Dynamic Symbolic Execution with Interpolation Based Path Merging

Andreas Ibing

Chair for IT Security
TU München
Boltzmannstrasse 3, 85748 Garching, Germany
Email: andreas.ibing@tum.de

***Abstract*—This paper presents a dynamic symbolic execution engine for automated bug detection in C code. It uses path merging based on interpolation with unsatisfiable cores to mitigate the exponential path explosion problem. Code coverage can be scaled by varying the interpolation. An algorithm for error and branch coverage is described. The implementation extends Eclipse CDT. It is evaluated on buffer overflow test cases from the Juliet test suite in terms of speed-up through merging, reduction of the number of analyzed program paths and proportion of merged paths.**

***Keywords*–Symbolic execution, interpolation, branch coverage, error coverage.**

I. Introduction

Symbolic execution [1] is a program analysis technique, that can be used for automated bug detection. In order to find bugs with arbitrary program input, the program input is treated as symbolic variables. Operations on these variables then yield logic equations. Satisfiability of program paths and satisfiability of bug conditions are decided by an automated theorem prover (constraint solver). The current state of automated theorem provers are Satisfiability Modulo Theories (SMT) provers [2].

Symbolic execution can be applied both as static analysis (without executing the progran under test) and as dynamic analysis (using binary instrumentation) [3]. Dynamic symbolic execution follows a program path with a complete concrete program state, and additionally a partial symbolic program state. The partial symbolic program state comprises the constraints on symbolic variables which have been collected on the path (path constraint). The concrete program state satisfies the constraints on the symbolic variables. Dynamic symbolic execution is also known as concolic execution (concrete/symbolic [4]). Dynamic symbolic execution has several advantages compared to the static only approach. Complicated program constructs can be concretized, i.e., executed only concretely by dropping the relevant symbolic variables [3]. Concretization is sound with respect to bug detection, i.e., while it does lead to false negative detections, it does not lead to false positive bug detections. Concretization also provides more flexibility in handling library function calls. Function call parameters can be concretized and the function executed concretely. Dynamic symbolic execution with configurable concretization is also called selective symbolic execution [5]. Another argument for dynamic symbolic execution is that execution of concrete code is much faster than symbolic interpretation.

The number of satisfiable program paths in general grows exponentially with the number of branch decisions, for which more than one branch is satisfiable. This bad scaling behaviour is known as path explosion problem. In order to alleviate the path explosion problem, it is shown in [6] that a live variable analysis can be applied so that program paths, that only differ in dead variables, can be merged. A more comprehensive sound path merging approach is described in [7]. It is based on logic interpolation (Craig interpolation [8]), i.e., on automated generalization of constraint formulas. The interpolation uses unsatisfiable cores (unsat-cores) and approximates weakest precondition computation. Given an unsatisfiable conjunction of formulas, an unsat-core is a subset of the formulas whose conjunction is still unsatisfiable. This approach leads to better scaling behaviour by finding more possibilities to merge program paths.

The accuracy of bug detection tools is typically evaluated as percentage of false positive and false negative bug detections in a sufficiently large bug test suite. Currently the most comprehensive test suite for C/C++ is the Juliet suite [9]. In order to systematically test a tool's accuracy, it combines 'baseline' bugs with different data and control flow variants. The maximum context depth spanned by a flow variant is five functions in five different source files. Each test case is a program that contains 'good' (bug-free) as well as 'bad' functions (which contain a bug), so that both false positives and false negatives can be measured.

This paper presents a dynamic symbolic execution engine which uses unsat-core based interpolation of unsatisfiable program paths and unsatisfiable bug conditions in order to achieve scalability through merging as many program paths as early as possible. The engine is applied to the problem of automated bug detection (testing). This includes that with each bug detection, the constraints for merging program paths are automatically adapted.

The remainder of this paper is organized as follows: Section II describes the motivation and details of the algorithm. Section III describes scaling of code coverage by varying interpolation, and puts the described algorithm for error and branch coverage into context. The implementation as plug-in extension to the Eclipse C/C++ development tools (CDT) is depicted in section IV. Section V evaluates the tool in terms of speed-up through path merging and of the number of completely and partly analyzed program paths on buffer overflow test cases from the Juliet test suite. Related work is described in section VI. Evaluation results are discussed in section VII.

Copyright (c) IARIA, 2016.

II. Algorithm

This section describes the motivation (in subsection II-A) and details of the algorithm and gives an analysis example in subsection II-E.

