Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages
And Compilers

Improved symbolic execution loop modeling for C like languages

Supervisors:

Zoltán Porkoláb
Associate professor

Gábor Horváth
PhD student

Author:

Péter Szécsi
Computer Science MSc

This study was supported by the ÚNKP-17-2 New National Excellence Program of the Hungarian Ministry of Human Capacities.

Budapest, 2018

Contents

1 Introduction

During software development it is natural to make mistakes. Consequently, writing various test cases is required in order to validate the behavior of the program. In addition to the costs of test writing, it is possible that the developers fail to cover all possible critical cases. Furthermore, test writing and running often happens later than code development, but the costs of error correction increases proportionally to elapsed time [Boehm and Basili, 2001]. This proves that testing alone is not necessarily sufficient to ensure code quality.

The static analysis tools offer a different approach for code validation [Michael and Robert, 2009] [Bessey et al., 2010], moreover, they can potentially check for some characteristics of the code – which cannot be verified by testing – e.g. the adherence to conventions.

These tools are given the source code as input, using which they build a model and make inferences based on that. The efficiency, accuracy, and runtime of the analysis changes depending on the complexity of the model. The final output is a list of bug reports. However bugs are used quite liberally here; they do not necessarily imply program malfunctions, as optimisations or code convention nits are also included. Code chunks deemed fragile or to have potential portability problems are listed too, so there can be a wide variety of possible outputs. This document focuses on static analysis primarily as a method of uncovering bugs.

Unfortunately, it is impossible to detect every bug only by using static analysis [Rice, 1953]. Static analyzer tools might not be able to discover some bugs (these are called false negatives) or report correct code snippets as incorrect (false positives, see Fig 1). In the industry the goal of these tools is to keep the ratio of the false positive reports low while still be able to find real bugs.

	True error	Non-error
Reported error	True Positive	False Positive
No error report	False Negative	True Negative

Figure 1: The different types of the static analysis results.

Its important to note that another advantage of the static analyzer tools (against test cases) that the analysis happens without executing the source code, so it can be used without the running environment. Moreover, the code coverage of the analysis does not depends on the already written test case set and even can find bugs which is not covered by test cases e.g. unexpected corner cases on the input variables.

The LLVM Clang Static Analyzer is a source code analysis tool which aims to find bugs in C, C++, and Objective-C programs using - one of the most precise analysis methods - symbolic execution [King, 1975] [Hampapuram et al., 2005], i.e.

it simulates the possible execution paths of 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 each analyzed function. This method has exponential runtime and space complexity in the number of branches.

2 The Clang Static Analyzer

Clang is an open-source compiler, which is based on the *LLVM* project [**?**]. This is a quickly improving project, supported by various companies including e.g. Google, Apple, Samsung, Ericsson, etc. Moreover it is becoming more and more popular in the circle of compilers. Since Clang is built outstandingly modular, it is not simply a compiler but a complete framework for creating developer tools on C/C++/Objective-C languages. Its API is well documented and the software is well tested due to the wide user base.

The Clang Static Analyzer (CSA) – as it is indicated by its name – build around the Clang compiler. It is one of the most quickly developing open source static analysis tool. In this section the principles and the relevant implementation details of the CSA will be presented.

2.1 Symbolic execution

The CSA uses the symbolic execution static analysis algorithm which means it simulates the possible execution paths and tries make conclusions. It basically interprets the source code but instead of using the actual semantics, it defines an abstract one and models the state according to the defined one.

Unknown values is represented by symbols/symbolic values. The CSA makes constraints on these symbolic values during the analysis of the different paths. Among others, these constraints came from the condition on which a path may bifurcate. There is a constraint solver module in the analyzer which handles these and tries to deduce the conflicts (exclude the non-feasible paths). This is important since a bug found on a non-feasible path is considered as a false positive. The analysis stops if all of the execution paths are simulated and checked or it reaches a predefined limit.

The memory is represented by a hierarchic memory region system [**?**], the correspondence between symbolic values and memory regions is tracked. The state of a given program point contains this relation of bindings as well.

The analysis of the different execution paths can be done in any order, however, the evaluation of an expression (on a given path) must happen according to the language standard. During the simulation of the expressions the analyzer models the state of the program, means that one expression/instruction will create a transition between two states. A symbolic state can be viewed as a set of concrete program states, moreover, a simulated path is basically an analysis of these symbolic states in a well defined order.

