Efficient State Merging in Symbolic Execution

Volodymyr Kuznetsov Johannes Kinder Stefan Bucur George Candea

School of Computer and Communication Sciences
École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
{vova.kuznetsov,johannes.kinder,stefan.bucur,george.candea}@epfl.ch

Abstract

Symbolic execution has proven to be a practical technique for building automated test case generation and bug finding tools. Nevertheless, due to state explosion, these tools still struggle to achieve scalability. Given a program, one way to reduce the number of states that the tools need to explore is to merge states obtained on different paths. Alas, doing so increases the size of symbolic path conditions (thereby stressing the underlying constraint solver) and interferes with optimizations of the exploration process (also referred to as search strategies). The net effect is that state merging may actually lower performance rather than increase it.

We present a way to automatically choose when and how to merge states such that the performance of symbolic execution is significantly increased. First, we present *query count estimation*, a method for statically estimating the impact that each symbolic variable has on solver queries that follow a potential merge point; states are then merged only when doing so promises to be advantageous. Second, we present *dynamic state merging*, a technique for merging states that interacts favorably with search strategies in automated test case generation and bug finding tools.

Experiments on the 96 GNU COREUTILS show that our approach consistently achieves several orders of magnitude speedup over previously published results. Our code and experimental data are publicly available at `http://cloud9.epfl.ch`.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.2.5 [*Software Engineering*]: Testing and Debugging – Symbolic Execution, Testing Tools; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Testing, Symbolic Execution, Verification, Bounded Software Model Checking, State Merging

1. Introduction

Recent tools [5–7, 18, 19] have applied symbolic execution to automated test case generation and bug finding with impressive results—they demonstrate that symbolic execution brings unique practical advantages. First, such tools perform dynamic analysis, in that they actually execute a target program and can directly execute any calls to external libraries or the operating system by concretizing arguments; this broadens their applicability to many real-world programs. Second, these tools share with static analysis the ability to simultaneously reason about multiple program behaviors, which improves the degree of completeness they achieve. Third, symbolic execution does not use abstraction but is fully precise with respect to predicate transformer semantics [11]; it generates "per-path verification conditions" whose satisfiability implies the reachability of a particular statement, so it generally does not have false positives. Fourth, recent advances in SAT and SMT (SAT Modulo Theory) solving [10, 12, 15] have made tools based on symbolic execution significantly faster. Overall, symbolic execution promises to help solve many important, practical program analysis problems.

Nevertheless, today's symbolic execution engines still struggle to achieve scalability, because of path explosion: the number of possible paths in a program is generally exponential in its size. States in symbolic execution encode the history of branch decisions (the *path condition*) and precisely characterize the value of each variable in terms of input values (the *symbolic store*), so path explosion becomes synonymous with state explosion. Alas, the benefit of not having false positives in bug finding (save for over-approximate environment assumptions) comes at the cost of having to analyze an exponential number of states.

One way to reduce the number of states is to merge states that correspond to different paths. This is standard in classic static analysis, where the resulting merged state over-approximates the individual states that were merged. Several techniques, such as ESP [9] and trace partitioning [27], reduce but do not eliminate the resulting imprecision (which can be a source of false positives) by associating separate abstract domain elements to some sets of execution paths. In symbolic execution, as a matter of principle, a merged state would have to precisely represent the information from *all* execution paths *without any* over-approximation. Consider, for example, the program *if (x<0) {x=0;} else {x=5;}* with input X assigned to x. We denote with (pc, s) a state that is reachable for inputs obeying path condition pc and in which the symbolic store $s = [v_0 = e_0, \ldots, v_n = e_n]$ maps variable v_i to expression e_i, respectively. In this case, the two states $(X < 0, [x = 0])$ and $(X \geq 0, [x = 5])$, which correspond to the two feasible paths, can be merged into one state $(\text{true}, [x = \text{ite}(X < 0, 0, 5)])$. Here, $\text{ite}(c, p, q)$ denotes the if-then-else operator that evaluates to p if c is true, and to q otherwise. If states were merged this way for every branch of a program, symbolic execution would become similar to verification condition generation or bounded model checking, where the entire problem instance is encoded in one monolithic formula that is passed in full to a solver.

State merging effectively decreases the number of paths that have to be explored [16, 20], but also increases the size of the symbolic expressions describing variables. Merging introduces disjunctions, which are notoriously difficult for SMT solvers, particularly for those using eager translation to SAT [15]. Merging also converts differing concrete values into symbolic expressions, as in the example above: the value of x was concrete in the two separate states, but symbolic $(\text{ite}(X < 0, 0, 5))$ in the merged state. If x were to ap-

PLDI'12, June 11–16, Beijing, China.

pear in branch conditions or array indices later in the execution, the choice of merging the states may lead to more solver invocations than without merging. This combination of larger symbolic expressions and extra solver invocations can drown out the benefit of having fewer states to analyze, leading to an actual *decrease* in the overall performance of symbolic execution [20].

Furthermore, state merging conflicts with important optimizations in symbolic execution: search-based symbolic execution engines, like the ones used in test case generators and bug finding tools, employ *search strategies* to prioritize searching of "interesting" paths over "less interesting" ones, e.g., with respect to maximizing line coverage given a fixed time budget. To maximize the opportunities for state merging, however, the engine would have to traverse the control flow graph in topological order, which typically contradicts the strategy's path prioritization policy.

Contributions. In this paper, we describe a solution to these two challenges that yields a net benefit in practice. We combine the state space reduction benefits of merged exploration with the constraint solving benefits of individual exploration, while mitigating the ensuing drawbacks. Experiments on the GNU COREUTILS show that employing our approach in a symbolic execution engine achieves speedups over the state of the art that are exponential in the size of symbolic input. Our contributions are:

- We present *query count estimation*, a way to statically approximate the number of times each variable will appear in future solver queries after a potential merge point. We then selectively merge two states only when we expect differing variables to appear infrequently in later solver queries. Since this selective merging merely groups paths instead of pruning them, inaccuracies in the estimation do not hurt soundness or completeness.
- We present *dynamic state merging*, a merging algorithm specifically designed to interact favorably with search strategies. The algorithm explores paths independently of each other and uses a similarity metric to identify on-the-fly opportunities for merging, while preserving the search strategy's privilege of dictating exploration priorities.

Organization. In §2, we characterize the design space of precise symbolic analysis using a generic algorithm. With bounded model checking on one end of the spectrum and symbolic execution on the other, we analyze middle-ground approaches (such as function summaries [16]) and argue for an opportunistic, dynamic approach to navigating this design space based on cost estimates. In this context, we introduce query count estimation (§3) and dynamic state merging (§4). We then describe our implementation in the KLEE symbolic execution engine and present a systematic evaluation of the individual and combined effects of the two proposed methods (§5). We review related work in §6 and conclude with §7.

2. Trade-offs in Symbolic Program Analysis

Test generation by symbolic execution is just one of a multitude of precise symbolic program analyses that are facilitated by SAT or SMT solvers. Tools such as CBMC [8], Saturn [31], and Calysto [2] have shown that exact, abstraction-free path sensitive local reasoning is feasible and can be fully outsourced to an external solver. With the help of a generic worklist algorithm (§2.1), we illustrate the relationship among precise symbolic program analyses and explain the trade-offs in the resulting solver queries (§2.2). We conclude that state merging critically affects all types of precise symbolic program analysis. Motivated by this insight, we give a brief overview of our proposed approach to state merging (§2.3).

2.1 General Symbolic Exploration

Precise symbolic program analyses essentially perform forward expression substitution starting from a set of input variables. The resulting formulae are then used to falsify assertions and find bugs, or to generate input assignments and generate test cases. Algorithm 1 is a generic algorithm for symbolic program analysis that can be used to implement different analysis flavors. For illustration purposes, we consider only a simple input language with assignments, conditional goto statements, assertions, and halt statements.

Input: Choice function *pickNext*, similarity relation $\sim$, branch checker *follow*, and initial location ℓ_0.
Data: Worklist w and set of successor states S.

```
1  w := {(ℓ0, true, λv.v)};
2  while w ≠ ∅ do
3      (ℓ,pc,s) := pickNext(w); S := ∅;
       // Symbolically execute the next instruction
4      switch instr(ℓ) do
5          case v := e                                  // assignment
6              S := {(succ(ℓ),pc,s[v ↦ eval(s,e)])};
7          case if(e) goto ℓ'                           // conditional jump
8              if follow(pc ∧ s ∧ e) then
9                  S := {(ℓ',pc ∧ e,s)};
10             if follow(pc ∧ s ∧ ¬e) then
11                 S := S ∪ {(succ(ℓ),pc ∧ ¬e,s)};
12         case assert(e)                               // assertion
13             if isSatisfiable(pc ∧ s ∧ ¬e) then abort;
14             else S := {(succ(ℓ),pc,s)};
15         case halt                                    // program halt
16             print pc;
       // Merge new states with matching ones in w
17     forall (ℓ'',pc',s') ∈ S do
18         if ∃(ℓ'',pc'',s'') ∈ w : (ℓ'',pc'',s'') ∼ (ℓ'',pc',s') then
19             w := w \ {(ℓ'',pc'',s'')};
20             w := w ∪ {(ℓ'',pc' ∨ pc'', λv.ite(pc',s'[v],s''[v]))};
21         else
22             w := w ∪ {(ℓ'',pc',s')};
23 print "no errors";
```

Algorithm 1. Generic symbolic exploration.