A. Motivation

Motivation for the algorithm are the following points:

- It is sufficient to detect each bug on one program path only. It is not necessary to detect each bug on all paths where this bug might be triggered. A program path can therefore be pruned if it is impossible to detect any new bugs on any extension of the path. This information can be gained from backtracking program paths, that were analyzed till program end. This implies a depth-first traversal of the program execution tree (the tree of satisfiable program paths).
- Since interpretation is much slower than execution of code, as much code as possible should not be interpreted. Execution should be interrupted only at locations, that need to be interpreted symbolically. Constraint-based analysis is needed only for the detection of input dependent bugs. In this paper, a debugger is used for adaptive binary instrumentation, i.e., breakpoints are path-dependent for efficiency. Variables can become symbolic (through assignment of a symbolic value) and concrete (through assignment of a concrete value). Breakpoints are only set for locations where symbolic variables are used.

B. Algorithm overview

The algorithm has two analysis steps. The first step is a path-insensitive static analysis as preparation, in order to determine locations that must be symbolically interpreted (initial debugger breakpoints). Program input is treated as symbolic. This includes the return values of certain standard library functions (these functions can be configured). From these starting points, the static analysis uses inference over the data flow to find out which locations can be reached with these variables. This is often called a taint analysis. Details are described in subsection II-C.

The second step is dynamic symbolic execution, which is a path-sensitive and context-sensitive analysis and therefore needs a constraint solver as logic backend. For an efficient dynamic analysis where variables path-sensitively can become symbolic or concrete, the program locations that need to be symbolically interpreted are also path-sensitive. Conceptually, read/write breakpoints are needed for all symbolic variables. This is implemented by setting breakpoints on all locations where a symbolic variable is used. Breakpoints are adaptively set and removed during analysis. A path can be merged (pruned) during symbolic analysis when the path constraint implies a merge formula for the same location. Merge formulas are generated by backtracking unsat-cores for unsatisfiable error conditions and unsatisfiable program paths. Locations where path merging possibilities are checked (merge locations) are branch nodes in control flow graphs (CFG). Details are described in subsection II-D.

The algorithm overview is also listed as pseude-code in Algorithm 1. The static pre-analysis corresponds to line 1. The depth-first dynamic symbolic execution corresponds to lines 4-37.

```
1  Set{Location} symlocs = findInitialBreakLocations();
2  debugger.setBreaks(symlocs);
3  direction = forward;
4  while (! (direction == exhausted) ) do
5      if (direction == forward) then
6          Location loc = debugger.continue();
7          if (isProgramEnd(loc)) then
8              direction = backtrack;
9              continue;
10         if (mergeLocs.contains(loc)) then
11             if (cansubsume(loc)) then
12                 direction = backtrack;
13                 continue;
14         cfgnode = getNode(loc);
15         interprete(cfgnode);
16     else if (direction == backtrack) then
17         foundNewInputVec = false;
18         while (!foundNewInputVec) do
19             backtrackErrorGuards(cfgnode);
20             if (cfgnode instanceof BranchNode) then
21                 setNewMergeLocation(cfgnode);
22             cfgnode = backtrackLastNode(path);
23             if (isProgramStart(cfgnode)) then
24                 direction = exhausted;
25                 break;
26             if (cfgnode instanceof DecisionNode) then
27                 if (hasOpenBranch(cfgnode)) then
28                     boolean isSat = checkSat(path +
                         openBranch);
29                     if (isSat) then
30                         InputVector newInput =
                             getModel(path + openBranch);
31                         foundNewInputVec = true;
32                     else
33                         uc = getUnsatCore(path +
                             openBranch);
34                         setGuard(openBranch, uc);
35         if (foundNewInputVec) then
36             direction = forward;
37             debugger.restart();
```

Algorithm 1: Dynamic symbolic execution with interpolation based path merging

C. Preparation: path-insensitive extended taint analysis

The static pre-analysis determines for all program locations:

- which variable definitions may reach the location (reaching definitions [10])
- whether a *symbolic* variable might be used (read) at the location; in the following this location property is called 'maybe symbolic'
- whether the location is a potential bug location; This property depends on the bug types that are to be found. For the example of buffer overflow detection, a location is considered a potential bug location when it contains an array subscript expression or a pointer dereference.
- whether the location is a control flow decision node

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9

From this information, it is then determined, for which locations breakpoints must be set for *all* program paths. That is:

- input dependent bug locations: a potential bug location where a symbolic variable might be used. These locations must be symbolically interpreted in order to detect the bugs with a solver satisfiability check or to compute an unsat-core.
- input dependent control flow decisions: input dependent branches must be symbolically interpreted for correct merge formula generation during backtracking and to avoid incorrect path merging. This is described in more detail is subsection II-D.