In order to be able to perform this process efficiently, the analyzer creates an inner model of the execution paths, called ExplodedGraph [**?**]. The nodes of the ExplodedGraph (called ExplodedNodes) are pairs, containing a symbolic program state

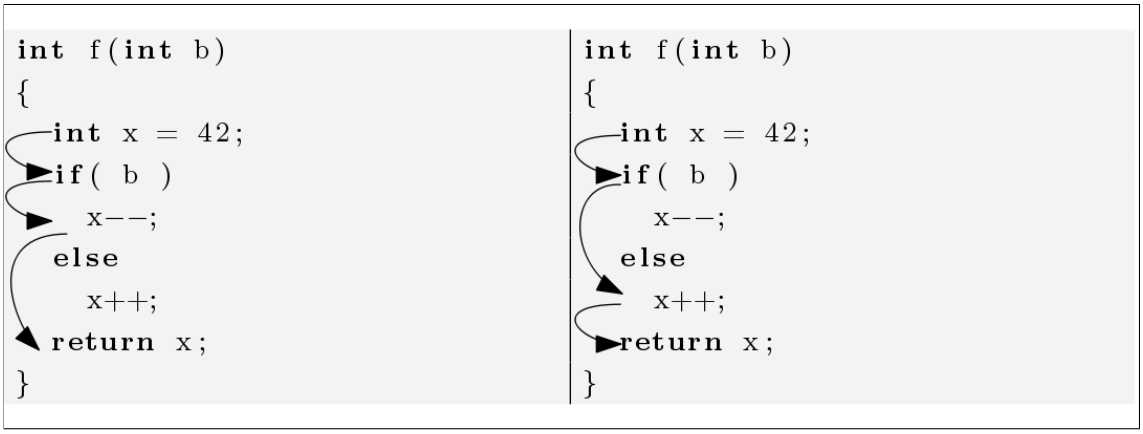

Figure 2: The different execution paths of a simple function

and a program point. The program point determines the code location where the symbolic execution is in the source text. The directed edge between ExplodedNodes determines the sequencing.

Whenever the CSA encounters a branch in the symbolic execution, it creates new branches in the ExplodedGraph (see Fig 3). The paths from the root to the leaves represents the analyzed paths. This means, that the number of leaves in the ExplodedGraph (and so the size of the graph) is exponential with respect to the branches in the execution. That results us a complex, precise but resource consuming method.

The ExplodedGraph is a simple directed acyclic graph. Whenever the analysis reachesan ExplodedNode which was already visited before, then the simulation will be stopped since the CSA would do the same (deterministic) reasoning what was already done. Note, that in case of concrete program states this would lead to an infinite loop, on the symbolic level this does not stand, since a symbolic program state represents a set of real program states.

In practice, the CSA constructs ExplodedNodes which does not represent any concrete transition between nodes but just help the simulation e.g. at some points the analyzer cleans up the stored but not necessary symbols and so it introduces a new node in order to smoothly fit this procedure into the flow of the analysis.

According to the properties of the model, it can be declared, that the symbolic execution is a path sensitive algorithm, i.e. reaching any program state the CSA knows the whole path leading to the simulated point. Thereby the displaying of the bugs shows not only the location of the bug but the corresponding execution path as well. This productively helps the programmer to understand the problem more easily, even spending less time on fixing the bug.

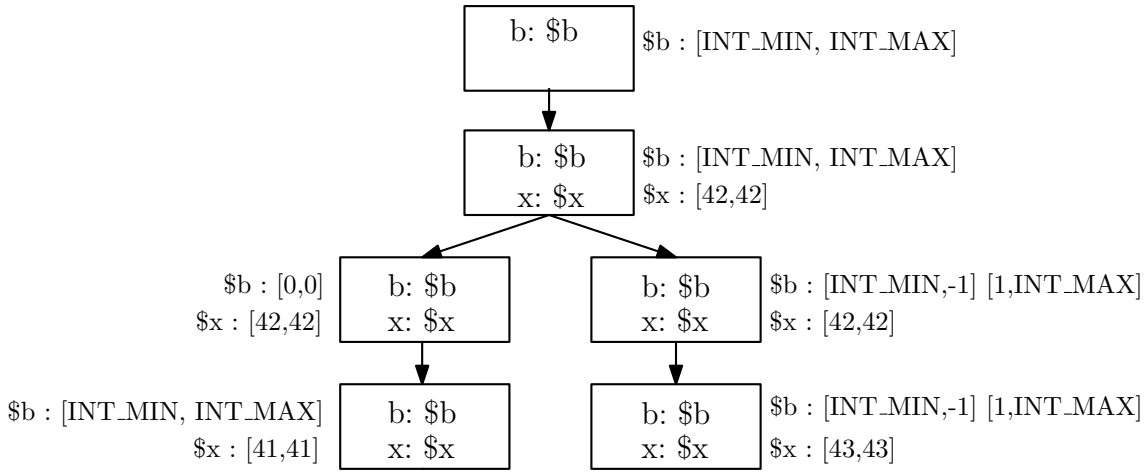

Figure 3: The corresponding ExplodedGraph to the function presented on Fig 2. The constraints made on the symbolic values are shown next to the ExplodedNodes it belongs.

