Source Code Assertion Verification Using Backward Symbolic Execution

Robert Husák and Filip Zavoral

Charles University, Ovocný trh 5, Prague 1, 116 36, Czech Republic

{husak, zavoral}@ksi.mff.cuni.cz

Abstract. In order to prevent, detect and fix errors in software, various code analysis techniques have been developed and are practically used, e.g. abstract interpretation or concolic execution. We want to bring into attention backward symbolic execution, a technique not widely used despite its advantages such as a demand-driven approach. We synthesize its current state-of-the-art into an algorithm extensible by various heuristics. This algorithm is then implemented and evaluated on a set of example programs which contain both valid and invalid assertions to be verified and refuted. According to the results, backward symbolic execution can successfully complement both abstract interpretation and concolic execution to achieve significantly higher success rate.

INTRODUCTION

When working on large software projects, developers are currently mostly opposed to using static analysis techniques that reason about exact semantics of programs. This is caused both by scalability issues and the overhead involved in integrating them into the development process. These techniques comprise, for example, bounded model checking or symbolic execution [1, 2, 3]. However, recently there have been successful attempts to use a combination of symbolic and concrete execution, called concolic execution, to efficiently generate test inputs [4]. Furthermore, it is possible to utilize abstract interpretation [5, 6] to verify even larger programs in modular fashion, but the amount of hints needed by such checkers is substantial. In this paper we will focus on the backward variant of symbolic execution, which utilizes all the resources to verify one program assertion at a time [7, 8]. Our first contribution is a general extensible algorithm created by summarizing the current approaches to the optimization of backward symbolic execution. The second contribution is an evaluation of this algorithm in comparison with other existing techniques.

BACKWARD SYMBOLIC EXECUTION

We present a synthesis of existing techniques used for backward symbolic execution [7, 9, 10] into a general algorithm, which can be easily parametrized. The basic goal of the algorithm is to explore all the execution paths leading to a certain assertion in the code and discover whether it can be invalid in any of them. In the following text, we will explain the general logic of the algorithm as well as all the opportunities to extend it, which will take the form of subroutines. We assume that we are capable of turning procedures written in code to a set of control flow graphs (CFG). For simplicity, each inner node can contain only one assignment of a symbolic expression to a symbolic variable. We also expect that there are special types of nodes to represent routine entry points, subroutine calls, exception throws etc.

A CFG node uniquely represents a *location* in a program. A *path* is either a simple sequence of locations or an oriented acyclic graph resulting from path merging [9], where the edges represent the possible control flow. The *extendPath* function adds a location to the beginning of the given path. The *location* function retrieves the last location added to a path. A *path condition* is a formula in first order logic describing a path. By passing a path condition to an SMT solver we can determine the feasibility of the path. Because a single program variable can hold multiple values throughout the path execution, each of them is distinguished by a version number in the path condition.

```
1:  // Set up the starting state
2:  p_0 := (l_0)
3:  w := {(p_0, ¬a_0, λv.0)}
4:  while w ≠ ∅ do
5:      (p, pc, m) := pickNext(w)
6:      w := w \ {(p, pc, m)}
7:      // Process the successors of the selected state
8:      for all {e ∈ ingoingEdges(location(p)) : doFollow(e, p, pc)} do
9:          p' := extendPath(p, e.from)
10:         (pc', m') := addEdgeCondition(p, pc, m, e)
11:         // Process the assignment if present
12:         if content(e.from) is v ← e then
13:             m' := m[v ↦ m(v) + 1]
14:             pc' := addCondition(p', pc', v_{m(v)} = versionedExpression(e, m'))
15:         end if
16:         // Optionally merge the successor with another state in the worklist
17:         if ∃(p'', pc'', m'') ∈ w : doMerge((p', pc', m'), (p'', pc'', m'')) then
18:             w := w \ (p'', pc'', m'')
19:             (p', pc', m') := merge((p', pc', m'), (p'', pc'', m''))
20:         end if
21:         // Check the path condition if needed and optionally report the result
22:         if isEntryPoint(location(p')) then
23:             reportResult(p', pc', check(pc'))
24:             continue
25:         else if doCheck(p', pc', m') then
26:             r := check(pc')
27:             if r ≠ SAT then
28:                 reportResult(p', pc', r)
29:                 continue
30:             end if
31:         end if
32:         // Add the successor to the worklist
33:         w := w ∪ {(p', pc', m')}
34:     end for
35: end while
```

FIGURE 1. Backward symbolic execution algorithm.