The analysis is implemented as a monotoneous propagation of changes and uses the worklist algorithm [10]. Source files are parsed into abstract syntax trees (AST), and control flow graphs are computed for all function definitions in the ASTs. When the properties of a control flow node change, the change is propagated to its children (which are then added to the worklist). Since the propagation is monotoneous, the reaching of a fixed-point (empty worklist) is guaranteed. The analysis is not path-sensitive and does not need a constraint solver. It therefore has a better scaling behaviour than symbolic execution.

D. Selective symbolic execution with unsat-core based interpolation

The symbolic execution is essentially a depth-first traversal of the execution tree. As such, it has a forward and a backtracking mode. The backtracking mode generates program input for the next path. The current path is backtracked to the last input-dependent control flow decision. If possible (satisfiable), the last decision is switched to obtain a new path.

1) Forward symbolic execution: The forward symbolic execution mode corresponds to lines 5-15 in Algorithm 1. The debugger is run until it stops at a breakpoint. For this program location, the corresponding CFG node is resolved (line 14). This CFG node is then interpreted and translated into an SMT logic equation. Values of concrete variables are queried from the debugger when needed. More details regarding the translation are provided in the implementation section (Section IV). Functions from the standard library are wrapped, so that input can be traced and forced as desired using debugger commands.

a) Unsat-core interpolation for unsatisfiable bug conditions: Another path can be merged if no new bug detection is possible along any of its extensions, i.e., when potential bug locations remain unsatisfiable for the new path. This is the case when the new path's path constraint implies the unsat-cores of the potential bug locations.

b) Updating for bug detections: When a bug is detected, any new detections of same bug (same type and location) become irrelevant. Therefore, any unsat-cores that were computed for this potential bug location before, can be deleted and the constraints removed from merge formulas. Unsat-cores are computed using the idea of serial constraint deletion from [7]. A path constraint is a conjunction of a set of formulas. For each of these formulas it is checked in turn with the solver, whether the conjunction remains unsatisfiable if the formula is removed. The function is only kept if the conjunction would become satisfiable otherwise. In the following, a computed unsat-core is also called an error guard.

```
1  void CWE121_fgets_12_bad() {
2    int data = -1;
3    if(global_returns_t_or_f()) {
4      char input_buf[ARR_SIZE] = "";       path ⇒ (data=-1) ?
5      if (fgets(input_buf, ARR_SIZE, stdin) !=
         NULL) {
6        data = atoi(input_buf);             path ⇒ ⊤ ?
7      } else {
8        printLine("fgets() failed.");      path ⇒ (data=-1) ?
9      }
10   } else {
11     data = 7;                             path ⇒ ⊤ ?
12   }
13   if(global_returns_t_or_f()) {
14     int i;
15     int buffer[10] = { 0 };              path ⇒ ⊤ ?
16     if (data >= 0) {
17       buffer[data] = 1;
18       for(i = 0; i < 10; i++) {
19         printIntLine(buffer[i]);
20       }
21     } else {
22       printLine("ERROR: Array index
           negative.");
23     }
24   } else {
25     int i;
26     int buffer[10] = { 0 };              path ⇒ ⊤ ?
27     if (data >= 0 && data < (10)) {
28       buffer[data] = 1;
29       for(i = 0; i < 10; i++) {
30         printIntLine(buffer[i]);
31       }
32     } else {
33       printLine("ERROR: index
           out-of-bounds");
34     }
35   }
36 }

37 int global_returns_t_or_f() {
38         return (rand() % 2)
39 }
```

Figure 1. Example function, from the Juliet test suite [9]

c) Path merging: Breakpoints are set during backtracking for branch locations, for which at least one merge formula has been computed. When the current path constraint implies the merge formula, the path is pruned. The implication check uses the solver and corresponds to line 11 in Algorithm 1. The implication is valid if its negation is not satisfiable.