2.2 Implementation details

2.2.1 The process of the analysis

The CSA works on a translation unit, analyzing its content function by function. So the the inner model is constructed for an analysis of a function and its states are represented in the ExplodedGraph.

Whenever – during the simulation of a given path – the CSA encounters a branch in the execution, it splits the states creating more children for the actually simulated ExplodedNode, and so constructs new paths to analyze. Moreover, the condition which caused the branching is stored and the assumption whether it is true or false as well. This can potentially results in restriction on the symbolic values, and so we collect information which help us on the upcoming decisions.

It is possible that the simulation encounters a call of an other function. At this point the called function will be evaluated/analyzed on the spot of the call, knowing the calling context. This allows us to make a more precise analysis of the called function, since we potentially have information on the possible values of its arguments. The analysis of a function while knowing the calling context is called an *inline* analysis. The other possibility – when we have absolutely no information or precondition on the values of a function and it is analyzed so – is called an *top level* analysis.

In order to decrease the rate of false positive reports and make the analysis more time efficient, the CSA will not reanalyze a function as top level if it was already analyzed as an inline function. The analyzer tries to inline as much function as possible. This is reached by creating the function call graph of the translation unit

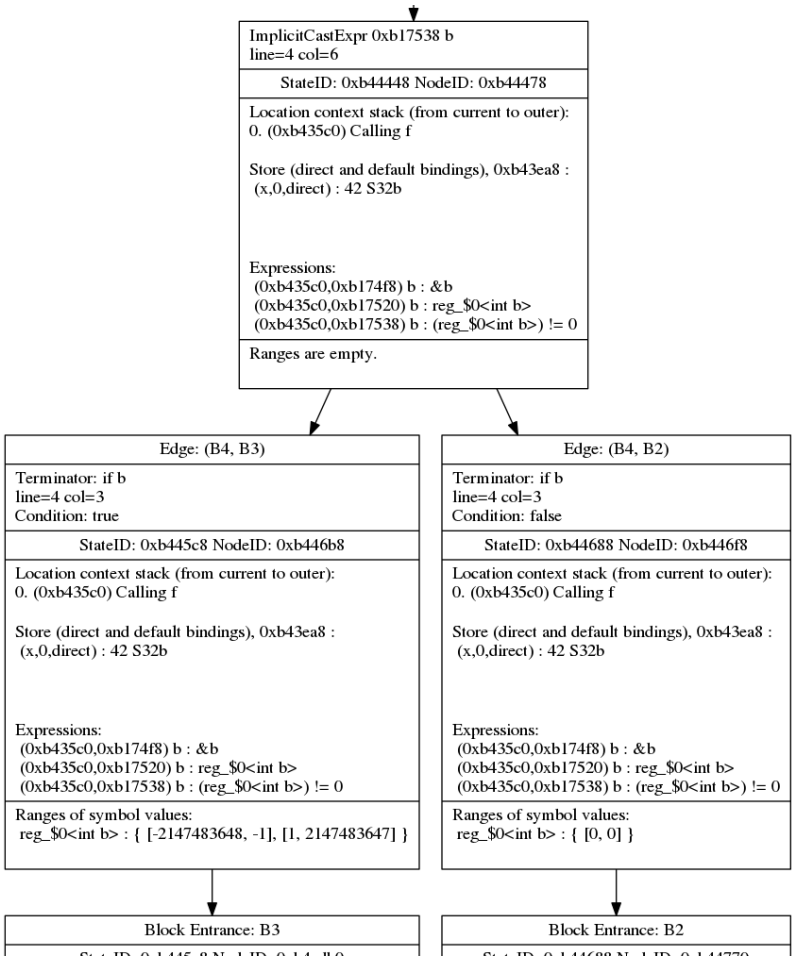

Figure 4: A small part of the actual ExplodedGraph built by the Clang Static Analyzer for the function shown on Fig 2.

and the analysis order of the functions is determined by a post-traversal-walk TODO on this call graph.