In the algorithm, depicted in Figure 1, we manage a worklist of exploration states named w. Each state (p, pc, m) is represented by its path p, path condition pc and variable version mapping function m. A location l_0 represents the location to start the search from and an expression a_0 is the assertion to verify. As we can see on line 3, the worklist starts only with the state containing the initial location and the negated assertion to be verified. Until the worklist is empty, we repetitively select a state using the *pickNext* function on line 5 and process it. This function is designed as one of the heuristics that can be injected to the algorithm. Another one, *doFollow*, can prevent the algorithm from exploring unreasonable paths. For example, it can limit the number of times a cycle is unwound.

The *doFollow* function is used on line 8 to filter the results of the *ingoingEdges* function. For most of the CFG node types, this function just returns its ingoing edges in the current graph. However, enter and call nodes are handled separately by creating custom temporary edges that model the interprocedural flow. The *addEdgeCondition* function on line 10 finishes this task together with optionally updating the variable version map and the path condition according to the edge.

On lines 12-15, an assignment of an expression to a variable is processed. For the case the variable appears in the expression, we need to make sure its version is different there by incrementing it in the version map. As a result,

the path condition contains variables with version numbers increasing against the control flow, where each variable is assigned at most once. For example, when $\forall v : m(v) = 0$, an assignment $x \leftarrow x + y$ forms the condition $x_0 = x_1 + y_0$.

The resulting state can be then merged with another state on the same location by the *merge* function. As we can see on lines 17-20, we can influence the merging by the heuristic function *doMerge*. The full merging mechanism and useful heuristics are described in [9]. In general, state merging can reduce the number of calls to an SMT solver, but it can also significantly increase the complexity of each path condition.

On lines 22-31 is depicted the process of checking the satisfiability of the path condition and possibly reporting the result. If we reached an entry point, which is determined by *isEntryPoint*, we always perform the check, report its result and continue without adding the state to the worklist. The same situation happens when the heuristic function *doCheck* tells us to perform the check and it is either unsatisfiable or unknown. When implemented properly, it can help the algorithm to cut infeasible paths early on, saving resources. Path condition checking is then implemented in *check*. Various optimizations can be used there, possibly utilizing assertion stacks of SMT solvers [10].

The *reportResult* function analyses the result of the satisfiability check. In the case of SAT, it converts the model to an execution path with corresponding values assigned to each variable version along the way. If the result is UNSAT, the checked path and the nodes and edges corresponding to the unsatisfiable core are returned. If the result is unknown, only the path and a description of the reason is returned. If the result was not reported, we add the successor to the worklist on line 33 and loop again.

The complexity of the algorithm depends on the number of distinct execution paths leading from entry points to l_0 and their path conditions. If the number is finite, the path conditions are decidable and *doFollow* returns *true* for each valid path, we can guarantee that only the paths with satisfiable path conditions can lead to breaking a_0. Due to the path explosion problem, the number of paths can be too high to handle in a reasonable time or even infinite. In that case, it is important to implement *pickNext*, *doFollow* and *isEntryPoint* in a way that we can provide at least weak guarantees based on under-approximation. For example, we can limit the number of times each cycle is unwound, the recursion depth, or we can shift certain entry points closer to l_0.

In general, the main outcome for developers is the ability to point them in the right direction when they search for a cause of one particular problem in the program. A simple but useful scenario is when a developer searches for the origin of a certain value that caused an exception in a method. Instead of manually traversing the call graph and searching for the logic behind passing that value, the algorithm can be used to do it automatically.

EVALUATION

According to the algorithm, we implemented an extension of Microsoft Visual Studio called AskTheCode [11], which uses the Roslyn compiler to analyse C# code and Z3 to solve path conditions. The heuristic functions used in the evaluation are rather simple: *pickNext* extracts states in a random fashion, *doFollow* always returns *true*, *doMerge* always returns *false*, *isEntryPoint* returns *true* for entry points of public methods in public classes and *doCheck* returns *true* iff the current CFG node has more than one incoming edge.