2) Backtracking: The backtracking mode corresponds to lines 16-37 in Algorithm 1. It generates program input for the next path and backtracks error guards to generate merge formulas. Backtracking is only concerned with the partial symbolic program state along the current path, i.e., only with locations which were symbolically interpreted.

a) Input generation: The symbolic program state is backtracked to the last decision node. For child branches that were not yet covered in the context of the current (backtracked) path, it is checked with the solver whether or not this path extension is satisfiable. If it is satisfiable, the solver's model generation functionality is used to generate corresponding program input values for the next path. If not, an unsat-core is computed and the symbolic program state is further backtracked.

b) Unsat-core interpolation for unsatisfiable paths: Because unsatisfiable paths are not further explored, any potential error locations after the unsatisfiable branch are not

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9

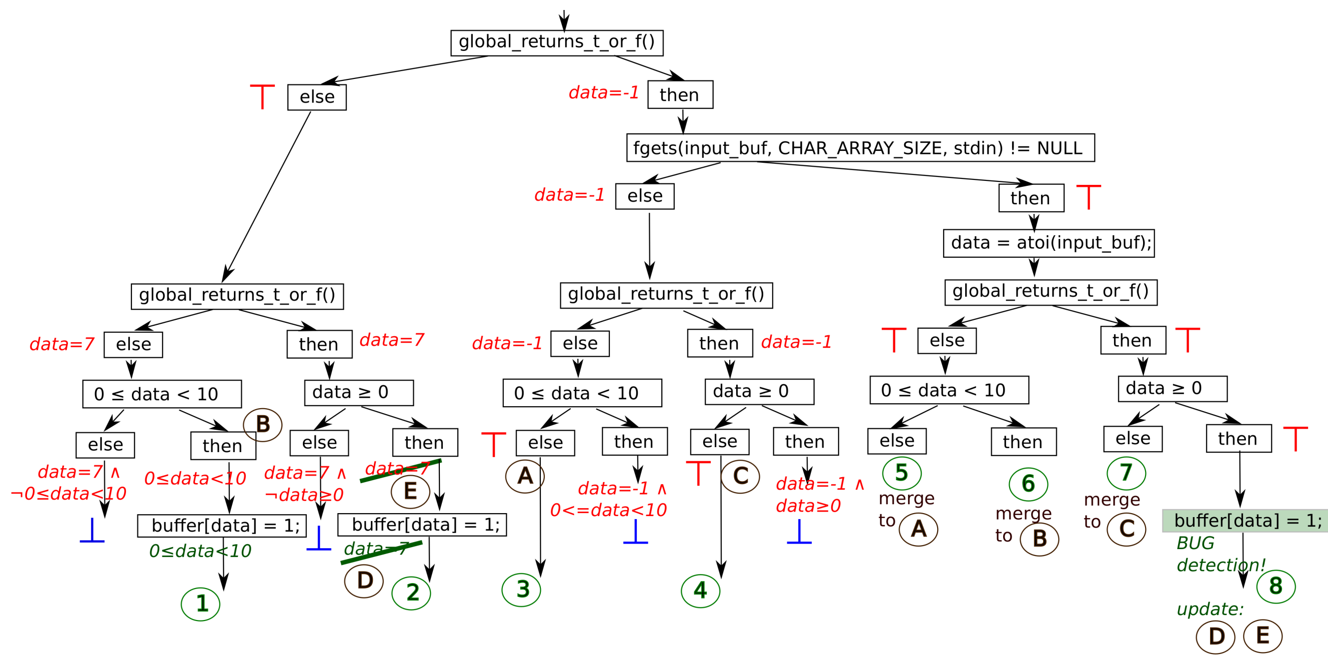

Figure 2. Algorithm progress for the example function from Figure 1

evaluated in this context. Another path can therefore only be merged as long as unsatisfiable branches remain unsatisfiable. This means, that an unsat-core for an unsatisfiable path is also treated as an error guard.

c) Backtracking error guards: Error guards are generated as unsat-cores in forward symbolic execution at the locations of unsatisfiable bugs, or during backtracking at unsatisfiable branch nodes. Backtracking also backtracks these formulas. The conjunction of backtracked error guards for one path (one execution tree node) is a merge formula. When a node's child is backtracked, then any formulas which were generated in this child (as symbolic interpretation) are removed from the node's error guards. Because constraints are removed, backtracking means a generalization of merge formulas. Decision nodes are the only control flow node type that has more than one child node, i.e., several branch nodes. The childrens' contribution to a decision node's error guard is determined during backtracking as the conjunction of the childrens' error guards. The reason is that path merging requires, that no new bug detection becomes possible on *any* extension of the current path. When backtracking reaches a branch node, a breakpoint is set and associated with the merge formula.

