CPA-SymExec: Efficient Symbolic Execution in CPAchecker

Dirk Beyer
LMU Munich
Germany

Thomas Lemberger
LMU Munich
Germany

ABSTRACT

We present CPA-SymExec, a tool for symbolic execution that is implemented in the open-source, configurable verification framework CPAchecker. Our implementation automatically detects which symbolic facts to track, in order to obtain a small set of constraints that are necessary to decide reachability of a program area of interest. CPA-SymExec is based on abstraction and counterexample-guided abstraction refinement (CEGAR), and uses a constraint-interpolation approach to detect symbolic facts. We show that our implementation can better mitigate the path-explosion problem than symbolic execution without abstraction, by comparing the performance to the state-of-the-art Klee-based symbolic-execution engine Symbiotic and to Klee itself. For the experiments we use two kinds of analysis tasks: one for finding an executable path to a specific location of interest (e.g., if a test vector is desired to show that a certain behavior occurs), and one for confirming that no executable path to a specific location exists (e.g., if it is desired to show that a certain behavior never occurs). CPA-SymExec is released under the Apache 2 license and available (inclusive source code) at https://cpachecker.sosy-lab.org. A demonstration video is available at https://youtu.be/qoBHtvPKtnw.

CCS CONCEPTS

• **General and reference → Verification**; • **Software and its engineering → Formal methods**; **Formal software verification**;

KEYWORDS

Symbolic Execution, Program Analysis, Test-Case Generation

ACM Reference Format:
Dirk Beyer and Thomas Lemberger. 2018. CPA-SymExec: Efficient Symbolic Execution in CPAchecker. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240478

1 INTRODUCTION

Symbolic execution [18] has important applications in the area of software verification and testing. Many techniques rely on generating (symbolic) paths through the execution tree of a program, for example, test-case generation [9], fault localization [13], program repair [19, 20], and equivalence checking [17]. Even for software verification, approaches using symbolic execution are able to show remarkable results at the competition on software verification (SV-COMP) [1] [1]. Symbolic execution is a static program analysis that stores explicit values for variables whenever possible, and if an explicit value is not available, a new symbolic value is assigned to the variable and used as its value in further computations. This way, the symbolic value is propagated to other program variables depending on the undeterminable variable. Whenever a symbolic value occurs in an assumption, this assumption describes a constraint on the symbolic value for the remaining program path. Symbolic execution tracks these constraints alongside variable assignments to reason about the feasibility of a program path and to determine possible concrete values for symbolic variable assignments. Additionally, using symbolic values, relationships between program variables can be observed in a clear manner. Because of its high precision, symbolic execution suffers in its original form from path explosion.

Our symbolic-execution engine, CPA-SymExec, is able to mitigate path explosion through *abstraction* with CEGAR [11]. CEGAR computes the *precision* that is necessary for an analysis: it starts with an initial, coarse precision and then iteratively refines the precision based on infeasible target paths that are found during the analysis. The precision of CPA-SymExec, in particular, specifies for which variables the analysis tracks (symbolic) values and which constraints are important. CPA-SymExec computes a special instantiation of Craig interpolants [12] from infeasible target paths that allows us to derive program variables and constraints that must be tracked [7]. Through this, we often find an abstract model for the program that is detailed enough to reason about the executability of program paths, but still coarse enough to avoid path explosion.

CPA-SymExec is implemented in the software-verification framework CPAchecker, which offers many abstract domains and verification algorithms, is developed and maintained by over 20 active, international contributors from different institutions, and is released under the open-source license Apache 2. Modular design is a main objective of the project. The framework provides many utilities that are useful for performing software-analysis tasks: After a run, it provides detailed statistics about its different modules, and target paths are made available in JSON and as XML witnesses [4, 5]. For presentation purposes, visualizations are available for: the control-flow automaton (CFA), the abstract reachability graph (ARG), and the target paths [3]. For a found target, CPA-SymExec also returns the symbolic execution path to that target in a text file for further parsing in other tools, as well as an exemplary concrete execution path. In addition to this, CPA-SymExec can also be used for test-case generation based on condition coverage. In addition to CEGAR, CPA-SymExec uses several optimizations to speed up state exploration. For example, after every satisfiability check (SAT check) of path constraints, it stores a model for the constraints and checks

