Bug path reduction strategies for symbolic execution

Tibor Brunner, Péter Szécsi, Zoltán Porkoláb

Abstract: Symbolic execution is one of the most powerful tools in static analysis for finding bugs. In this technique the source code is not executed by the CPU, but interpreted statement-by-statement by preserving their semantics as much as possible. During the interpretation the possible values of variables are recorded so the analyzer can report when a variable gets to an invalid state. An additional benefit of such a provided bug report is that not only the place of the bug is returned but the control flow is also displayed which results the erroneous program state. Many times the bug path contains statements which are not related to the bug. These irrelevant statements make it harder to understand in the debugging process, how the error can occur. In this article we present three methods for shortening the bug paths by removing unnecessary bug points and by recognizing two different bug paths as unique if those refer the same bug.

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. 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). In the industry the goal of these tools is to keep the ratio of the false positive reports low while still being able to find real bugs.

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 [King, 1975] [Hampapuram et al., 2005]. 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.

Concepts

C++ is a compiled language which means that programs have to be transformed to a binary code for running. In static analysis techniques this phase is skipped, thus our bug finding process makes observations in compile-time. Symbolic execution is one of the most powerful techniques for bug finding as it not only considers the abstract syntax tree (AST) of the program, but also maintains the possible values of variables.

The source code is interpreted statement-by-statement while their semantics are preserved as much as possible. For example when the analyzer meets an `if` statement then the analysis is divided to two branches too. Or if a loop statement is arriving, then the analyzer also interprets the body statements with some limitations. The point is that the code is not run by CPU, but by the static interpreter. This interpreter is equipped with a constraint solver which aims to maintain the possible value range of the variables and states of composite objects. When a *checker* notices that a variable or an object is in invalid state, it can report the error to the users.

The analysis starts from certain functions. These functions which are the sources of the simulated execution are called *top level* functions. The execution path from top level functions to an error point is called a *bug path*. Along the bug path there may be some statements which are essential points concerning the bug. These statements take part in the establishment of a bug. Suppose that there is a division in the program. If the denominator is zero then an error has to be reported. The value of the denominator expression can be assembled of several variables. Their values are set in previous statements, maybe in the body of an `if` statement. All the statements which affect the value of the denominator are so called *bug points*. The *bug length* is the number of bug points on the bug path.

Notice that this method also provides the path leading to the erroneous statement besides the place of potential error in order to help the programmer to find how the bug can arise. Along the bug path the value ranges of variables and the object states are also available, so the context is provided to the users too. These information are crucial in understanding what circumstances play role in the occurrence of a bug. The goal of the Static Analyzer is not only to report possible errors, but to cut the debugging session short.

Problems

The problem with this approach is that some bug points are also added to the bug path which are not relevant from the bug point of view. It is not helpful if the user has to inspect each step of the path and check the values of variables on certain control flows which have nothing to do with the actual bug. The appearance of such bug points is inevitable, since it might not be possible for a checker to decide whether that point is important in case of the bug.

Another problem is that there may be several bugs which can be reached from multiple places. Let us consider a function which contains a bug. There are as many potential bug paths as many paths lead to this bug from other functions. These paths are reported to the users as separate bugs which are to be fixed one by one. This is time consuming and also unnecessary, since the bug should be fixed at one place in the code base, namely in the bug container function. The situation is even worse if the bug happens in a header file. Headers are included in several translation units, so it is likely that their functions are invoked from quite a lot of locations.

Solutions

In this section we propose solutions on the above mentioned problems. For the sake of simplicity a "division by zero error" or "null pointer dereference" will illustrate the bug we intend to find and reduce its bug path length in the following examples.

Throwing unnecessary points by slicing

The first solution intends to eliminate the unnecessary statements from the bug path which are not related to the specific report. We say that a bug point is irrelevant if the variables in the final bug point are not affected by any assignment in the statements of the previous bug points. This technique is the so called backward-slicing. In our case this algorithm can be accomplished in a straightforward manner, since the list of these preceding statements is already given by Static Analyzer as the nodes of the exploded graph. These nodes contain exactly those statements and their changes which were important in some way according to the checker in the composition of the final error state. The typical example is the sequence of `if` or other conditional statements before the bug. The analyzer engine takes these conditions into account even if their *false* branch is taken and thus not related to the bug.

```
void f(int a, int b) {
  int x = 0;
  if (a < 42) { action1(); }
  if (b > 3)  { action2(); }
  int y = 1 / x; // Division by zero error
}
```

Listing 4: Code snippet, where the important statement from the view of the bug (seting the value of `x` to zero) is far from the actual bug occurrence (division by zero), however, the statements between them are not relevant.

In the example shown on Listing 4 the conditions of the branching statements will be the part of the bug path by default, assuming that the *false* branches are taken even if these are not important for the bug at all. These can be removed from the bug path in a post-processing session.

Determining unique paths by merge locations

Another category of bug shortening possibilities is finding the locations to unique the bug paths which are considered the same from a specific point. In order to combine these reports, the checkers can specify a statement which is the root of all the following errors. This common statement is called the *uniqueing location*.

```cpp
void main() {
  int x;
  std::cin >> x;
  int *p = nullptr;
  // ... Not setting p
  if (x) { *p; } // Null pointer dereference
  else   { *p; } // Null pointer dereference
}
```

Listing 5: The null pointer dereference occurs at two different positions, however, logically the error is committed once, when the value of variable `p` was not set other than a null pointer.

Listing 5 demonstrates that it does not matter from which direction the null pointer dereference is reached, because the problem is that the pointer is not set correctly before the `if` statement. In the current solution of Static Analyzer, checkers can provide the specific statement, i.e. the *uniqueing location*, that resulted the error (null pointer assignment to `p` in this case), but this is always local to a translation unit. With a post-processing session we can collect all these statements for the bugs among translation units thus determining a global uniqueing location which enables us to reduce the number of separate bug paths.

Cutting a bug path prefix

By the nature of Static Analyzer the bug paths are determined from the beginning of the top level function, where the analysis started. This doesn't mean that all statements along the found bug path are needed. In practice there is a relevant point on the path which is enough to start from while debugging. For finding this point we rerun the analysis starting in the bug points from the end towards the top function. The first location from where the bug arises is enough to display.

Listing 6 shows a case where the analyzer has to split the simulation, and simulate both branches of the `if` statement since it has no information about the concrete value of `x`. This will result in two different bugs since both paths contain the error and the bug paths are different. However, the calling context does not affect the occurrence of the error. Thus, we could cut the bug path to begin from `foo` function, as the bug can be surely determined by the top level analysis of `foo`. This method not only helps the programmers to understand the code easier but also shows that the two reported bugs on Listing 6 are equivalent.

```cpp
int foo(int k) {
  int num = 42;
  return k / (num - 42); // Division by zero error
}

void main() {
  int x;
  cin >> x;
  if (x) foo(2);
  else   foo(3);
}
```

Listing 6: Since the calling context is irrelevant in the view of the division by zero error, there is no need report it twice.

Conclusion

Clang Static Analyzer is a powerful tool for bug finding with static analysis. However, in some cases it generates unnecessarily long bug paths which are not worth to entirely review by programmers while debugging, since it contains irrelevant information from the specific bug point of view. We can introduce three methods using post-processing for shortening these paths by removing unnecessary bug points and by recognizing two different bug paths as unique if those refer the same bug.

Acknowledgement

This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

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.

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