E. Example

To illustrate the algorithm, it is applied to the example function shown in Figure 1. The example is a 'bad' function from the Juliet suite [9], which contains a path-sensitive buffer overflow in line 17. The standard library functions `fgets()`, `atoi()` and `rand()` are treated as giving arbitrary (unconstrained) symbolic input. These functions are called in lines 5, 6 and 38. Static pre-analysis additionally yields breakpoints for lines 3, 13, 16 and 27 as input-dependent control flow decisions, and for lines 17 and 28 as input-dependent potential error locations. Together, these lines are indicated as shaded in the figure. Breakpoints are set on these lines, so that the debugger stops there and they are symbolically interpreted. The remaining not shaded locations are always just executed concretely, the debugger is not stopped for them. For the example, we assume that the function is called directly before program end, i.e., there are no backtracked formulas from other functions called later.

Algorithm progress is illustrated in Figure 2. The explored satisfiable program paths are marked with green numbers 1-8 in exploration order. Unsat-cores for unsatisfiable bugs are shown as green formulas (on path 1 and 2). Unsatisfiable branches are marked with a blue 'false' symbol ($\perp$, four times). Backtracked unsat-cores are shown as red formulas next to the respective control flow nodes. Merge locations are the branch nodes, i.e., 'then' and 'else'. The merge formulas are shown in red next to them. The 'true' symbol ($\top$) indicates an empty backtracked unsat-core (7 times). Path merges occur during the exploration of paths 5, 6, and 7. The respective merge targets are marked 'A' to 'C'. The bug is detected on path 8. This potential bug was unsatisfiable on path 2. The respective computed (backtracked) unsat-core is removed, because any re-detection of this bug on another path would be irrelevant. By removing constraints, more path merging can become possible in general. The updated tree nodes are marked 'D' and 'E', the removed constraints are crossed out in green.

Analysis results are also illustrated in Figure 1: merge locations and the corresponding merge conditions (implications) are indicated in red on the right side of the figure. For any further call of this function in the program under analysis, all paths can be merged at latest in lines 15 or 26 (because the respective implications are always valid).

III. A Coverage and Interpolation Hierarchy

Interpolation is an automated generalization of formulas. Through interpolation, interpolated path constraints may become equal. Here, interpolation by removing constraints

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-458-9

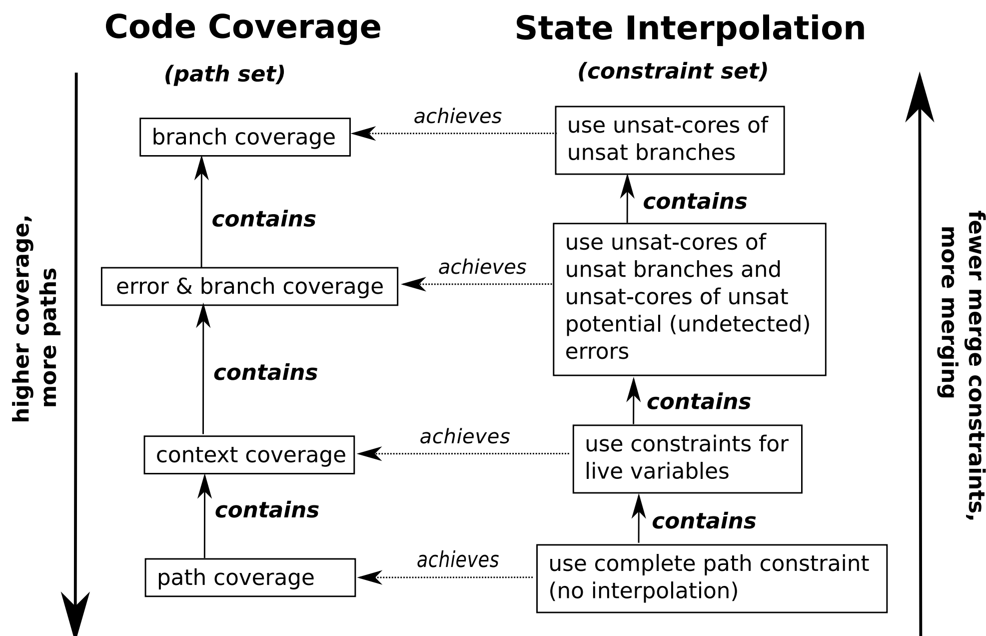

Figure 3. Coverage and interpolation hierarchy

from the path constraint during depth-first path exploration is considered.