As it was mentioned earlier, branches in the execution lead to an exponential number of `ExplodedNode`s. This combinatorial explosion is handled in the Static Analyzer by restricting or even stopping the analysis when given conditions are fulfilled. Terminating the analysis 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

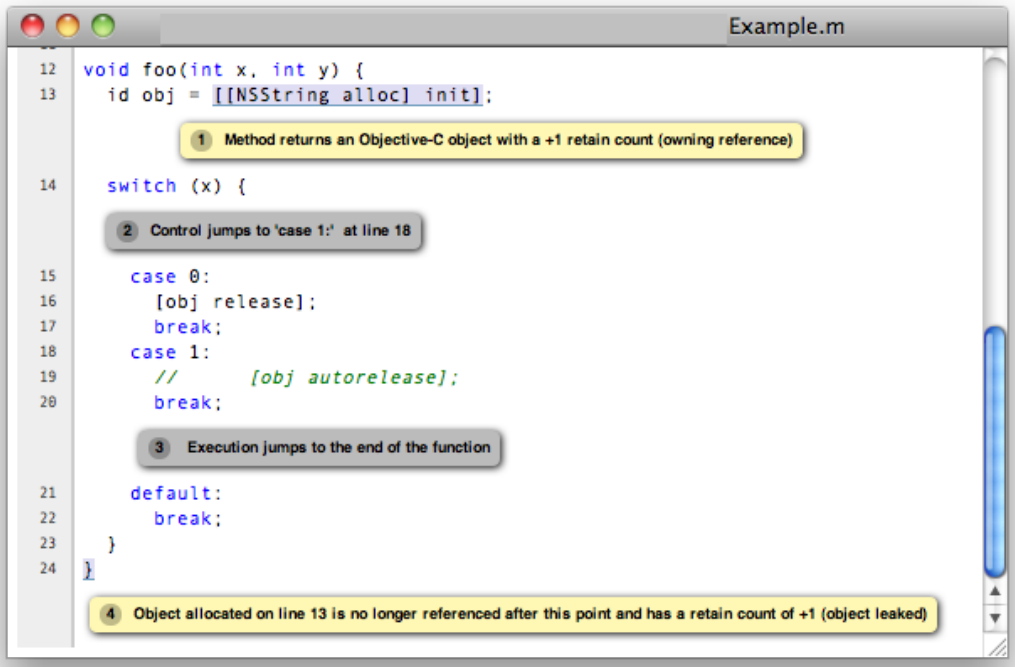

Figure 5: A bug found by the Clang Static Analyzer can be represented in HTML format to make it more clear and understandable.

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 is 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.

2.2.2 The building process of the model

The syntactic and semantic information of variables, functions and types are required to perform the above described simulation. A szimulációhoz szükséges a különböző változók, függvények, típusok szintaktikus és szemantikus tulajdonságainak ismerete. Ezen információk a C/C++ forrásfájlok esetén a fordítás

során is szükségesek. A fordítók ezeket egy saját, jól struktúrált adatszerkezetben tárolják, egy absztrakt szintaxisfát építenek fel a kód elemzése során. A gyakorlatban ez konkrét (adott platformra jellemző) információkat, illetve a szemantikus tulajdonságokat is tartalmazza. Sőt, számos fordító esetében kör is előfordulhat benne. Ilyenkor csupán csak történeti okokból hívják az adott adatszerkezetet fának. A felsorolt tulajdonságok alapján ez az adatszerkezet a szimbolikus végrehajtás számára nélkülözhetetlen információkat tartalmaz. Ebben rejlik a fő motivációja annak, hogy egy statikus elemző rendszert egy fordító köré építsünk, amely képes az alapvetően szükséges információk strukturált előállítására. Fontos még, hogy az adatszerkezetben tárolt információk helyessége jól tesztelt, hiszen minden fordítás során a kódgeneráláskor ugyanezek az információk lesznek felhasználva. A CSA a Clang fordító által épített szintaxisfát használja.

Ezek után már elegendő információ áll rendelkezésre az elemző rendszer számára, hogy (pár köztes lépésen keresztül) képes legyen felépíteni a szimulálás során létrejövő Exploded Graphot.