The algorithm is parameterized by a function *pickNext* for choosing the next state in the worklist, a function *follow* that returns a decision on whether to follow a branch, and a relation $\sim$ that controls whether states should be merged. We now extend the notation for states used in §1 to triples (ℓ, pc, s), consisting of a program location ℓ, the path condition pc, and the symbolic store s that maps each variable to either a concrete value or an expression over input variables. In line 1, the worklist w of the algorithm is initialized with a state whose symbolic store maps each variable to itself (for simplicity, we exclude named constants). Here, $\lambda x.e$ denotes the function mapping parameter x to an expression e (we will use $\lambda(x_1, \ldots x_n).e$ for multiple parameters). In each iteration, the algorithm picks a new state from the worklist (line 3).

On encountering an assignment $v := e$ (lines 5-6), the algorithm creates a successor state at the fall-through successor location $succ(\ell)$ of ℓ by updating the symbolic store s with a mapping from v to a new symbolic expression obtained by evaluating e in the context of s, and adds the new state to the set S. At every branch (lines 7-11), the algorithm first checks whether to follow either path and, if so, adds the corresponding condition to the successor state, which in turn is added to S. Analyses can decide to not follow a branch if the branch is infeasible or would exceed a limit on loop unrolling. For assertions (line 12-14), the path condition, the symbolic store, and the negated assertion are put in conjunction

and checked for satisfiability. Since the algorithm does not over-approximate, this check has no false positives. Halt statements terminate the analyzed program, so the algorithm just outputs the path condition, a satisfying assignment of which can be used to generate a test case for the execution leading to the halt.

In lines 17-22, the new states in S are then merged with any matching states in the worklist before being added to the worklist themselves. Two states match if they share the same location and are similar according to $\sim$. Merging creates a disjunction of the two path conditions (which can be simplified by factoring out common prefixes) and builds the merged symbolic store from ite expressions that assert one or the other original value, depending on the path taken (line 20). The ite expressions that assert an identical value in both cases (because it was equal in both symbolic stores) can be simplified to that value.

2.2 The Design Space of Symbolic Program Analysis

The differences between various implementations of precise symbolic analysis lie in the following aspects:

1. the handling of loops and/or recursion;
2. whether and how the feasibility of individual branches is checked to avoid encoding infeasible paths;
3. whether and how states from different paths are merged;
4. compositionality, i.e., the use of function summaries.

Loops affect soundness and completeness, while the other aspects are trade-offs that critically affect analysis performance. We now illustrate these different aspects using Algorithm 1.

Loops and Recursion. Bounded model checkers [8] and extended static checkers [2, 13, 31] unroll loops up to a certain bound, which can be iteratively increased if an injected unwinding assertion fails. Such unrolling is usually performed by statically rewriting the CFG, but can be fit into Algorithm 1 by defining *follow* to return false for branches that would unroll a loop beyond the bound. Symbolic execution explores loops as long as it cannot prove the infeasibility of the loop condition. A search strategy, implemented in the function *pickNext*, can bias the analysis against states that perform many repetitions of the same loop. For example, a search strategy optimized for line coverage selects states close to unexplored code and avoids states in deep loop unrollings [6].

Dynamic test generation as implemented in DART [18] starts with an arbitrary initial unrolling of the loop and explores different unrollings in subsequent tests. That is, DART implements *pickNext* to follow concrete executions, postponing branch alternatives until they are covered by a subsequent concrete execution.

All these approaches essentially perform loop unrolling and are generally incomplete for finite analysis times. Loop invariants are rarely used (though [17] is an exception) since weak invariants can introduce false positives, which these precise analyses are specifically designed to avoid. Weakest precondition-based program verification engines such as Boogie [26] and Havoc [24], which also interface with external solvers, rely on the user to supply sufficiently strong invariants for proving all properties of interest.

Feasibility Checking. While performing expression substitution along individual paths, certain combinations of conditional branches can turn out to be infeasible. Not propagating states that represent infeasible paths helps to reduce path explosion by investing solving time earlier in the execution. Intermediate feasibility checks are usually performed only by symbolic execution engines that follow a single path at a time (when reasoning about groups of paths, branches are less likely to be infeasible), in which case *follow* simply invokes the constraint solver.

State Merging. When states meet at the same control location, there are two general possibilities for combining their information: either the states are maintained separately, or the states are merged into a single state. In precise symbolic analysis, merging is not allowed to introduce abstraction. From a conceptual viewpoint, state merging therefore only changes the shape of a formula that characterizes a set of execution paths: if states are kept separate, a set of paths is described by their disjunction; if states are merged, there is only one formula with disjunctions in the path condition and ite expressions in the symbolic store that guard the values of the variables depending on the path taken.

In general, we distinguish two extremes: (i) complete separation of paths, as implemented by *search-based symbolic execution* (e.g., [4–6, 18, 19, 23]), and (ii) complete *static state merging*, as implemented by verification condition generators (e.g., [2, 8, 21, 31]). Static state merging combines states at join points after completely encoding all subpaths, i.e., it defines *pickNext* to explore all subpaths leading to a join point before picking any states at the join point, and it defines $\sim$ to contain all pairs of states. In search-based symbolic execution engines, *pickNext* can be chosen freely according to the search goal, and $\sim$ is empty. Thus, they can, for example, choose to explore just the successors of a specific state and delay exploration of additional loop iterations.

Some approaches adopt intermediate merging strategies. In the context of bounded model checking (BMC), Ganai and Gupta [14] investigate splitting the verification condition along subsets of paths. This moves BMC a step into the direction of symbolic execution, and corresponds to partitioning the $\sim$ relation. Hansen et al. [20] describe an implementation of static state merging in which they modify the exploration strategy to effectively traverse the CFG in topological order and merge all states that share the same program location. For two of their three tested examples, the total solving time increases with this strategy thus showing this approach to be sub-optimal. Another prominent example of state merging is the use of function summaries in symbolic execution, which we explain below.

Compositionality. For precise interprocedural symbolic execution, the simplest and most common approach is function inlining. This causes functions to be re-analyzed at every call site, which could be avoided using function summaries. Summaries that do not introduce abstraction and are thus suitable for symbolic execution can be implemented by computing an intraprocedural path condition in terms of function inputs, and then merging all states at the function exit.

Alas, applying such a function summary is essentially as expensive as re-analyzing the function, if the translation effort from the programming logic into the representation logic is negligible. Using a summary instead of inlining avoids only the feasibility checks for intraprocedural paths that are infeasible regardless of the function input. The cost of the other feasibility checks that a non-compositional symbolic execution would perform is not eliminated by function summaries. Instead, the branch conditions are contained in the ite expressions of the summary and will increase the complexity of later SMT queries.

For dynamic test generation, Godefroid [16] suggests to collect summaries as disjunctions of pairs of input and output constraints. In further work [1], this is extended to record summaries one path at a time and to apply partial summaries whenever they match the input preconditions. Dynamic test generation re-executes the full program (with heavy instrumentation) for each branch of which the alternate case is to be analyzed. Due to re-execution, analyzing all branches in functions would come at an especially high cost, so the savings outweigh the additional solving costs for the merged summary states.

For simplicity, Algorithm 1 is just intraprocedural, supporting function calls by inlining. It can generate precise symbolic function summaries, if invoked per procedure and with a similarity relation that merges all states when the function terminates.

2.3 Our Approach: Dynamically Navigate the Design Space

Precise symbolic analyses are roughly equivalent in their treatment of loops, but all other design choices (merging at control points, feasibility checking, and function summaries) boil down to choosing which paths to analyze separately vs. which ones to combine into common formulae. In other words, all analyses lie along a spectrum, with search-based symbolic execution (no state merging) at one extreme and whole-program verification condition generation (static state merging) at the other extreme. Instead of making a static design choice of where to be in this spectrum, our approach is to enable a symbolic analysis to choose dynamically the most advantageous point and merge states according to the expected benefit of such a merge.

On the one hand, merging reduces the number of states, but on the other hand, the remaining states become more expensive to explore. In the merged state, each variable that had distinct values in the original states must be constrained by an input-dependent ite expression, which increases the time required to solve future queries involving such variables. If the distinct values were concrete, merging would cause additional solver invocations where expressions could have been evaluated concretely in separate states.

Furthermore, there is an inherent incompatibility between partial searches using coverage-guided search strategies (as is often done in test generation) and static state merging at control flow join points: merging can be maximized by exploring the CFG in topological order, so that a combined state can be computed from its syntactic predecessors. A coverage-guided search strategy, however, will dynamically deprioritize some states (e.g., defer for later the exploration of additional iterations of a loop), which prevents using a topological order.

Therefore, to make state merging practical, we must solve two problems: (1) Automatically identify an advantageous balance between exploring fewer complex states vs. more simpler states, and merge states only when this promises to reduce exploration time; and (2) Efficiently combine state merging with search strategies that deprioritize "non-interesting" execution paths.