Code coverage can be scaled by varying interpolation. Merge formulas are yielded from interpolation. With 'more' interpolation it is meant here to remove more constraints from the path constraint. With fewer constraints in merge formulas, more paths imply the merge formula and are pruned from the execution tree. The achieved coverage is given by the set of remaining paths.

Figure 3 illustrates four interesting algorithms and corresponding coverage. Unsat-cores are not unique. This corresponds to different path sets that can achieve the same coverage criterion. The annotation 'contains' for interpolation constraint sets on the right side of the figure assumes that unsat-cores are computed in the same way with serial constraint deletion.

A. *Branch coverage*

(Backtracked) unsat-cores of unsatisfiable branches are used as merge formulas. A path is only pruned if it implies the previously computed backtracked unsat-cores. This means that any extension of the (pruned) path can not cover any yet uncovered branch. Therefore, this interpolation achieves branch coverage. Branch coverage means that every branch in the program that can be covered with any program input is actually covered.

B. *Error and branch coverage*

This is the algorithm described in Section II. It uses unsat-cores for unsatisfiable branches and additionally unsat-cores of potential (and yet undetected) errors. This comprises unsat-cores necessary to achieve branch coverage. The additional constraints require to only prune a path when all previously unsatisfiable error conditions remain unsatisfiable. This means that any extension of the pruned path can not witness any yet undetected error. Error coverage means that every error that is satisfiable with any program input, and for whose potential existence a constraint is generated, is actually witnessed on a remaining (not pruned) path.

C. *Context coverage*

In backtracking, the sets of dead and live variables are exactly known. Live variables are the ones that are read on at least one extension of the current path. Only live variables can contribute to unsat-cores in a path extension. This interpolation therefore comprises the interpolation needed to achieve error and branch coverage. By removing dead constraints, path constraints can become identical. Context coverage means that every program location is covered in every distinct (live) context.

D. *Path coverage*

Complete path constraints (including constraints for dead variables) are used, no interpolation is done. Every path constraint is different, so no paths are pruned. Depth-first traversal without path merging achieves path coverage, i.e., every satisfiable program path is actually covered. This includes context coverage.

IV. Implementation

The implementation extends previous work, that is described in [11]. This previous work is a dynamic symbolic execution engine for Eclipse CDT, that does not have any path merging functionality.

A. *Dynamic symbolic execution using Eclipse CDT*

This subsection shortly review [11]. The engine is a plug-in extension for CDT's code analysis framework (Codan [12]). It uses CDT's C/C++ parser, AST visitor and debugger services framework (DSF [13]). The DSF is an abstraction layer over the debuggers' machine interfaces that are supported by CDT. The plug-in further uses Codan's CFG builder.

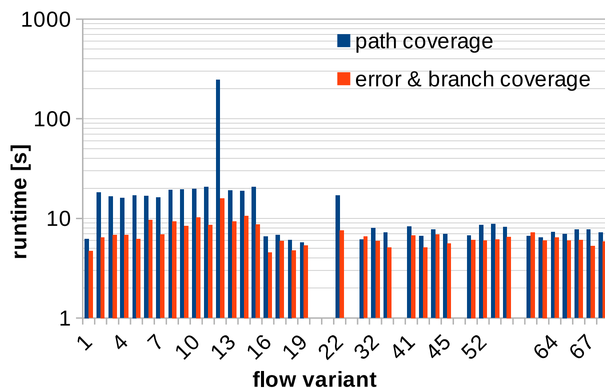

Figure 4. Run-times with and without path merging

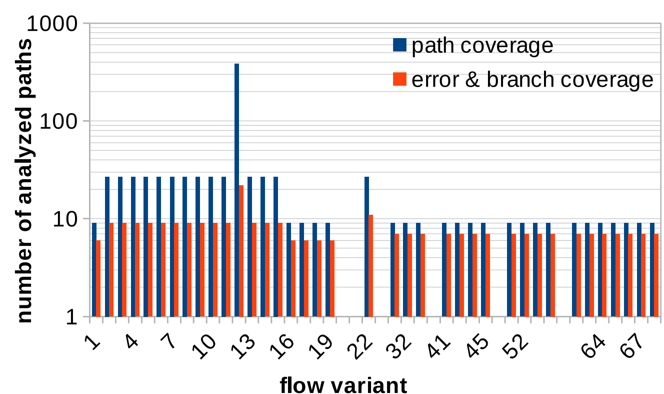

Figure 5. Number of analyzed paths with and without path merging

Initial breakpoints are set on function calls that are configered to return unconstrained symbolic input values. Further breakpoints are set during symbolic execution for locations, where symbolic variables are used. Breakpoints are also set on pointer assignments, because pointer targets might become symbolic through assignment of a symbolic value.