To evaluate AskTheCode, we assembled a set of short example programs used in the research of automated software testing. The *Integers* [12] example contains several short functions covering linear and non-linear integer arithmetic. *Trityp* [13] represents a problem with excessive branching and a complicated data flow, which, however, does not contain any loops. *BankAccount* [14] is aimed at refuting invalid assertions about a data structure in a nondeterministic sequence of operations. As its name suggests, *Loops* [15] comprises various types of loops containing assertions, some of which can be verified by backward symbolic execution even without reaching their entry points.

In order to discover whether our approach is not redundant within the other techniques practically used for high-level languages such as C#, we included also the most popular ones in the evaluation. Code Contracts static checker [6] uses abstract interpretation, which analyses individual methods and infers various facts about program variables and relations between them. These can be then be composed in an interprocedural manner and used to prove or refute assertions. Microsoft Pex [4], representing concolic execution, automatically generates test inputs using an SMT solver to achieve high code coverage and discover possible program errors. Because of its purpose, it does not aim to prove assertions, but it can be very powerful in refuting them.

From the results in Table 1 we can see that neither of the techniques is superior to the others in all the cases. Furthermore, even when abstract interpretation and concolic execution are used together, they still miss certain assertions which can be covered by backward symbolic execution. Therefore, we can generally conclude that backward symbolic execution has the potential to complement the existing practically used techniques well and can also substitute them to some extent. For the particular test programs and a detailed interpretation of the results we refer to [11].

TABLE 1. Comparison of different code analysis techniques on four scenarios. Each scenario provides the numbers of valid and invalid assertions in the header. Every technique then provides the numbers of proved valid and refuted invalid assertions. For each assertion, a time limit of 30 seconds was given to the tools.

	Integers [12]	**Trityp** [13]	**BankAccount** [14]	**Loops** [15]
Assertions: valid - invalid	**4 - 5**	**1 - 5**	**0 - 3**	**4 - 6**
Backward symbolic execution	4 - 2	1 - 5	0 - 3	2 - 4
Abstract interpretation	2 - 4	0 - 3	0 - 0	2 - 0
Concolic execution	0 - 4	0 - 5	0 - 3	0 - 5

CONCLUSION

We have synthesized several approaches to backward symbolic execution optimization, presenting a general algorithm structure, which can be easily extended by various heuristics. Its evaluation on several examples shows that it can be successfully used to complement other code analysis techniques.

ACKNOWLEDGMENTS

This work was supported by project PROGRESS Q48.

REFERENCES

[1] J. C. King, Commun. ACM **19**, 385–394 (1976).

[2] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, Automated Software Engineering **20**, 391–425 (2013).

[3] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08 (USENIX Association, 2008), pp. 209–224.

[4] N. Tillmann and J. De Halleux, "Pex: White box test generation for .NET," in *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08 (Springer-Verlag, 2008), pp. 134–153.

[5] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77 (ACM, 1977), pp. 238–252.

[6] M. Fähndrich, "Static verification for code contracts," in *Static Analysis*, edited by R. Cousot and M. Martel (Springer Berlin Heidelberg, 2010), pp. 2–5.

[7] R. Sites, Branch resolution via backward symbolic execution, 1995, US Patent 5,428,786.

[8] S. Chandra, S. J. Fink, and M. Sridharan, SIGPLAN Not. **44**, 363–374 (2009).

[9] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12 (ACM, 2012), pp. 193–204.

[10] T. Liu, M. Taghdiri, M. Araújo, and M. d'Amorim, *A comparative study of incremental constraint solving approaches in symbolic execution.*, Lecture Notes in Computer Science, Vol. 8855 (Springer Verlag, 2014).

[11] R. Husák, "Code assertions verification using backward symbolic execution," (2017), Master's thesis, under the supervision of Jan Kofroň, `https://dspace.cuni.cz/handle/20.500.11956/2087`.

[12] P. Dinges and G. Agha, "Targeted test input generation using symbolic-concrete backward execution," in *29th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (ACM, 2014).

[13] P. Cellier, M. Ducassé, S. Ferré, and O. Ridoux, "Formal concept analysis enhances fault localization in software," in *Formal Concept Analysis: 6th International Conference, ICFCA 2008, Montreal, Canada, February 25-28, 2008. Proceedings*, edited by R. Medina and S. Obiedkov (Springer, 2008), pp. 273–288.

[14] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)* (2008), pp. 297–306.

[15] F. Charreteur and A. Gotlieb, "Constraint-based test input generation for java bytecode," in *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010* (2010), pp. 131–140.