To solve the first problem, we developed *query count estimation* (QCE), a way to estimate how variables that are different in two potentially mergeable states will be used in the future. We preprocess the program using a lightweight static analysis to identify how often each variable is used in branch conditions past any given point in the CFG, and use this as a heuristic estimate of how many subsequent solver queries that variable is likely to be part of. Using this heuristic, we check whether two states are sufficiently similar that merging them would yield a net benefit. That is, the additional cost of solving more and harder SMT queries is outweighed by the savings from exploring fewer paths. The results of this static analysis affect only the completion time of the symbolic analysis—not its soundness or completeness.

To solve the second problem, we introduce *dynamic state merging* (DSM), a way to dynamically identify opportunities for merging regardless of the exploration order imposed by the search strategy. Without any restrictions on the search strategy, only states that meet at the same location by chance could ever be merged. To increase the opportunities for merging, we maintain a bounded history of the predecessors of the states in the worklist. When picking the next state to process from the worklist, we check whether some state a_1 is similar to a predecessor a_2' of another state a_2 in the worklist. If yes, then state a_1, which is in some sense lagging behind a_2, is prioritized over the others. This causes it to be temporarily *fast-forwarded*, until its own successor matches up with the candidate-for-merging state a_2. If the state diverges, i.e., one of a_1's successors is no longer sufficiently similar to a predecessor of a_2, the merge attempt is abandoned. Thus, while the search strategy is still in control, DSM identifies merge opportunities dynamically within a fixed distance and only briefly takes over control to attempt the merge. After the merge attempt, the search strategy continues as before. Like QCE, DSM does not affect soundness or completeness of the symbolic analysis.

```
1 void main(int argc, char **argv) {
2   int r = 1, arg = 1;
3   if (arg < argc)
4     if (strcmp(argv[arg], "-n") == 0) {
5       r = 0; ++arg;
6     }
7   for (; arg < argc; ++arg)
8     for (int i = 0; argv[arg][i] != 0; ++i)
9       putchar(argv[arg][i]);
10  if (r)
11    putchar('\n');
12 }
```

Figure 1. Simplified version of the `echo` program.

We combine the solutions to these two problems by using QCE (explained in detail in §3) to compute the similarity relation used by DSM (§4). Even though we initially developed these techniques to improve the performance of search-based symbolic execution engines for test generation, we believe our analysis and the insights into building efficiently solvable symbolic formulae from programs are applicable to other symbolic program analyses as well.

3. Query Count Estimation

We now illustrate the need for estimating the expected benefit of merging using an example (§3.1), show how to compute the query count estimates (§3.2), and then justify our decisions (§3.3).

3.1 Motivating Example

Consider the example program in Figure 1, a simplified version of the UNIX `echo` utility that prints all its arguments to standard output, except for argument 0, which holds the program name. If the first regular argument is `"-n"`, no newline character is appended. We analyze this program using Algorithm 1, assuming bounded input. Specifically, we assume that $\texttt{argc} = N+1$ for some constant $N \geq 1$, and that each of the N command-line arguments, pointed to by the corresponding element of `argv`, is a zero-terminated string of up to L characters. For simplicity, we assume that `strcmp` and `putchar` do not split paths. Under these preconditions, the total number of feasible program paths is $L^N + L^{N-1}$, and the branch condition at line 3 is always true.

The execution paths first split at line 4 on the condition C that *argv*[1] points to the string "`-n`". Line 6 is then reached by the two states $(6, C, [\texttt{r} = 0, \texttt{arg} = 2])$ and $(6, \neg C, [\texttt{r} = 1, \texttt{arg} = 1])$. These two can be merged into the single (but fully precise) state $(6, \text{true}, [\texttt{r} = \text{ite}(C,0,1), \texttt{arg} = \text{ite}(C,2,1)])$. Consider now the loop condition $\texttt{arg} < \texttt{argc}$ in line 7. If the states were kept separate, this condition could be evaluated concretely in both states, as $1 < N+1$ and $2 < N+1$, respectively. In the merged state, however, the condition would become the disjunctive expression $\text{ite}(C,2,1) < N+1$, which now requires a solver invocation where it was not previously necessary. The consequences of having merged at line 6 become even worse later in the execution, for the condition at line 8. The array index is no longer concrete, so the SMT solver is required to reason about symbolic memory accesses in the theory of arrays on every iteration of the nested loop. In this example, merging reduces the total number of states, but the merged state is more expensive to reason about. Our experiments confirm that the total time required to fully explore all feasible paths in this program is significantly shorter if the paths are *not* merged on line 6.

Now consider the branching point in the inner loop header at line 8. Since this loop may be executed up to L times, each state

that enters the loop creates L successor states, one for each loop exit possibility. For example, a state exiting after the second iteration is $(8, \ldots \wedge \texttt{argv}[1][0] \neq 0 \wedge \texttt{argv}[1][1] = 0, [\ldots, \texttt{i} = 1])$. On the next iteration of the outer loop (line 7), each of these L states again spawns L successors. At the end of the N outer loop iterations, there is a total of L^N states. However, all of the states created in the loop at line 8 during the same iteration of the outer loop differ only in the value of the temporary variable `i`, which is never used again in the program. Therefore, merging these states does not increase the cost of subsequent feasibility checks, yet it cuts the number of states after the outer loop down to the number of states before the loop (2 in our example). Note that, while the path condition of the merged state is created as a disjunction, here it can be simplified to the common prefix of all path conditions.

There is another, less obvious, opportunity for merging states. Looking back at the first feasible branch at line 4, consider the state $(7, C, [\texttt{r} = 0, \texttt{arg} = 2])$, which corresponds to the path through the "then" branch, and the state $(7, \neg C, [\texttt{r} = 1, \texttt{arg} = 2])$, which corresponds to the path through the "else" branch and one first iteration over the outer loop. Merging these two states yields the state $(7, \text{true}, [\texttt{r} = \text{ite}(C, 0, 1), \texttt{arg} = 2])$, which introduces a disjunction for the symbolic expression representing the value of the variable `r`. Unlike the `arg` variable we discussed above, `r` is used only once on line 10, just before the program terminates. Therefore, the time saved by exploring the loops at lines 7-9 with fewer states can outweigh the cost of testing the more complex branch condition on line 10 in the merged state.

This example demonstrates that the net benefit of merging two states depends heavily on how often variables whose values differ between two states affect later branch conditions. This is the key insight behind QCE, which we explain next.

3.2 Computing the Heuristic

To make an exact merging decision, one would have to compute the cumulative solving times for both the merged and unmerged cases. But this is impractical, so the query count estimation heuristic (QCE) makes several simplifications that allow it to be largely pre-computed before symbolic execution begins. QCE can be calibrated using a number of parameters, which we denote using the Greek letters α, β, and κ.

At each program location ℓ, QCE pre-computes a set $H(\ell)$ of "hot variables" that are likely to cause many queries to the solver if they were to contain symbolic values. The heuristic is to avoid introducing new symbolic values for these "hot variables". Specifically, states should be merged only if every hot variable either has the same concrete value in both states or is already symbolic in at least one of the states. Formally, QCE is implemented by defining the similarity relation $\sim$ of Algorithm 1 as

$$(\ell, pc_1, s_1) \sim_{qce} (\ell, pc_2, s_2) \iff \forall v \in H(\ell) : s_1[v] = s_2[v] \vee I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v], \quad (1)$$

where $I \blacktriangleleft s[v]$ denotes that variable v has a symbolic value in the symbolic store s (i.e., it depends on the set of symbolic inputs I).

In order to check whether a variable v is hot at location ℓ, QCE estimates the number of additional queries $Q_{add}(\ell, v)$ that would be executed after reaching ℓ if variable v were to be made symbolic. Variable v is determined to be hot if this number is larger than a fixed fraction α of the total number of queries $Q_t(\ell)$ that will be executed after reaching ℓ:

$$H(\ell) = \{v \in V \mid Q_{add}(\ell, v) > \alpha \cdot Q_t(\ell)\} \quad (2)$$

To estimate these numbers of queries efficiently, we assume that every executed conditional branch leads to a solver query with a fixed probability (which could be taken into account by suitably adjusting the value of α), and that each branch is feasible with a fixed probability β. Consider function q that descends recursively into the control flow graph counting the number of queries that are selected by a function c:

$$q(\ell', c) = \begin{cases} \beta \cdot q(succ(\ell'), c) + \beta \cdot q(\ell'', c) + c(\ell', e) & instr(\ell') = \mathsf{if}(e)\ \mathsf{goto}\ \ell'' \\ 0 & instr(\ell') = \mathsf{halt} \\ q(succ(\ell'), c) & \text{otherwise} \end{cases} \quad (3)$$

Then $Q_{add}(\ell, v)$ and $Q_t(\ell)$ can be computed recursively as follows:

$$Q_{add}(\ell, v) = q\big(\ell, \lambda(\ell', e).\, \text{ite}\left((\ell, v) \triangleleft (\ell', e), 1, 0\right)\big)$$
$$Q_t(\ell) = q\big(\ell, \lambda(\ell', e).\, 1\big), \quad (4)$$

where $(\ell, v) \triangleleft (\ell', e)$ denotes the fact that expression e at location ℓ' may depend on the value of variable v at location ℓ. For the sake of simplicity, we assume all program loops to be unrolled and all function calls to be inlined. For loops (and recursive function calls) whose number of iterations cannot be determined statically, QCE assumes a fixed maximum number of iterations κ.