Az elemzés során használt modellt (az Exploded Graphot) a CSA egy optimalizált *mélységi bejárás* segítségével építi fel és járja be egyszerre. Ennek fő oka, hogy a mélységi bejáráshoz elegendő csak az adott útvonalhoz tartozó csúcsok memóriában tartása. Ezzel szemben egy szélességi bejárás során az összes elemzett útvonalat a memóriában kell tartanunk az útvonalérzékenység megtartása érdekében. Továbbá a mélységi bejárás esetén kevesebb kontextus váltásra van szükség: az adott útvonal elemzése során kevés memóriarégió és szimbólum kerül felhasználásra, nagyobb eséllyel találhatóak meg a felhasznált adatszerkezetek a processzor gyorsítótárában, így hatékonyabban tudjuk végrehajtani az elemzést.

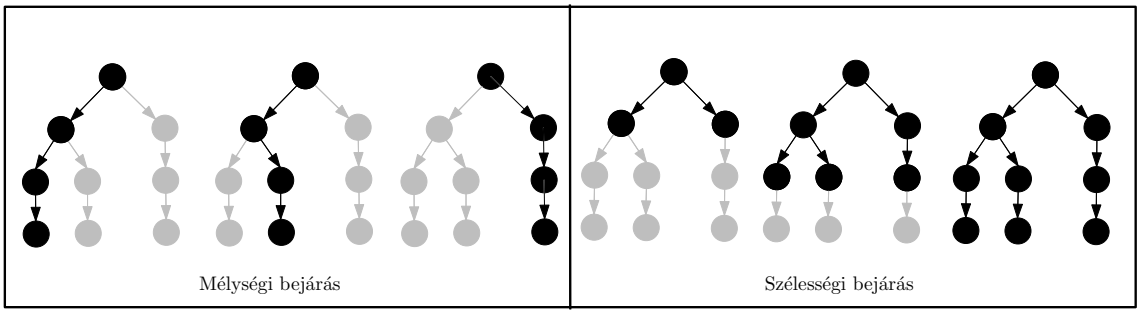

Figure 6: The different traverses of the Exploded Graph and the parts which needs to be stored in the memory (black). Depth first search (left) and breath first search (right).

Moreover, in order to use less memory during the analysis, the CSA clears the ExplodedNodes which are unnecessarily kept in the memory. This mainly means, the nodes that are created to help the analysis but not representing concrete state changes in the actual running of the program.

An important technical note is that the building of the `ExplodedGraph` is based on the `Control Flow Graph` (`CFG`) of the functions. The `CFG` represents a source-level, intra-procedural control flow of a statement. This statement can potentially be an entire function body, or just a single expression. The `CFG` consists of `CFGBlocks` which are simply containers of statements. The `CFGBlocks`s essentially represent the basic blocks of the code but can contain some extra custom information. Although basic blocks and `CFGBlocks` are technically different, in the rest of the article the term basic blocks will be used for `CFGBlocks` as well for the sake of easier understanding and better illustration.

Thus during the analysis – based on the function CFGs – an `ExplodedGraph` is built up. A node of this graph (called `ExplodedNode`) contains a `ProgramPoint` (which determines the location) and a `State` (which contains any 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.

3 Motivation

Currently, the analyzer handles loops quite simply. 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. This behavior is reached by the above presented basic block visiting budget (restriction no. 4).

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.

```
1 void foo() {
2   int arr[6];
3   for (int i = 0; i < 6; i++) {
4     arr[i] = i;
5   }
6   /*rest of the function*/
7 }
```

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:

```
1 int num();
2 void foo() {
3   int n = 0;
4   for (int i = 0; i < num(); ++i) {
5     ++n;
6   }
7   /*rest of the function, n < 4 */
8 }
```

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 in an analysis which keeps track of 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 that a

new node is inserted into the `ExplodedGraph` with the new values. Since the body of the `num()` function is unknown, the analyzer can not find out its return value. Thus it is considered as unknown. This circumstance makes the graph to split into 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, at the 4th time, assuming the condition is true, the path will be cut short according to the budget rule. Even though the analyzer generates paths to simulate the code after the loop in the above described case, 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. Since the analysis did not had any precondition on the possible values of the variables, this constraint not necessary correct. Most of the time this method results in coverage loss as well, since the analyzer will ignore the paths where n is more than 4.

4 Proposed Solution

In this section two solutions are presented to resolve the above 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 the original method is brought back on examples which are too complex to handle at the moment. For the sake of simplicity a "division by zero" will illustrate the bug we intend to find in the following examples.

