# Efficient State Merging in Symbolic Execution (Extended Abstract)

Volodymyr Kuznetsov[1] Johannes Kinder[1,2] Stefan Bucur[1] George Candea[1]
[1]École Polytechnique Fédérale de Lausanne (EPFL)
{vova.kuznetsov,stefan.bucur,george.candea}@epfl.ch
[2]Royal Holloway, University of London
johannes.kinder@rhul.ac.uk

Recent tools [CDE08, GLM08, CKC11] 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 and actually execute a target program, including any external calls; this broadens their applicability to many real-world programs. Second, like static analysis, these tools can simultaneously reason about multiple program behaviors. Third, symbolic execution is fully precise, so it generally does not have false positives.

While recent advances in SMT solving have made symbolic execution tools significantly faster, they still struggle to achieve scalability due to 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 comes at the cost of having to analyze an exponential number of states.

**State merging.** One way to reduce the number of states is to merge states that correspond to different paths. 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 $(\mathrm{true}, [x = \mathrm{ite}(X < 0, 0, 5)])$. Here, $\mathrm{ite}(c, p, q)$ denotes the if-then-else operator that evaluates to $p$ if $c$ is true, and to $q$ otherwise.

State merging effectively decreases the number of paths that have to be explored [God07, HSS09], but also increases the size of the symbolic expressions describing variables. Merging introduces disjunctions, which are notoriously difficult for SMT solvers. 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 $(\mathrm{ite}(X < 0, 0, 5))$ in the merged state. If $x$ were to appear 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 [HSS09].

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.

**Our solution.** In this work (published as [KKBC12]), 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. Our main contributions are the introduction of query count estimation and dynamic state merging. *Query count estimation* is 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. *Dynamic state merging* is 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.

Experiments on all 96 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, and can cover up to 11 orders of magnitude more paths. Our code and experimental data are publicly available at `http://cloud9.epfl.ch`.

## References


[CDE08] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. 8th USENIX Symp. Oper. Syst. Design and Implem. (OSDI 2008)*, pages 209–224. USENIX, 2008.

[CKC11] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proc. 16th. Int. Conf. Architectural Support for Prog. Lang. and Oper. Syst. (ASPLOS 2011)*, pages 265–278. ACM, 2011.

[GLM08] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proc. Network and Distributed Syst. Security Symp. (NDSS 2008)*. The Internet Society, 2008.

[God07] P. Godefroid. Compositional Dynamic Test Generation. In *34th ACM SIGPLAN-SIGACT Symp. Principles of Prog. Lang. (POPL 2007)*, pages 47–54. ACM, 2007.

[HSS09] T. Hansen, P. Schachte, and H. Søndergaard. State Joining and Splitting for the Symbolic Execution of Binaries. In *9th Int. Workshop Runtime Verification (RV 2009)*, volume 5779 of *LNCS*, pages 76–92. Springer, 2009.

[KKBC12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proc. ACM SIGPLAN Conf. Prog. Lang. Design and Implem. (PLDI 2012)*, pages 193–204. ACM, 2012.