Note that the implementation of QCE is limited to estimating the number of additional queries without taking into account the fact that queries may become more expensive due to ite expressions. Our evaluation shows that this suffices in most cases, but we also found a few cases in which lifting this limitation would improve our results. The justification of QCE in §3.3 describes how to integrate the cost of ite expressions in the computation.

Interprocedural QCE. In our implementation, we avoid the assumption of inlined functions by computing $Q_{add}(\ell, v)$ and $Q_t(\ell)$ for all function entry points ℓ as function summaries. We do this compositionally, by computing per-function *local* query counts in a bottom-up fashion. The local query counts for a function `F` include all queries issued inside `F` and all functions called by `F`. To compute these, we extend Equation (3) to handle function calls. At every call site, the local query counts are incremented by the local query counts at the entry point of the callee. Since the local query counts do not include queries issued after the function returns to the caller (this would require context-sensitive local query counts), we perform the last step of the computation dynamically during symbolic execution. We obtain the global query counts by adding the local query counts at the location of the current state to the sum of the local query counts of all return locations in the call stack.

Parameters. In our implementation, QCE is parametrized by α, β, and the loop bound κ. Optimal values for these parameters are difficult to compute analytically. For a given program, one can empirically find good parameter values using a simple hill-climbing method. In our experiments, we determined the parameter values this way using four programs and then used these values for the other programs, with good results (see §5.1).

Illustrating Example. Consider again the program in Figure 1 with the same input constraints we described in §3.1. We now illustrate how QCE can be used to decide whether to merge states at lines 6 and 7. We use the heuristic parameters $\alpha = 0.5$, $\beta = 0.6$ and, to keep the example brief, we set $\kappa = 1$. First, we pre-compute $Q_t(7)$ and $Q_{add}(7, v)$ for $v \in \{\texttt{r}, \texttt{arg}\}$ using Equation (4). For brevity, we omit the computation of Q_{add} for `argc`, `argv`, and array contents referenced by `argv`. For $Q_{add}(7, \texttt{arg})$, we get

$$\begin{aligned} &Q_{add}(7, \texttt{arg}) = q(7, c) \\ &= \beta q(8, c) + \beta q(10, c) + c(7, \texttt{arg} < \texttt{argc}) = \beta q(8, c) + 1 \\ &= \beta(\beta q(9, c) + \beta q(10, c) + c(8, \texttt{argv}[\texttt{arg}][\texttt{i}] \neq 0)) + 1 \\ &= \beta(\beta q(9, c) + 1) + 1 = \beta(\beta q(10, c) + 1) + 1 = \beta + 1 = 1.6, \end{aligned}$$

where $c = \lambda(\ell', e).\,\text{ite}((7, \texttt{arg}) \triangleleft (\ell', e), 1, 0)$. Similarly, we compute that $Q_{add}(7, \texttt{r}) = \beta + 2\beta^2 = 1.32$ and $Q_t(7) = 1 + 2\beta + 2\beta^2 = 2.92$ and, according to Equation (2), $H(7) = \{\texttt{arg}\}$. As there are no branches between lines 6 and 7, we have $H(6) = H(7) = \{\texttt{arg}\}$. Hence, in this example, the QCE similarity relation (1) allows the states at line 6 or 7 to be merged if the values of `arg` in the two states are either equal or symbolic. This is consistent with the results of our manual analysis in §3.1.

3.3 Justification

We now link our design of QCE to a cost model through the successive application of five key simplifying assumptions. Since QCE is merely a heuristic and not a precise computation, the following only provides a justification for the reasoning behind it, but not a formal derivation. Here we explain a full variant of QCE that includes an estimate of the cost for introducing ite expressions, even though it is not currently implemented in our prototype.

As mentioned above, an optimal heuristic for the similarity relation would compute whether the cumulative solving time T_m for all descendants of the merged state is guaranteed to be less than the combined respective times T_1 and T_2 for the two individual states, i.e., whether $T_m < T_1 + T_2$. In the ideal case of merging two identical states, we would have $T_m = T_1 = T_2$. Thus, merging just two states could theoretically cut the remaining exploration time in half. This is why, in principle, repeated merging can reduce the cumulative solving time by an exponential factor.

Precisely predicting the time required for solving a formula without actually solving it is generally impossible, therefore we apply a first simplification:

Simplifying Assumption 1. *A query takes one time unit to solve. Introducing new* ite *expressions into the query increases the cost to $\zeta > 1$ time units, where ζ is a parameter of the heuristic.*

Thus, we assume the estimated solving time to be linear in the number of queries of each type. In a further simplification, we treat the number of queries that each one of two merge candidates would *individually* cause in the future as equal:

Simplifying Assumption 2. *Two states at the same program location that are candidates for merging will cause the same number Q_t of queries if they are explored separately.*

This simplification is a prerequisite for statically computing query counts for a location in a way that is independent of the actual states during symbolic execution. The merged state then will also invoke these Q_t queries, but some queries will take longer to solve due to introduced ite expressions, and some additional queries become necessary. We denote the number of queries into which merging introduces ite expressions by Q_{ite} (with $Q_{ite} \leq Q_t$). The total cumulative cost of solving these queries is $\zeta \cdot Q_{ite}$, as per our first simplification. Additionally, the merged state can require extra solver invocations for queries corresponding to branch conditions that depend on constant but different values in the individual states (as in the loop conditions on lines 9 and 10 of Figure 1). This number of additional queries is Q_{add}.

Note that we ignore the possible cost of introducing disjunctions into the path condition. In many common cases, the different conjuncts of the two path conditions are just negations of each other, and thus the disjunctive path condition can be simplified to the common prefix of the two individual path conditions.

With these simplifications, the total cost of solver queries in the merged state is $1 \cdot (Q_t - Q_{ite})$ for the remaining regular queries plus $\zeta \cdot Q_{ite}$ for queries involving new ite expressions, plus $1 \cdot Q_{add}$ for the additional queries. We can thus formulate the criterion for performing a single merge as $Q_t - Q_{ite} + \zeta \cdot Q_{ite} + Q_{add} < 2 \cdot Q_t$, which simplifies to

$$(\zeta - 1) Q_{ite} + Q_{add} < Q_t. \tag{5}$$

The values for Q_t, Q_{ite}, and Q_{add} must be computed over the set of all feasible executions of the merged state. To statically estimate the feasibility of future paths, we add the following simplification:

Simplifying Assumption 3. *Each branch of a conditional statement is feasible with probability $0.5 < \beta < 1$, independently of the other branch.*

We can now estimate the query counts recursively. In the following definition, which is restated from (3), function $c(\ell', e)$ can be instantiated for Q_t, Q_{ite}, and Q_{add} individually to return 1 if checking the feasibility of a branch condition e at location ℓ' causes a query of the specific type (regular, involving ite expressions, or additional), or 0 otherwise:[1]

$$q(\ell', c) = \tag{6}$$
$$\begin{cases} \beta \cdot q(succ(\ell'), c) + \beta \cdot q(\ell'', c) + c(\ell', e) & instr(\ell') = \mathsf{if}(e)\,\mathsf{goto}\ \ell'' \\ 0 & instr(\ell') = \mathsf{halt} \\ q(succ(\ell'), c) & \text{otherwise} \end{cases}$$

For this definition, loop unrolling ensures that conditional statements in loops are counted as many times as the loop can execute. Loops and recursive calls with bounds that are not statically known are unrolled up to a fixed depth, given by the heuristic parameter κ.

The symbolic execution engine issues a query whenever a state (ℓ, pc, s) encounters a branch with a conditional expression e that depends on program input, i.e., e evaluates to an expression $s[e]$ containing variables from the set of inputs I. We denote this by $I \blacktriangleleft s[e]$. To ease notation, we add the following shorthands: we use $s_1[v] \neq_s s_2[v] \overset{def}{\Leftrightarrow} (I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge s_1[v] \neq s_2[v]$ for the condition causing ite expressions, i.e., symbolic but non-equal variables in two states, and we use $s_1[v] \neq_c s_2[v] \overset{def}{\Leftrightarrow} \neg(I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge s_1[v] \neq s_2[v]$ for the condition causing additional queries, i.e., concrete and non-equal variables in two states.

To define a function $c(\ell', e)$ for the different types of query counts, we need a method to check whether the branch condition e depends on inputs when reached from one of the individual states. We approximate this statically using a path-insensitive data dependence analysis, and write $(\ell, v) \triangleleft (\ell', e)$ if expression e at location ℓ' may depend on the value of variable v at location ℓ. Thus, we can define the query counts as follows:

$$Q_t((\ell, pc_1, s_1), (\ell, pc_2, s_2)) = q\big(\ell, \lambda(\ell', e).\ \text{ite}(\exists v : (I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge (\ell, v) \triangleleft (\ell', e)), 1, 0\big)$$
$$Q_{ite}((\ell, pc_1, s_1), (\ell, pc_2, s_2)) = q\big(\ell, \lambda(\ell', e).\ \text{ite}(\exists v : s_1[v] \neq_s s_2[v] \wedge (\ell, v) \triangleleft (\ell', e)), 1, 0\big)$$
$$Q_{add}((\ell, pc_1, s_1), (\ell, pc_2, s_2)) = q\big(\ell, \lambda(\ell', e).\ \text{ite}(\exists v : s_1[v] \neq_c s_2[v] \wedge (\ell, v) \triangleleft (\ell', e), 1, 0\big)$$

Computing this recursive relation is expensive, and it cannot be pre-computed before symbolic execution because it requires determining which variables depend on program inputs in the states considered for merging. We therefore assume a fixed probability of input dependence:

Simplifying Assumption 4. *The number of branches whose conditions are dependent on inputs is a fixed fraction φ of the total number of conditional branches.*

[1] Note that, for simplicity of exposition, we only refer to branch conditions here. In practice, other instructions, such as assertion checks or memory accesses with input-dependent offsets, will also trigger solver queries. Our implementation extends the definition of c to account for these queries.