4.1 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:

```
1 void foo() {
2   for (int i = 0; i < 6; i++) {
3     /*simple loop which does not
4     change 'i' or split the state*/
5   }
6   int k = 0;
7   int l = 2/k; // Division by zero
8 }
```

Listing 3: Complete unrolling of the loop makes it 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 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 change the loop variable only once in its body and the difference needs to be constant. (This way the maximum number of steps can be estimated.)

3. There is no alias created to the loop variable.

4. The estimated number of steps should be less than 128. (Simulating loops which takes thousands of steps because they could single handedly exhaust the budget.)

5. The loop must not generate new branches or use `goto` statements.

By using this method, the bug on the Listing 3 example can be found successfully.

4.2 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. The analyzer already has a widening algorithm 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 in order to avoid losing the first precise simulation branches, the widening (means the invalidating) happens before the 4th step. This way the coverage will be increased, however, this method is disabled by default, since it can easily result in too much false positives. Consider the example on Listing 4.

```
1  int num();
2  void foo() {
3    bool b = true;
4    for (int i = 0; i < num(); ++i) {
5      /*does not changes 'b'*/
6    }
7    int n = 0;
8    if (b)
9      n++;
10   n = 1/n; // False positive:
11            // Division by zero
12 }
```

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

In this case the analyzer will create and check that unfeasible 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. There was already conversation within the community about some possible enhancements [Phabricator, 2015].

One of the main principles is that the analysis should still continue after the block visiting budget is exhausted and the information of only those variables should be invalidated which are possibly modified by the loop, e.g. a statement, like `arr[i] = i` where `i` is the loop variable, means that we discard the data on the whole `arr`

array but nothing else). For this I developed a solution which checks every possible way in which a variable can be modified in the loop. Then these cases are evaluated 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 that contain invalid states. This approach helps us to cover cases and find bugs like the one illustrated on Listing 5 without reporting false positives presented on Listing 4.

```
1  int num();
2  void foo() {
3    int n = 0;
4    for (int i = 0; i < num(); ++i) {
5      ++n;
6    }
7    if (n > 4) {
8      int k = 0;
9      k = 1/k;  // Division by zero error
10   }
11 }
```

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

The bug is found by invalidating the known information on variable `n` (and `i` as well). This makes 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.

```
1  int num();
2  void foo() {
3    int n = 0;
4    for (int i = 0; i < num(); ++i) {
5      ++n;
6      for (int j = 0; j < 4; ++j) {
7        /*body that does not change n*/
8      }
9    }
10   /*rest of the function, n <= 1 */
11 }
```

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 it assumes that `i < num()` is true) and encounter the inner loop, 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 moving on to the next iteration, we may assume that we are on the path where the outer loop condition is true again. Due to the fact that the budget was already exhausted 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 in the outer loop not reaching the step number where it would been widened. Furthermore, the outer loop will not even reach the 3rd step, even the 2nd is stopped at in its body (as described above). This causes the problem that even though the loop widening method is used, the rest of the function will be analyzed by 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 after 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.

An additional note to the widening process is 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.

```
1  int num();
2  void foo() {
3    int n = 0;
4    int i;
5    for (i = 0; i < num(); ++i) {
6      if (i == 7) {
7        break;
8      }
9      for (int j = 0; j < 4; ++j) {/* */}
10   }
11   int n = 1 / (7 - k);
12           // ^ Possible division by zero
13 }
```

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.

4.3 Infrastructure improvements

The discussed approaches heavily rely on the fact that we are able to perform the following actions:

1. Decide on any `ExplodedNode` whether it simulates a body of a loop or not,
2. Recognize the entering and exiting point of a loop on the simulated path.

However, this information was not provided by the analyzer earlier. Considering the lexical nature of the loops, their entrance and exit points can unambiguously fit into the `CFG`.

The `ProgramPoint` provides the `LocationContext` which implements a stack data structure for having the information on the different locations of the code. This implies that the callstack can be represented via this structure in a straightforward manner (and is contained by the `LocationContext`). Since storing the currently simulated loops fits into a stack data structure as well, this information – called `LoopContext` – understandably was implemented as the part of the `LocationContext` too instead of having them in the `Store`. Both the `Store` and the `LocationContext` are part of an `ExplodedNode` in form of a pointer. However, these structures use copy-on-write semantics, and the `LocationContext` changes way less times, this decision saves us memory.

5 Evaluation

The effect of the described loop modeling approaches was measured on various C/C++ open source projects. These are listed on Table 1.