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240478

[1] https://sv-comp.sosy-lab.org/2018/results/results-verified

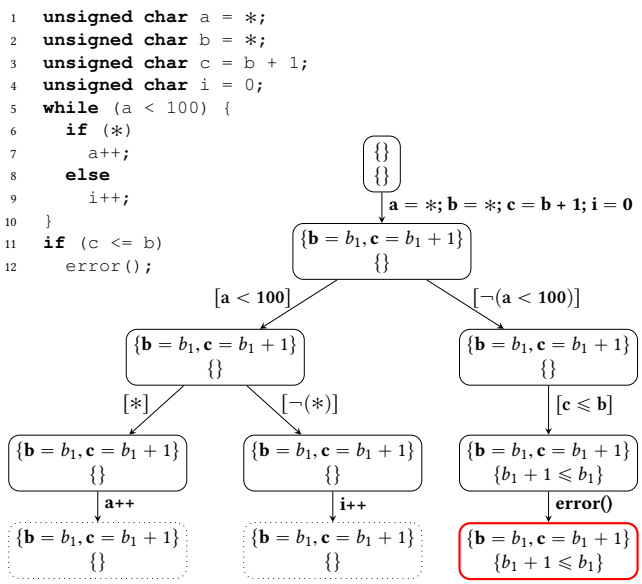
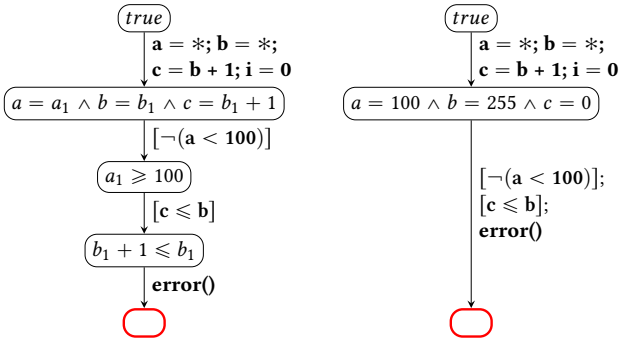


Figure 1: Example program with symbolic execution tree of CPA-SymExec to illustrate the effect of abstraction

whether it still holds for successive path constraints (to avoid expensive SAT checks), and it also uses that model to check whether any symbolic value is constrained to a single concrete value. If so, it permanently stores the corresponding assignment and replaces the symbolic value with the concrete value. Through this, the number of symbolic values in SAT checks can be minimized. As a further advancement, CPA-SymExec uses a random-weighted traversal strategy for state-space exploration that is inspired by Klee [9].

To show the maturity of our implementation, we performed experiments on thousands of C programs from the largest available repository of C verification tasks [2]. The results show that CPA-SymExec can compete with both state-of-the-art tools for symbolic execution of C programs, Klee and Symbiotic.

Example. Figure 1 shows the effect of using symbolic execution with CEGAR. The special symbol $*$ in the C program (top left in Fig. 1) represents a non-deterministic value. The ARG of the program (bottom Fig. 1) illustrates the explored state space. Every rectangle represents an abstract state, with the first line containing the variable assignments and the second line containing the tracked constraints. Assume we want to check whether the call to function **error** can be reached. Instead of tracking all assumptions and assignments that occur in the program and analyzing an exponentially growing (and even infinite) number of paths due to the non-deterministic value of variable **a** and the if-condition in the loop, symbolic execution with CEGAR only tracks the values of those variables that are necessary to prove the property; namely **b** and **c**. Using this level of abstraction, all possible program states of the loop are represented by the abstract states that are computed during the first iteration of the loop. The dotted rectangles represent abstract states at which the analysis stops because they are already covered by the computed state space. The red, bold rectangle in



(a) Symbolic path **(b) Concrete path**

Figure 2: Symbolic and concrete paths to the target

the lower right is an abstract state that represents a found target (due to overflow, cf. Fig. 2b).

If CPA-SymExec finds a target, it creates the symbolic path (Fig. 2a) and a concrete path (Fig. 2b) that lead to that target. These paths can be used for further analysis tasks, e.g., validation of verification results or program repair. In addition, CPA-SymExec can create a compilable test to check reachability through execution.

Contributions. The tool CPA-SymExec contributes the following:

- Support for the combination of abstraction refinement with symbolic execution by extending the previous implementation [7] to a mature symbolic-execution engine.
- Availability of all configuration options existing in the CPAchecker framework, including combinations with other abstract domains and analysis algorithms (e.g., pointer analysis or memory graphs).
- Optimizations (e.g., subset-superset caching) that are considered state-of-the-art and offered in Klee [9].
- Violation witnesses and executable tests [6].
- Automatic test-case generation [2] (also implemented in Klee) for condition coverage.
- Symbolic target paths to support comprehension of program paths and for further use.
- HTML-based reports to view and analyze control-flow automata, abstract reachability graphs, and target paths independent from specific tools (requires a standard web browser).

We show that our implementation is competitive against the state-of-the-art (Klee, Symbiotic) in a thorough experimental evaluation.

Supplementary Artifact and Webpage. All experimental data, additional evaluation of mentioned features, and further examples are available as supplement [8] [3].

Related Work. Klee [9] is a symbolic-execution tool that uses search heuristics to mitigate the path-explosion problem. It is able to symbolically and dynamically handle system calls in a program, while our implementation always handles them symbolically. Handling a system call dynamically executes it and uses the concrete value returned by the operating system for further analysis. This allows faster execution, but may not cover all possible program

[2] https://github.com/sosy-lab/sv-benchmarks

[3] https://sosy-lab.org/research/cpa-symexec-tool

states. Handling a system call symbolically uses a symbolic value that represents all possible values the operating system could return for the system call. The advantages and disadvantages are the opposite of dynamic handling. Klee can also provide a test case for every explored program path during the analysis of a program.