This enables us to eliminate all variable dependencies from Q_t and simplify it to $Q_t(\ell) = \varphi \cdot q(\ell, \lambda(\ell', e).1)$. Now, Q_t depends only on the program location and can thus be statically pre-computed.

Q_{ite} and Q_{add} count queries for which specific variable pairs are not equal in the two merge candidates. Therefore, we would need to statically pre-compute Q_{ite} and Q_{add} for each subset of variables that could be symbolic in either state during symbolic execution. To eliminate this dependency on the combination of specific variables, we compute query counts for individual variables. The *per-variable query counts* $Q_{ite}(\ell, v)$ and $Q_{add}(\ell, v)$ are defined as the value of $Q_{ite}(\ell)$ and $Q_{add}(\ell)$, respectively, computed as if v was the only variable that differs between the merge candidates. The per-variable query counts can be computed as $Q_{ite}(\ell, v) = Q_{add}(\ell, v) = q(\ell, \lambda(\ell', e).\text{ite}((\ell, v) \triangleleft (\ell', e), 1, 0))$.

Summing the per-variable query counts for all variables that differ between the merge candidates will grossly over-estimate the actual values of Q_{ite} and Q_{add}, since conditional expressions often depend on more than just one variable, and many queries would thus be counted multiple times. Similarly, using just the maximum per-variable query count would cause an under-estimation. In fact,

$$\max_{\{v \in V | s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) \leq Q_{add}(\ell) \leq \sum_{\{v \in V | s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v)$$

and analogously for Q_{ite}. We therefore make a final simplification:

Simplifying Assumption 5. *Total query counts are equal to the maximum per-variable query counts for an individual variable times some factor σ, i.e.,*

$$Q_{ite}(\ell) \approx \sigma \cdot \max_{\{v \in V | s_1[v] \neq_s s_2[v]\}} Q_{ite}(\ell, v)$$
$$Q_{add}(\ell) \approx \sigma \cdot \max_{\{v \in V | s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v).$$

The intuition behind this assumption is that the number of independent variables correlates with the input size and not with the total number of variables. Applying this substitution to Equation (5) we can now define the similarity relation $\sim_{qce}$ as

$$(\ell, pc_1, s_1) \sim_{qce} (\ell, pc_2, s_2) \overset{def}{\iff} \tag{7}$$
$$(\zeta - 1) \max_{\{v \in V | s_1[v] \neq_s s_2[v]\}} Q_{ite}(\ell, v) + \max_{\{v \in V | s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) < \frac{Q_t}{\sigma}$$

with

$$Q_{ite}(\ell, v) = Q_{add}(\ell, v) = q\big(\ell, \lambda(\ell', e).\text{ite}\left((\ell, v) \triangleleft (\ell', e), 1, 0\right)\big),$$
$$Q_t(\ell) = \varphi \cdot q\big(\ell, \lambda(\ell', e).1\big),$$

and the recursively descending q as defined in Equation (6). For convenience, we rename $\frac{\varphi}{\sigma}$ to the unified parameter α. Thus, α, β, ζ and the unrolling bound κ remain as the only parameters to QCE. The variant of QCE implemented in our prototype is derived from Equation (7) by removing Q_{ite} from the criterion, to arrive at

$$\max_{\{v \in V | s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) < \alpha Q_t,$$

which is equivalent to

$$\forall v \in V : s_1[v] \neq_c s_2[v] \rightarrow Q_{add}(\ell, v) < \alpha Q_t.$$

To facilitate an efficient implementation in combination with dynamic state merging, as discussed in the next section, we collect a set of variables that exceed the threshold $H_{add}(\ell) = \{v \in V \mid Q_{add}(v) > \alpha Q_t\}$ and can state the similarity relation as (1).

This motivates the use of QCE for estimating the similarity of states. We show that QCE is effective in practice in §5.

4. Dynamic State Merging

We now explain the challenges for applying state merging in fully automated, precise, but incomplete symbolic program analysis (§4.1). To overcome these problems, we motivate (§4.2) and introduce (§4.3) the dynamic state merging algorithm.

```
1 if (logPacketHash) {
2   hash = computeHash(pkt);
3   log("Packet: %s, hash: %s", pkt->name, hash);
4 } else {
5   log("Packet: %s", pkt->name);
6 }
7 handlePacket(pkt);
```

Figure 2. Example code illustrating how static state merging can interfere with search heuristics.

4.1 Static Merging and Incomplete Exploration

A symbolic program analysis using static state merging traverses the CFG in topological order and attempts to merge states at every joint point. This allows to perform exhaustive exploration with state merging in the fewest possible steps. To ensure termination, the analysis has to stop loop unrolling at a certain depth, unless loops can be summarized by loop invariants. This method is optimal for verification condition generators that encode full programs with bounded or summarized loops. Search-based symbolic execution engines, however, which typically perform incomplete explorations, do not bound loops but are guided by search strategies that prioritize exploring new code over unrolling additional loop iterations. An exploration in strict topological order would override such strategies and stall the engine by requiring it to fully unroll the possibly infinitely many iterations of a loop before proceeding.

This is a problem even for loops that symbolic execution could, in principle, explore exhaustively. Coverage-oriented search strategies are designed to quickly maximize metrics such as statement coverage. The restriction to topological order interferes with such strategies, reducing their performance or even completely stopping them from achieving any progress towards their goal. We support this argument with experimental evidence in §5.5.

Consider the example code in Figure 2. Depending on the flag `logPacketHash`, the program writes to a log either just the name of a packet or both the name and a hash value. The code then processes the packet. In this example, exploring the "else" branch of the conditional statement on line 1 is fast, while exploring the "then" branch is expensive due to the `computeHash` function processing the entire input packet. A coverage-oriented search strategy will likely choose to explore the "else" branch first to quickly reach `handlePacket`, or switch to the "else" branch after not making progress towards its coverage goal being stuck unrolling loops inside the `computeHash` function. With static state merging, the symbolic execution engine has to merge all states at line 7, which requires all execution paths to be explored exhaustively up to that location. Therefore, no code in the `handlePacket` function can be reached before exploring every path in the `computeHash` function, thus being in conflict with the coverage-oriented search heuristic. This conflict could be solved by allowing the merge of only those states that, according to the chosen search strategy, would reach the same point in a program at about the same time. In our example, the state that takes the "else" branch should not be merged with any state that takes the "then" branch, allowing the search strategy to prioritize it independently.

4.2 Rationale Behind Dynamic State Merging

To solve the problems of static state merging, we propose *dynamic state merging* (DSM). DSM does not require states to share the same program location in order to be considered for merging. The rationale behind dynamic state merging is the following: consider two abstract states $a_1 = (\ell_1, pc_1, s_1)$ and $a_2 = (\ell_2, pc_2, s_2)$, with $\ell_1 \neq \ell_2$, that are both in the worklist. Assume that a_1', one of the transitive successors of a_1 (which have not been computed yet)

Input: Worklist w, choice functions $pickNext_D$ and $pickNext_F$, similarity relation $\sim$, trace function $pred$, threshold δ
Data: Forwarding set F.
Result: The next state to execute.
// Determine the forwarding set
$F := \{a \in w \mid \exists a' \in w : \exists a'' \in pred(a', \delta) : a \sim a''\}$;
if $F \neq \varnothing$ **then** // Choose a state from the forwarding set
  return $pickNext_F(F)$
else // Choose a state using the driving heuristic
  return $pickNext_D(w)$

Algorithm 2. The *pickNext* method for dynamic state merging.

will reach location ℓ_2. Provided that the number of steps required to reach ℓ_2 from a_1 is small, and the expected similarity of a_1' and a_2 is high, enough that merging them will be beneficial, it is worth overriding a coverage-oriented search strategy to compute a_1' next and to then merge it with a_2. We refer to this override as *fast-forwarding*, because a_1 is forwarded to a_2's location with temporary priority before resuming the regular search strategy.

To check whether a_1 can be expected to be similar to a_2 in the near future, we check whether a_1 could have been merged with a predecessor a_2' of a_2, i.e., whether $a_1 \sim a_2'$. The underlying expectation, which our experiments confirm, is that if two states are similar, then their two respective successors after a few steps of execution are also likely (but not guaranteed) to be similar.

Note that fast-forwarding deals with special cases automatically: if a state forks while being fast-forwarded, all children that are still similar to a recent predecessor of a state in the worklist are fast-forwarded. If a state leaves the path taken by the state it is similar to, i.e., fast-forwarding diverges, the state is no longer similar to any predecessor and is thus no longer prioritized.

4.3 The Dynamic State Merging Algorithm

The DSM algorithm is an instance of Algorithm 1 with a *pickNext* function as defined in Algorithm 2. DSM relies on an external "driving" heuristic (given as a function $pickNext_D$), such as a traditional coverage-oriented heuristic, to select the next state. However, when the algorithm detects that some states, computed as a set F, are likely to be mergeable after at most δ steps of execution, DSM overrides the driving heuristic and picks the next state to execute from F according to another external search heuristic $pickNext_F$.