When the debugger stops at a location, the corresponding CFG node is resolved. The AST subtree that corresponds to the CFG node is then interpreted according to the tree based interpreter pattern [14]. The visitor pattern [15] is used to traverse the AST subtree and translate it into an SMT logic equation. SMT queries are formulated in the SMTlib's [16] sublogic of arrays, uninterpreted functions and bit-vectors (AUFBV), and the Z3 SMT solver [17] is used to decide them.

B. *Interpolation and path merging*

Correct merging requires the 'maybe symbolic' static pre-analysis described in section II. Interpolation based path merging means that more breakpoints are set than without merging. On a path that can not be merged, more locations are symbolically interpreted than without merging. The implication check for merging further requires variable projections as described in the following. Single assignments are used in the translation to logic to avoid destructive updates, i.e., single assignment names are used for variables in the logic equations. Because a merge formula was computed on a different path and the translation into logic uses single assignment names, a subset of variable names in both formulas (merge formula and path constraint) has to be substituted. These variable names are the last single assignment versions in both formulas of variables whose definitions reach the merge location. These variables are projected (substituted) to the corresponding syntax tree names (names in the source code). Because branch nodes are not necessarily explicit in the source node, the merge locations are not exactly branch nodes, but rather the next following program location where a debugger breakpoint can be set. This is the following expression or declaration with initializer. Merge location examples are given in Figure 1. The computation of unsat-cores with serial constraint deletion is straight-forward.

V. EXPERIMENTS

The implementation is evaluated with 39 buffer overflow test programs from the Juliet suite (buffer overflows with `fgets()`) that cover Juliet's different control and data flow

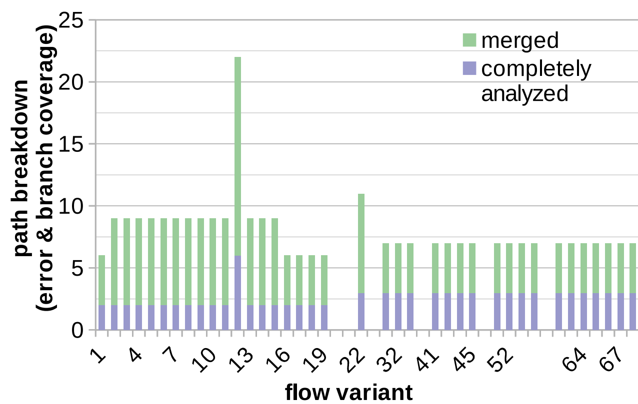

Figure 6. Breakdown of the number of analyzed paths with merging

variants for C. The test programs are analysed with the Eclipse plug-in, as JUnit plug-in tests. Eclipse version 4.5 is used on a i7-4650U CPU on 64-bit Linux kernel 3.16.0, with GNU debugger `gdb` version 7.7.1. The presented algorithm for error and branch coverage is compared with straight-forward dynamic symbolic execution without any merging (i.e., path coverage), as described in [11].

The results are shown in Figures 4, 5 and 6. The horizontal axes show the flow variant number. Juliet's flow variants are not numbered consecutively, to allow for later insertions in future test suite verions. Both the presented algorithm and the path coverage algorithm accurately detect the contained errors for all flow variants except for flow variant 21. In this flow variant (control flow depends on a static global variable), CDT's CFG builder falsely classifies a branch node as dead node, which leads to false negative detection.

A. *Speedup with merging*

The measured run-times both with and without path merging are shown in Figure 4. The figure's time scale is logarithmic. Despite of the additional analyses for path merging, there is a clear speed-up for all test cases. The biggest speed-up is achieved for flow variant 12, which also contains the largest number of satisfiable program paths.

B. Reduction in the number of analyzed paths

Figure 5 shows the number of analyzed paths with path coverage on the one hand and with error and branch coverage on the other. There is a clear reduction in the number of analyzed paths for all test programs. Merging prunes a subtree, which in general splits into more than one satisfiable program path. The figure shows a strong correlation with Figure 4, so that the reduction in the number of analyzed paths can be seen as the main reason for the speed-up.

