresponding CFG blocks to every stored loop statement. That was not really a scalable method. Also we could not clean up the State since we could not be sure when the loop is left (this came from the problem defined above).

So for a more efficient solution I had to change the core. Because of the lexical nature of the problem it was enough to add a LoopExit element to the CFG and then handle these in the core itself. Note: It is not perfect so it is possible that on certain paths we don't encounter the LoopExit element even if we stepped into one. (This was done in patches D35668 (landed) and D35670 (landed).)

The feature allowed me to always track a so called LoopStack in the State. This stack indicates which loop statement bodies are being simulated at a given point of the analysis.
So for the LoopUnrolling it was enough to mark loops as Unrolled in the LoopStack and then

always check if we are in an unrolled loop or not. (D35684 (landed))

The other challenge was to find the optimal heuristics for deciding which loops to unroll.
At the end of the project a loop has to fulfill the following requirements to be considered for being completely unrolled:

- Currently only for-loops can be considered.
- The bound has to be an integer literal. (eg. i < 5, i >= MACRO_TO_LITERAL)
- The counter variable (i) has not escaped before/in the body of the loop and changed only in the increment statement corresponding to the loop. It also has to be initialized by a literal in the corresponding initStmt.
- Does not contain goto, switch and returnStmt.
- Whenever an unrolled loop splits the state (creates more branches), we consider it as a “normal” (not-unrolled) loop in the remaining part of the analysis. (D36962 (landed))
- Only loops which are known to take at most 128 steps will be unrolled. (D37181 (landed))

Note: some fixes and refactoring were sent in patch D37103 (landed).

2.2 Loop Widening:

The final aim of Loop Widening is quite the same as the unrolling, to increase the coverage of the analysis. However, it reaches it in a very different way. The principles are that we try to continue the analysis after the maxBlockVisitOnPath counter is exceeded but invalidate the information only on the variables which can be modified by the loop. The first version of this feature was developed years ago (in patch D12358 (not made by me)). This version invalidates every program state information after a loop and continues the analysis that way. The coverage of the analysis will be increased by that but this method can easily cause too much false positives in some cases.
For this I developed a solution which checks (or at least intends to check) 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. This mechanism ensures that we do not create nodes containing invalid states. (sent in D36690 (on review)) Sean (the author of the first version) suggested to replace the functionality with this second version instead of having two different widening options which could be confusing.
In order to handle nested loops better I used the same concept as the unrolling feature. So, tracking a LoopStack can be useful for this purpose. (D37108 (on review))
From the original proposal, there is one point I have not completed and remained for future work:

- Invalidate only the memory regions transitively reachable via the locals, in our example, it will be “a” and everything reachable from it. For example, if there was a statement “int *b = a”, both regions pointed to by “b” and “a” will be invalidated. Note, that here we rely on the aliasing relations already captured by the RegionStore class in the static analyzer core to determine what needs to be invalidated.

To summarize the point, this means that the pointer operation handling should be enhanced in case of widening.

3. Usage

The Loop Unrolling feature is hidden behind the flag ‘unroll-loops’ at the time. So in order to use it the user has to pass the ‘-analyzer-config unroll-loops=true’ option to the analyzer. However, it is considered for turning on by default soon. All of the above mentioned Loop Unrolling patches were accepted and already merged to the trunk. It can be downloaded and builded as it is explained in the Clang Getting Started page.

The Loop Widening feature is hidden behind the flag ‘widen-loops’ at the time. So in order to use it the user had to pass the ‘-analyzer-config widen-loops=true’ option to the analyzer. Since the patches on widening are still under review it means that the user has to get the source code (see: Getting Started page) and apply the patches. The patches can be downloaded from Phabricator (D36690 and D37108) by choosing the “**Download Raw Diff**” option.

4. Results

The loop unrolling results were summarized and sent to the cfe-dev mailing list (http://lists.llvm.org/pipermail/cfe-dev/2017-August/055210.html).
Measurements were run on different open source projects.

Time measurements (second):

	curl	libpng	vim	bitcoin	ffmpeg	xerces	LLVM
Normal	52.63	62.87	191.92	208.2	526.21	229.94	5044.62
LoopUnrolling	51.95	58.91	193.28	200.7	543.86	225.86	4838.26

Coverage measurements (% of reachable basic blocks statistics):

	curl	libpng	vim	bitcoin	ffmpeg	xerces	LLVM
Normal	58.05	55.91	51.13	68.58	49.78	74.86	71.15
LoopUnrolling	69.14	56.04	51.53	68.7	52.46	74.91	71.13

Based on these results I would suggest using the loop unrolling feature by default.

The widening results and the question about changing the functionality of the flag ‘loop-widening’ was sent to the cfe-dev as well: link.
The measurements produced the following results:
Coverage measurements (% of reachable basic blocks statistics):

	curl	libpng	vim	bitcoin	ffmpeg	xerces
Normal	58.05	55.07	51.12	69.37	49.78	74.86
Widen	75.8	56.69	51.54	72.06	65.63	76.06

Widen_GSoC	70.7	55.92	51.77	69.67	59.53	75.04

The number of founded bugs:

	curl	libpng	vim	bitcoin	ffmpeg	xerces
Normal	35	32	81	9	375	52
Widen	35	36	94	12	456	57
Widen_GSoC	27	34	84	10	369	51

Although Widen_GSoC has a small observable coverage loss (comparing to Widen) it's offset by the number of the false positives not discovered.
In conclusion it would be beneficial to replace the current implementation of the 'widen-loops' option.

5. Acknowledgement

I would like to thank my mentors Anna Zaks, Artem Dergachev, and Devin Coughlin for supporting me and giving me helpful reviews. Furthermore, I'd like to thank Sean Eveson and Gábor Horváth for their valuable comments on the patches.