The algorithm uses a function $pred(a, \delta)$ to compute the set of predecessors of a within a distance of δ. The function can be defined as $pred(a, \delta) = \{a' \mid \exists n \leq \delta : a \in post^n(a')\}$, where $post(a')$ denotes the set of immediate successor states computed by Algorithm 1 for a state a'. Keeping a precise history of reached states can incur prohibitive space costs, but we reduce the space requirements as follows: states from the history are only used for comparisons with respect to $\sim$, hence it is only required to store the parts of the state that are relevant to the relation. Moreover, if the $\sim$ relation is only sensitive to equality, the implementation can store and compare hash values of the relevant information from past states. In this case, checking whether the state belongs to F is implemented as a simple hash table lookup. Hash collisions do not pose a problem, because a full check of the similarity relation is still performed when fast-forwarding finishes and the states are about to be merged. Moreover, the set F is rebuilt after each execution step, so, if a state was added to F due to a collision, it is unlikely to be added again to F in the next step, as a second collision for two different hash values has low probability.

The relation $\sim_{qce}$, defined in Equation (1) in §3.3, can be modified to check for equality only, as required for using hashing in the implementation. We express the condition for variables to be either symbolic or equal in Equation (1) by $h(s_1[v]) = h(s_2[v])$, where $h(v) = \text{ite}(I \blacktriangleleft v, \star, v)$ filters out symbolic variables by mapping them to a unique special value. The implementation can thus store just the hash value of $\bigcup_{v \in H(\ell)} h(v)$ for a state. Then $\sim_{qce}$ can be checked by comparing the hash values of the two states (modulo hash collisions).

The function $pickNext_F$ determines the execution order among the states selected for fast-forwarding. In our implementation, we pick the first state from F according to the topological order of the CFG. Thus, states that lie behind with respect to the topological order first catch up and are merged with later states.

5. Experimental Evaluation

We now show that, in practice, our approach attains exponential speedup compared to base symbolic execution (Figure 5). We first introduce our prototype implementation (§5.1) and present our evaluation metrics (§5.2). We then evaluate how DSM combined with QCE improves the thoroughness of program exploration in a time-bounded scenario (§5.3). We then show that QCE lies at a sweet spot between single-path exploration and static merging (§5.4). Finally, we demonstrate that DSM is essential for combining state merging with coverage-oriented search strategies in incomplete exploration (§5.5).

5.1 Prototype

We built our prototype on top of the KLEE symbolic execution engine [6]. It takes as input a program in LLVM bitcode [25] and a specification of which program inputs should be marked as symbolic for the analysis: command-line arguments or file contents. KLEE implements precise non-compositional symbolic execution with feasibility checks performed at every conditional branch. It uses search strategies to guide exploration; the stock strategies include random search and a strategy biased toward covering previously unexplored program statements.

To support QCE, we extended KLEE to perform a static analysis of the LLVM bitcode to compute local query count estimates as explained in §3.2. The analysis is executed before the path exploration and annotates each program location with the corresponding query count estimates $Q_t(\ell)$ and $Q_{add}(\ell, v)$ as defined in §3.2. The pass is implemented as an LLVM per-function bottom-up call graph traversal (with bounded recursion) and performs the analysis compositionally. When analyzing each function, the pass attempts to statically determine trip counts (number of iterations) for loops. If it cannot, it approximates them with the loop bound parameter κ. The QCE analysis tracks the query count for local variables, function arguments, and in-memory variables indexed by a constant offset and pointed to by either a local variable, a function argument, or a global variable. We check data dependencies between variables by traversing the program in SSA form. As LLVM's SSA form handles only local variables, we do not track dependencies between in-memory variables except when loading them to locals. We modified KLEE to compute interprocedural query counts and sets of hot variables $H(\ell)$ dynamically during symbolic execution, following §3.2.

We implemented DSM as defined by Algorithm 2 in the form of a search strategy layer in KLEE's stacked strategy system plus an execution tracking system that incrementally computes state hashes (see §4.3). Each strategy uses its own logic to select a state from the worklist, but can rely on an underlying strategy whenever it has to make a choice among a set of equally important states. In our case, the DSM strategy returns a state from the fast-forwarding set of states ($pickNext_F$), or, if this set is empty, it resorts to the underlying driving heuristic to select a state from the general worklist ($pickNext_D$). Depending on the purpose of each experiment, we employ different driving heuristics: for complete

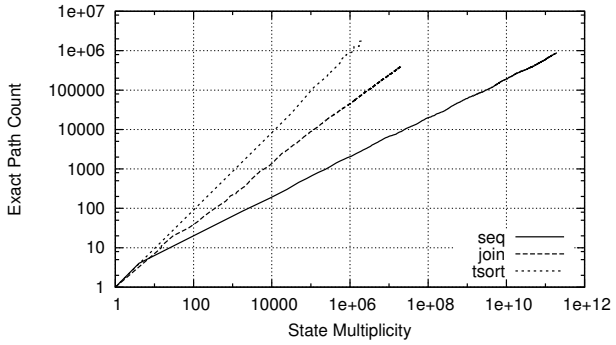

Figure 3. The exact number of paths as a function of state multiplicity for 3 COREUTILS tools. Both axes are logarithmic.

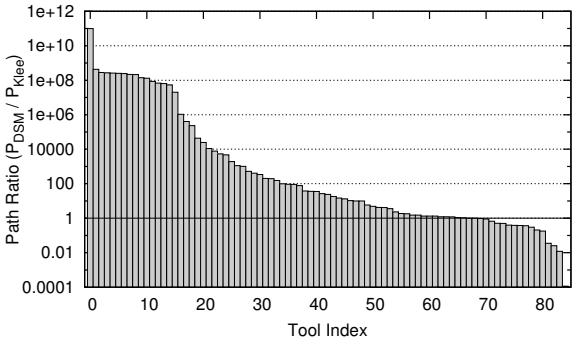

Figure 4. Relative increase in explored paths for DSM + QCE vs. regular KLEE (1h time budget). Each bar represents a COREUTIL.

explorations, we used random search, while for partial explorations aimed at obtaining statement coverage, we employed the coverage-oriented search heuristics in [6].

Evaluation Targets. We performed all our experiments on the COREUTILS suite of widely used UNIX command-line utilities, ranging from file manipulation (`cp`, `mv`, etc.) to text processing (`cut`, `sort`, etc.) and shell control flow (e.g., `test`). The total size of the COREUTILS code is 72.1 KLOC, as measured by SLOCCOUNT [30]. We tested these tools using symbolic command line arguments and `stdin` as input. In some cases, the symbolic input size is small enough for KLEE to complete the exploration in less than 5 minutes. We discarded these data points from our evaluation, since such a short period of time is dominated by the constant overhead of our static analysis, whereas we are interested in evaluating the prototype's asymptotic behavior.

5.2 Evaluation Metrics

We evaluate our prototype by comparing it to non-merging search-based symbolic execution as implemented in KLEE. We perform the comparison according to (1) the *amount of exploration* performed given a fixed time budget, and (2) the *time* necessary to complete a fixed exploration task, i.e., the exhaustive search of a set of paths determined by a given symbolic input.

Estimating Number of Paths in Merged States. A direct comparison of the amount of exploration between merge-based and regular symbolic execution is difficult, because counting the feasible paths that have been explored with state merging requires checking the feasibility of each path individually. This is as hard as repeating the exploration without merging, so it is impractical in the cases where symbolic execution without merging times out.

We therefore estimate the path count in a merged state with the help of *state multiplicity*: the multiplicity of a single-path state is 1; when two states merge, the multiplicity of the resulting state is the sum of their multiplicities. State multiplicity over-estimates the path count because, when the state is split at later branches, state multiplicity carries over to both child states, effectively doubling the counted number of paths at each branch (assuming that, as long as a branch is feasible for the merged state, it is also feasible for all the paths represented by it).

For our estimation, we made the assumption that "path explosion" can be modeled as an exponential function $a \cdot 2^{b \cdot n}$, where a and b are program-specific constants, and the growth parameter n indicates the progress of unrolling the program CFG along the paths of interest. Both the path count p and state multiplicity m are then based on this formula, each with different values for the constants a and b. Since n depends on the behavior of the search strategy, it cannot be easily determined. However, we can avoid computing n because, at any point in time, both the exact path count and the state multiplicity have the same growth parameter n corresponding to the current exploration progress. Hence, we can take the two model equations $m = a_m \cdot 2^{b_m \cdot n}$ and $p = a_p \cdot 2^{b_p \cdot n}$ and obtain n from the first equation and substitute it in the second. This yields a relation between p and m of the form $\log p \approx c_1 + c_2 \log m$, where c_1 and c_2 are program-dependent coefficients.

To empirically validate this relation, we extended our prototype to accurately track the number of feasible paths by maintaining all the original single-path states along with the merged states. We counted the exact number of paths and the state multiplicity for 1 hour and confirmed a linear relation between the logarithms of the two values. Figure 3 illustrates the dependency between m and p with representative measurements of c_2 for 3 COREUTILS.