C. Proportion of merged paths

Figure 6 shows a breakdown of the analyzed paths for merging only (error and branch coverage). The analyzed paths are distinguished into paths, that are completely analyzed until program end, and others, that are merged at some point. The figure shows that for all test cases the majority of analyzed paths is merged at some point, which is an additional reason for speed-up and explains another part of it (with the reduction of the analyzed lengths of the paths that are merged).

VI. Related Work

Work on symbolic execution spans over 30 years. An overview is given in [18]. Dynamic symbolic execution is presented in [3]. The concept of selective symbolic execution is described in [5]. The implementation uses breadth-first execution tree traversal without path merging. Path merging based on live variable analysis is presented in [6]. Path merging based on interpolation using unsat-cores is described in [7]. The latter approach is more comprehensive. It comprises elimination of constraints for dead variables, because those are not present in backtracked formulas. The interpolation based merging approach is used in a static symbolic execution tool for verification [19].

The work at hand differs in that it combines dynamic and selective symbolic execution with interpolation based path merging. Further, the efficient application to testing requires that merge conditions are updated in case of a bug detection, whereas a verification tool may terminate at the first error detection (when verification fails). The work at hand builds on the author's previous work described in [20], which performs static symbolic execution with path merging based on a live variables analysis. As already mentioned, it also builds on own previous work described in [11], which performs dynamic symbolic execution without any path merging.

VII. Discussion

Interpolation based path merging with the presented algorithm for error and branch coverage shows a clear speed-up already for the tiny Juliet test programs. Due to the improved scaling behaviour, it is expected to lead to increasing speed-ups for larger programs. Future work might include loop subsumption and the detection of infinite loops. Another point is the addition of checkers for different bug types.

Acknowledgement

This work was funded by the German Ministry for Education and Research (BMBF) under grant 01IS13020.

References

[1] J. King, "Symbolic execution and program testing," Communications of the ACM, vol. 19, no. 7, 1976, pp. 385–394.

[2] L. deMoura and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," Communications of the ACM, vol. 54, no. 9, 2011, pp. 69–77.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in Conference on Programming Language Design and Implementation, 2005, pp. 213–223.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in European Software Engineering Conference and International Symposium on Foundations of Software Engineering, 2005, pp. 263–272.

[5] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in Int. Conf. Architectural Support for Programming Languages and Operating Systems, 2011.

[6] P. Boonstoppel, C. Cadar, and D. Engler, "RWset: Attacking path explosion in constraint-based test generation," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 351–366.

[7] J. Jaffar, A. Santosa, and R. Voicu, "An interpolation method for CLP traversal," in Int. Conf. Principles and Practice of Constraint Programming (CP), 2009, pp. 454–469.

[8] W. Craig, "Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory," The Journal of Symbolic Logic, vol. 22, no. 3, 1957, pp. 269–285.

[9] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," IEEE Computer, vol. 45, no. 10, 2012, pp. 88–90.

[10] F. Nielson, H. Nielson, and C. Hankin, Principles of Program Analysis. Springer, 2010.

[11] A. Ibing, "Dynamic symbolic execution using Eclipse CDT," in Int. Conf. Software Engineering Advances, 2015, in press.

[12] E. Laskavaia, "Codan- a code analysis framework for CDT," in EclipseCon, 2015.

[13] P. Piech, T. Williams, F. Chouinard, and R. Rohrbach, "Implementing a debugger using the DSF framework," in EclipseCon, 2008.

[14] T. Parr, Language Implementation Patterns. Pragmatic Bookshelf, 2010.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[16] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB standard version 2.0," in Int. Workshop Satisfiability Modulo Theories, 2010.

[17] L. deMoura and N. Bjorner, "Z3: An efficient SMT solver," in Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2008, pp. 337–340.

[18] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," Communications of the ACM, vol. 56, no. 2, 2013, pp. 82–90.

[19] J. Jaffar, V. Murali, J. Navas, and A. Santosa, "TRACER: A symbolic execution tool for verification," in Int. Conf. Computer Aided Verification (CAV), 2012, pp. 758–766.

[20] A. Ibing, "A backtracking symbolic execution engine with sound path merging," in Int. Conf. Emerging Security Information, Systems and Technologies, 2014, pp. 180–185.