Project	LoC	Language
TinyXML	20k	C++
Curl	21k	C
Redis	40k	C
Xerces	228k	C++
Vim	540k	C
OpenSSL	550k	C
PostgreSQL	950k	C
FFmpeg	1080k	C

Table 1: The projects used for profiling, their length in code lines, and language.

5.1 Coverage and the number of explored paths

Keeping track of these statisics are already part of the analyzer. The coverage percentage is based on the ratio of the visited and the total number of basic blocks in the analyzed functions (instead of the number of visited statements), which results in a small imprecision. It is important to note that the introduced loop modeling methods require having additional loop entrance and exit point information (described in section 4.3) in the `CFG`. This can lead to having more basic blocks in the `CFG` and it can affect the statistics. As a result, even statistics produced by using the current loop modeling approach were measured with this information added to the `CFG`.

The coverage and the number of explored paths are generated for every translation unit and then summarized. This means that header files which are included in more than one translation unit can influence more statistics. However, by using this summarization process consistently for every measurement the results reflect the reality.

The tables presented in this section summarize measurement results using different loop modeling approaches: the current practice (denoted by Normal) and the hereby introduced loop unrolling (Unroll) and loop widening (Widen) methods separately and simultaneously (U+W).

Table 2 shows the coverage difference using the introduced approaches. On most of the projects, analysis coverage was strictly increased by using any of the proposed approaches. The widening method had a stronger influence on the coverage in the average case. However, the complete unroll of specific loops could result in a higher

Project	Normal	Unroll	Widen	U + W
TinyXML	84.2	84.2	85.1	85.1
Curl	76.2	76.9	77.7	77.2
Redis	68.5	69.1	68.5	71.3
Xerces	92.3	92.4	92.7	92.7
Vim	60.4	60.6	60.6	60.7
OpenSSL	97.4	97.5	97.7	97.7
PostgreSQL	76.9	77.0	76.9	76.9
FFmpeg	86.1	86.3	87.0	86.8

Table 2: The code coverage of the analysis on the evaluated projects expressed in percentage.

coverage as well (e.g. Curl, Redis). In general, enabling both of them was the most beneficial with respect to the coverage.

Project	Normal	Unroll	Widen	U + W
TinyXML	14 452	15 460	14 765	15 773
Curl	18 272	18 577	28 835	24 279
Redis	69 857	70 097	98 446	100 929
Xerces	395 615	398 077	430 989	433 358
Vim	155 451	157 266	188 136	173 121
OpenSSL	687 175	687 932	700 464	701 013
PostgreSQL	382 660	383 874	453 188	419 118
FFmpeg	466 613	458 480	571 399	521 725

Table 3: The numbers of explored execution paths using different loop modeling approaches.

Table 3 presents the numbers of analyzed execution paths. As expected, both introduced loop modeling methods resulted in a higher number of simulated paths on (almost) all of the projects. The only exception is the unrolling approach on the FFmpeg project, which caused the budget limiting the number of traversed `ExplodedNode`s to exhaust earlier, slightly decreasing the number of checked paths. Enabling both of the features resulted in similar or fewer number of explored paths than the runs using only widening. This effect can be explained in two ways: (1) the analyzer prefers to completely unroll loops rather than widen them, which results in a more precise modeling of the state and can exclude unfeasible paths, (2) the simultaneous use of the methods can lead to exhausting the budget on earlier paths, where the analysis will be terminated.

5.2 Found bugs

The number of bug reports using the different loop modeling methods can be seen in Table 4. The increase in analysis coverage and in the number of checked paths

Project	Normal	Unroll	Widen	U + W
TinyXML	1	1	3	3 (+200%)
Curl	16	16	16	16 (0%)
Redis	55	58	55	59 (+7.27%)
Xerces	62	62	61	61 (-1.61%)
Vim	74	74	76	78 (+5.4%)
OpenSSL	152	152	153	153 (+0.66%)
PostgreSQL	323	323	327	331 (+2.48%)
FFmpeg	425	420	423	454 (+6.82%)

Table 4: The number of bug reports produced by the analyzer.

usually implies an increased number of found bugs, which indeed can be observed on the numbers. However, it is important to note that the upsurge of the number of explored execution paths described in Table 3 considerably outweighs the moderate rise in the number of bug reports. Since the loop widening method creates more new paths by discarding informations on the values of variables, it could introduce the risk of analyzing paths that lead to false positives. However, from the results it seems that this was not a problem in practice: relative to the increase in the number of analyzed paths, the number of reports hardly increased. Moreover, based on studying the environment of the found bugs, the ratio of false positive findings was low (beside some clear true positive) among the newly detected bugs.