In the experiments reported in the rest of this section, we approximated the number of paths for state merging as follows. First, we ran the experiments for 1 hour, while accurately tracking the number of feasible paths as explained above (if exploration with merging was 1000× faster, this 1 hour would correspond to ∼4 seconds of exploration). From this data, we computed the values of c_1 and c_2. Second, we ran the full experiments, while tracking only the state multiplicity for merged states, which does not incur a significant overhead. Using c_1 and c_2, we then converted the state multiplicity values into the estimation of the feasible path count.

5.3 Faster Path Exploration with DSM and QCE

We were first concerned with how much DSM and QCE, when combined, speed up symbolic execution. We let both our prototype and KLEE run for 1 hour on each of the COREUTILS, and we measured the number of paths explored by each tool. The size of the symbolic inputs passed to each utility was large enough to keep each tool busy for the duration of each run.

Figure 4 shows, for each of the tested COREUTILS, a bar representing the ratio between the number of program paths explored by our prototype and KLEE, respectively. The results indicate that our technique explores up to 11 orders of magnitude more paths than plain symbolic execution in the same amount of time. On 14 utilities, our prototype explored fewer paths, which we believe to be due to ignoring ite expressions (see §3.3) and limitations of our prototype (see §5.1). Our prototype crashed on 5 utilities, which we do not include in our results. We show a single representative for every tool aliased by multiple names (e.g., `ls`, `dir`, and `vdir`).

5.4 Achieving Exponential Speedup with QCE

We now answer the question of how much faster can our technique exhaustively explore a program for a fixed input size. In exhaustive exploration, a coverage-oriented search strategy is not necessary, therefore we focused on the effects of QCE alone. We used it in implementing a selective form of static state merging instead of DSM. Static state merging (SSM) chooses states from the worklist

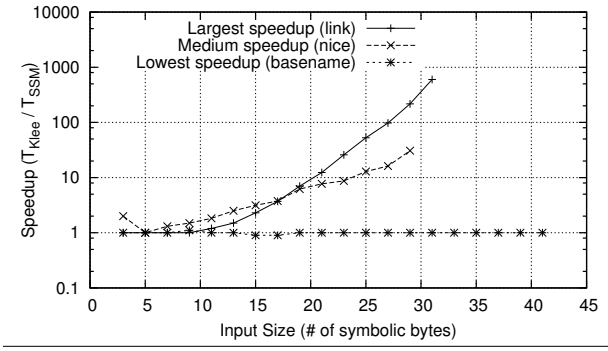

Figure 5. Speedup of QCE versus input size for exhaustive exploration of three representative COREUTILS.

in topological order and attempts to merge them at control flow join points. Selective SSM uses query count estimation to keep some states separate. Intuitively (and confirmed experimentally in §5.5), SSM performs better than DSM in exhaustive exploration, since it avoids unnecessary computation. However, for incomplete exploration, SSM performs worse than DSM.

We evaluate QCE from two perspectives. First, we look at how much QCE speeds up state merging for complete path exploration tasks. Second, we look at how the QCE heuristic parameters influence performance. We collected our measurements by running each of the COREUTILS for 2 hours, using plain KLEE, and QCE with SSM. For each configuration, we ran experiments with multiple values of symbolic input size, and we measured the corresponding completion time.

Exhaustive Exploration. Figure 5 shows the evolution of the completion time ratio (speedup) between SSM using QCE and plain KLEE for three representative COREUTILS programs, as we increase the symbolic input size. One of them achieved the highest speedup, another shows an average speedup, while the last does not show any improvement. This graph illustrates that our prototype completed the exploration goal exponentially faster as the symbolic input size increased. Figure 5 also shows that applying QCE does not always lead to speedup. However, Figure 6 shows this is actually an infrequent case. The scatter plot illustrates how the execution time of SSM using QCE compares to KLEE when aggregating over the entire set of experiments. The black dots correspond to experiment instances where both our prototype and KLEE finished on time, while the triangles on the right side correspond to situations where KLEE timed out after 2 hours (and thus indicate a lower bound on the actual speedup). The gray disks indicate the relative size of the symbolic inputs used in each experiment instance. Since, in the majority of cases, KLEE times out without completing the exploration, we only show on this graph the timeout points of the smallest input size for each tool, which give a loose lower bound on the speedup. We notice that most of the points in Figure 6 are located in the lower-right part of the graph, which correspond to higher speedup values. Moreover, in the cases where both KLEE and our prototype finish on time, large speedups (the lower-right part of the graph) tend to correspond to larger sizes of the symbolic input (larger gray circles). This re-confirms the fact that our speedup is proportional to the size of program input, i.e., the very source of the exponential growth of paths that bottlenecks KLEE.

To get a better understanding of the behavior of QCE, we took a deeper look at the execution of several COREUTILS tools: some that exhibit high speedup, and some for which the performance of our prototype is worse than KLEE's. We extended our prototype to explore states in both merged and unmerged forms. For every solver query in a merged state, we matched all the queries during the same execution step in all the corresponding unmerged states. We then

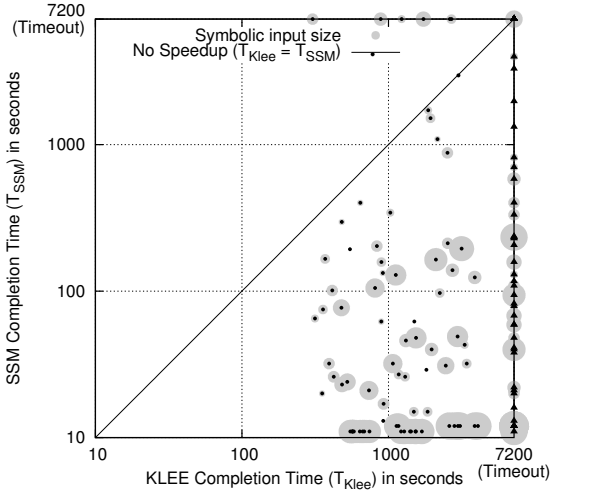

Figure 6. QCE + SSM vs. plain KLEE with varying input sizes (shown as gray disk size). Triangles denote that KLEE timed out after 2h and are thus *lower* bounds on the actual speedup.

compared query times in the merged and unmerged versions and identified those that became slower due to state merging.

We discovered that, even for tools that exhibited high overall speedup, some queries are more expensive in the merged state than the corresponding queries for the unmerged states combined. In these cases, however, the slowdown was amortized by the reduction in the total number of states to explore, and hence the total number of queries to solve. A typical example is the `sleep` utility, which reads a list of integers from the command line and sums their value in the variable `seconds`. It then validates the resulting value and performs the actual sleep operation. Here, QCE does not identify `seconds` as a hot variable, and all states forked during parsing are merged into a single state, avoiding the exponential increase in paths for each additional integer parsed. Since the value of `seconds` depends on the parsing result, it becomes a complex symbolic expression in the merged state, leading to several complex solver queries in the validation code. Nevertheless, these queries are amortized by the substantial reduction in the number of states to analyze. This example shows that QCE does allow merging of states that differ in live variables, so it is strictly more general than methods based on live variable analysis [3].

In cases where our prototype performed worse than plain KLEE, we observed a large number of queries that were more expensive in the merged state. These queries commonly contained ite expressions and disjunctive path conditions introduced by state merging. The former case shows that our QCE prototype can be improved by including the estimation of ite expressions introduced by state merging, as described in §3.3. The latter suggests using a constraint solver that can handle disjunctions more efficiently.

Influence of Heuristic Parameters. The values of the QCE parameters α, β, and κ affect the exploration time of each program. We determined optimal values for α and β experimentally, using hill-climbing over four COREUTILS chosen at random, and obtained $\alpha = 10^{-12}$ and $\beta = 0.8$. We then reused the same values for all other experiments in our evaluation and found that these values perform well in practice. Regarding the loop bound κ, we noticed that many of the loops with an input-dependent number of iterations actually iterate over program inputs. Hence, we chose $\kappa = 10$ corresponding to the average input size in our experiments.

We observed that, among these parameters, the value of α has the highest impact on the running time. In essence, α controls how aggressively the engine tries to merge. When α is ∞, no variables

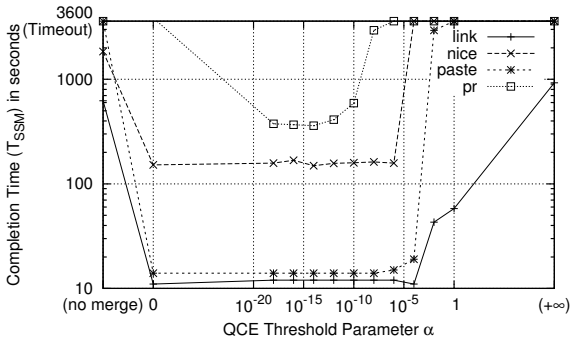

Figure 7. Impact on performance of the threshold parameter α.

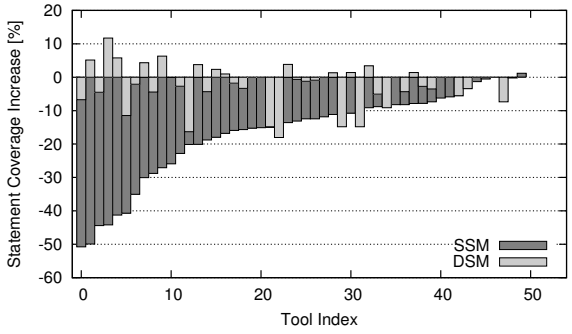

Figure 8. Change in statement coverage of DSM and SSM vs. regular KLEE for a coverage-oriented, incomplete exploration.

are determined to be hot, and QCE allows all states to be merged. When α is 0, states that contain variables with different concrete values are never merged. Due to this property, we call α the QCE *threshold parameter*. To illustrate this dependency, we randomly chose four COREUTILS (`link`, `nice`, `paste`, `pr`) and ran them using SSM for up to 1 hour with different values of α. Figure 7 shows, for each target program, the dependency of the completion time on the threshold parameter. The special point "(no merge)" on the x-axis corresponds to executions with state merging disabled. Note that here we did not cut execution times below 5 minutes.

5.5 Reaching an Exploration Goal with DSM

We now verify whether DSM allows the underlying driving heuristic to reach the goal while still merging states according to QCE. To isolate the effects of DSM, we compare DSM to our SSM implementation, with both using QCE in making merge decisions.

First, we use the coverage-oriented search heuristic from [6] with DSM, and look at how much statement coverage it can achieve in an incomplete setting (1 hour timeout and large search space). The value of the fast-forwarding distance δ was chosen experimentally to equal 8 basic blocks. Figure 8 compares the increase in statement coverage obtained by DSM and SSM over base KLEE on those COREUTILS for which the exploration remained incomplete after 1 hour. SSM consistently obtains worse coverage values, confirming its inability to adapt to the exploration goal. However, DSM roughly matches the coverage values of the underlying driving heuristic. Thus, the experiment confirms that DSM's merging avoids interfering with the logic of the driving heuristic, while traversing orders of magnitude more paths (§5.3). Even though these additional paths do not necessarily increase statement coverage, they can increase other coverage metrics and ultimately offer higher confidence in the resulting tests. We measured that, on average, 69% of the states selected for fast-forwarding were successfully merged with another state. Hence, the DSM approach to predict state similarity (§4.3) works well in practice.

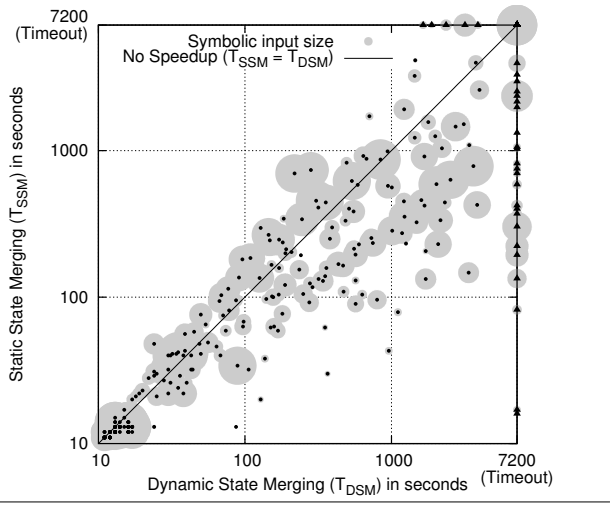

Figure 9. Comparison between the time needed to achieve exhaustive exploration for SSM and DSM.

Second, we evaluated the penalty of DSM compared to SSM in *exhaustive* exploration (see §5.4). For this experiment, we ran both techniques with varying input sizes. Figure 9 aggregates the results in a scatter plot. Most data points are grouped around the diagonal, indicating that the performance of both techniques is comparable, even though DSM is slower than SSM by 15% on average.

We conclude that DSM, while being slightly less efficient than SSM in exhaustive exploration, meets its purpose of allowing the driving heuristic to follow the exploration goal.

Overall, the combination of DSM and QCE offers exponential speedups over KLEE, which suggests that these are two important steps towards improving the performance of symbolic execution.

6. Related Work

We discussed the most closely related work in §2, where we focused on what we called precise symbolic program analysis. In this section, we look a bit further into alternative approaches that are similar in that they build symbolic expressions and rely on SAT or SMT solving, but use other techniques for improving scalability.

A first class of techniques focuses on pruning redundant states in symbolic execution. Boonstoppel et al. [3] dynamically determine variable liveness during symbolic execution. Their analysis considers the already explored paths through the current statement and determines the variables that are dead on all paths. It then uses the rest of the variables to check whether a state is equivalent to a previously explored one and can be safely pruned. In a sense, this is a special case of QCE, where no merging is performed unless the differing variables never used again. In our approach, we do not actually prune paths but still represent them in the merged state. If the differing variables are never used, this is equivalent to pruning one of the paths. This allows us to use imprecise static analysis and to merge in cases where variables are not dead but just rarely used.

McMillan [28] introduces lazy annotation in symbolic execution to build summaries on the fly and generalize them by Craig interpolation. This generalization goes beyond regular symbolic summaries, but is also computationally more expensive; we would like to measure the net effectiveness of this technique in future work.

A proven effective way to scale up symbolic program analysis is to forgo precision and introduce abstraction. Saturn [31] uses a symbolic exploration algorithm to build verification conditions for specific properties. It is specifically designed to find bugs in large system software and therefore sacrifices precision at several points. Loops are unrolled just once, and functions are aggressively sum-

marized. Similarly, Calysto [2] relies on structural abstraction to initially represent function effects as fresh variables. False positives are iteratively eliminated by replacing these variables with precise function summaries.

The bounded model checker in the Varvel/F-Soft verification platform uses lightweight static analysis to infer over-approximate function summaries that are only applied below a configurable depth in the call graph [21, 22]. Therefore, it introduces abstraction only at deeper levels, in an effort to reduce false positives. Sery et al. [29] describe the use of over-approximate summaries in bounded model checking. Whenever assertion violations are found, their method falls back to inlining, to avoid false positives. Therefore, speedups are only attainable for successful verification runs.

Abstraction-based analyses could scale significantly better than symbolic execution. However, they are prone to false positives and, perhaps more importantly, are harder to deploy. Symbolic execution engines do not require hand-written stubs for external functions or system calls, but can instead simply execute the call by concretizing its parameters. This sacrifices the theoretical guarantee of eventually achieving complete path coverage, but is a significant advantage for test case generation and bug finding.

7. Conclusions

In symbolic execution, state merging reduces the number of states that have to be explored, but increases the burden on the constraint solver. We introduced two techniques for reaping practical benefits from state merging: query count estimation and dynamic state merging. With this combination of techniques, state merging becomes completely dynamic and benefit-driven, unlike static strategies such as static merging or precise function summaries. We experimentally confirmed that our approach can significantly improve exploration time and coverage. This suggests that we have indeed come close to a sweet spot in balancing the simplicity of exploring single paths vs. the reduction of redundancy in exploring multiple paths in a merged state.

Other types of precise symbolic program analysis face similar design choices for grouping paths, but approach the sweet spot from a different angle. Therefore, we believe that our results generalize beyond just symbolic execution and that, for example, query count estimation can serve as a partitioning strategy for verification conditions in bounded model checking.

Acknowledgments

We would like to thank Péter Bokor, Aarti Gupta, Rupak Majumdar, Raimondas Sasnauskas, Jonas Wagner, and Cristian Zamfir for their valuable feedback on earlier drafts of this paper. We are grateful to Google and Microsoft for generously supporting our work.

References

[1] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[2] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *Intl. Conf. on Software Engineering (ICSE)*, 2008.

[3] P. Boonstoppel, C. Cadar, and D. R. Engler. RWset: Attacking path explosion in constraint-based test generation. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[4] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. In *Intl. Conf. on Reliable Software (ICRS)*, 1975.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Conf. on Computer and Communications Security (CCS)*, 2006.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation (SOSP)*, 2008.

[7] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[9] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *Intl. Conf. on Programming Language Design and Implem. (PLDI)*, 2002.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[11] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In *Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT)*, 2003.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Intl. Conf. on Programming Language Design and Implem. (PLDI)*, 2002.

[14] M. K. Ganai and A. Gupta. Tunneling and slicing: towards scalable BMC. In *Design Automation Conf. (DAC)*, 2008.

[15] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2007.

[16] P. Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages (POPL)*, 2007.

[17] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2011.

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Intl. Conf. on Programming Language Design and Implem. (PLDI)*, 2005.

[19] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, 2008.

[20] T. Hansen, P. Schachte, and H. Sondergaard. State joining and splitting for the symbolic execution of binaries. In *Intl. Conf. on Runtime Verification (RV)*, 2009.

[21] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2005.

[22] F. Ivancic, G. Balakrishnan, A. Gupta, S. Sankaranarayanan, N. Maeda, H. Tokuoka, T. Imoto, and Y. Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *Intl. Conf. on Automated Software Engineering (ASE)*, 2011.

[23] J. C. King. A new approach to program testing. In *Intl. Conf. on Reliable Software (ICRS)*, 1975.

[24] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Symp. on Principles of Programming Languages (POPL)*, 2008.

[25] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization (CGO)*, 2004.

[26] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[27] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symp. on Programming (ESOP)*, 2005.

[28] K. L. McMillan. Lazy annotation for program testing and verification. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2010.

[29] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *Haifa Verification Conf. (HVC)*, 2011.

[30] D. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[31] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Symp. on Principles of Programming Languages (POPL)*, 2005.