5.3 Analysis time

Project	Normal	Unroll	Widen	U + W
TinyXML	0:51	0:51	0:52	0:52 (+2%)
Curl	0:50	1:06	0:55	1:05 (+30%)
Redis	2:06	2:11	2:28	2:10 (+3%)
Xerces	3:38	3:34	3:37	3:39 (+0.5%)
Vim	3:11	3:26	3:18	3:27 (+3%)
OpenSSL	2:04	2:22	2:13	2:19 (+8.3%)
PostgreSQL	7:03	8:32	7:48	7:59 (+13%)
FFmpeg	9:40	10:22	10:14	11:20 (+17%)

Table 5: Average measured time of the analysis expressed in minutes. (Average of 5 runs.)

The running time on different projects is showed in Table 5. Although the widening method lead into more analyzed execution paths, the analysis time increase was more intense after enabling the unrolling process. This is possible due to the fact that unrolling leads to long paths where the `State` usually contains more information (constraints on variable values), which is very expensive in respect of running

time. In general there was only a minor increase in the analysis time at all examined projects which suggests a good scalability of the proposed improvements.

6 Conclusion

Two alternative approaches was introduced for improving the simulation of loops during symbolic execution. These were implemented and subsequently evaluated on various open source projects, with a clear improvement of code coverage in general. The new methods make it possible to explore previously skipped, feasible execution paths, especially when both of them are used in conjunction.

The required changes done to the underlying infrastructure should also ease the implementation of future enhancements. In particular, information tracked by the analysis about location contexts were expanded with additional fields. While code coverage was measured to have increased by an average of 0.8% and the number of explored execution paths by an average of 16%, there was a noticeable performance penalty as well. A general increase in the runtime was observed, with an average of 9.5% . The number of simulated paths also increased proportionally with the time taken, suggesting this time was well spent. In conclusion, if the user does not mind taking $\sim$10% more time for a more comprehensive analysis, then it is beneficial to enable the proposed feature set by default.

7 Future work

The heuristic patterns for completely unrolled loops could be extended to involve loops whose bound is a known variable which is not changed in the body. Furthermore, even more general rules would be beneficial: consider loops where the value variables are known at the beginning and they are affected by a known constant change by every iteration. These improvements have not been implemented yet due to some technical and framework limitations.

During the widening process we invalidate any possibly changed information. However, a change made on a pointer could mean that we need to invalidate all variables due to the lack of advanced pointer analysis. Therefore, introducing pointer analysis algorithms to the analyzer could help to develop a more precise invalidation process.

The infrastructural improvements enable the analyzer to provide entry points for bug finding modules (checkers) on loop entrances/exits and identify the currently simulated loop for every `ExplodedNode`. On top of these entry points new checkers can be implemented e.g. a check for finding possible infinite loops.

8 Acknowledgement

I would like to thank Laszlo Makk and the members of the `CodeChecker` team at Ericsson for their valuable and helpful suggestions on the paper.

This study was supported by the ÚNKP-17-2 New National Excellence Program of the Hungarian Ministry of Human Capacities.

Emberi Erőforrások
Minisztériuma

Abstract

The LLVM Clang Static Analyzer is a source code analysis tool which aims 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 a predefined constant number of times. However, this approach can result in a loss of coverage in various cases.

This study aims to introduce two alternative approaches which can extend the current method and can be applied simultaneously: (1) determining loops worth to fully unroll with applied heuristics, and (2) using a widening mechanism to simulate an arbitrary number of iteration steps. These methods were evaluated on numerous open source projects, and proved to increase coverage in most of the cases. This work also laid the infrastructure for future loop modeling improvements.

9 References

[Bessey et al., 2010] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. (2010). A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75.

[Boehm and Basili, 2001] Boehm, B. and Basili, V. R. (2001). Software defect reduction top 10 list. *Computer*, 34(1):135–137.

[Hampapuram et al., 2005] Hampapuram, H., Yang, Y., and Das, M. (2005). Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Softw. Eng. Notes*, 31(1):52–58.

[King, 1975] King, J. C. (1975). A new approach to program testing. In *Proceedings of the international conference on Reliable software*.

[Michael and Robert, 2009] Michael, Z. and Robert, K. C. (2009). The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90.

[Phabricator, 2015] Phabricator (2015). Community conversion about loop widening. https://reviews.llvm.org/D12358.

[Rice, 1953] Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366.