Symbiotic [10] is a symbolic-execution-based verifier that uses program-slicing with extended pointer analysis to mitigate path explosion. Symbiotic has been participating in SV-COMP successfully for several years. It uses Klee as a symbolic-execution back-end.

Tracer [14] computes (a) loop invariants using counterexample-guided loop unrolling to tackle the problem of unbounded loops and (b) weakest-precondition interpolants derived from infeasible target paths to weaken the precision of its symbolic execution. This is the opposite to our use of CEGAR. While we start with a low precision and refine it based on strongest-postcondition interpolants (precision increase), Tracer starts with a high precision and abstracts it based on weakest-precondition interpolants (precision decrease).

To make bug reports more readable, *path projection* [16] (see also [15]) takes an XML bug report and presents the corresponding program code after performing method inlining and code folding on it. This way, it creates one consecutive, readable stream of the executed program code that only shows information that is necessary to understand the bug without the need to jump between lines and files. The HTML interface of the target-path report of CPAchecker is similar to this, but displays a concrete target path.

2 ARCHITECTURE OF CPA-SYMEXEC

A run of CPA-SymExec consists of three main phases: The front-end module (Parser and CFA Builder) first creates an internal representation of the program under analysis as CFA, then the symbolic execution with CEGAR runs on that CFA and tries to find a target path for a provided specification. When the analysis terminates, statistics and other information are written for the user to better understand the result. Figure 3 shows this main architecture of CPA-SymExec.

Front-end. CPA-SymExec is able to build CFAs for programs written in C, Java, and LLVM bitcode. First, the front-end parses the program under analysis. From the resulting parse tree, it constructs the initial CFA and performs several optimizations to simplify the structure of the CFA. Then, the front-end derives additional information about the program based on syntactical information. This includes a live-variables analysis to determine which variables are active in the current scope, a classification of variables based on their usage, and an analysis of the loop structure of the program. The created CFA and the additional information collected is then passed to the symbolic execution.

Symbolic Execution. CPA-SymExec runs symbolic execution with an adjustable precision on the CFA. Initially, this precision is empty, i.e., symbolic execution does not track any variable assignments or path constraints. Whenever the symbolic execution finds a path that violates the given specification, that target path is symbolically executed with full precision. If this shows that the path is actually infeasible, the precision is refined by the precision-refinement procedure. The precision refinement extracts (by interpolation) from an infeasible target path which variables and constraints should be tracked. This information is used to increase the adjustable precision, such that the infeasible target path is not re-encountered by any successive symbolic execution, and the symbolic execution restarts with the new precision. If the symbolic execution finishes without finding a target path, CPA-SymExec terminates and the specification holds. If symbolic execution with full precision shows that a found target path is actually feasible, CPA-SymExec terminates and returns the target path with additional data to the user.

Analysis Output. After analysis, an HTML report [4] can be created to visualize the CFA, the explored state space and, if existing, target paths. The CFA is displayed in graph representation, the explored state space is displayed as an ARG, and target paths are displayed in two variants: (1) A concrete target path shows relevant concrete variable assignments at every step along the path, and (2) a symbolic target path shows the full symbolic state at every step along the path. If CPA-SymExec is used for test-case generation, a test harness is created for each reachable condition in the program as soon as it is reached. This test harness can be compiled together with the program under analysis to create an executable test.

Test-Case Generation. For test-case generation, an algorithm in CPAchecker is used that (a) defines every condition in the program as a target, and (b) only stops after, for each target, a path to it is found, or it is proven unreachable. For each target path, it then creates a compilable harness based on the work of execution-based result validation [6].

3 USING CPA-SYMEXEC

Installation. All steps for installing CPAchecker are explained on our supplementary webpage.[5] CPAchecker can be downloaded as binary release, or as source code using Subversion or Git. [6]

Execution. To run CPAchecker on the command line, execute **cpa.sh**, which can be found in directory **scripts/** of the installation, with the intended parameters followed by the file to analyze. To use CPA-SymExec, run CPAchecker with parameter **-symbolicExecution-Cegar**. If a C file is not yet pre-processed, CPAchecker can do that with option **-preprocess**.

Constraints of symbolic execution are resolved using an external SMT solver. Currently, CPAchecker supports SMTInterpol[7], MathSAT5[8], Z3[9], and Princess[10]. The solver can be selected by using parameter **-setprop solver.solver=SOLVER** with the intended solver; the default solver is MathSAT5. By default, CPA-SymExec uses bit-precise encoding of both bit-vectors and floating-point numbers.

A full line to execute an example program **test.c** looks like:

cpa.sh -symbolicExecution-Cegar -preprocess test.c

CPAchecker will create a set of files (in directory **output/**) for the user to better understand all findings (cf. Sect. 2). To perform test-case generation with CPA-SymExec, run it with configuration **-testCaseGeneration-symbolicExecution-Cegar**.

[4] cf. https://sosy-lab.org/research/cpa-symexec-tool for an example report
[5] https://sosy-lab.org/research/cpa-symexec-tool
[6] http://cpachecker.sosy-lab.org/download.php
[7] http://ultimate.informatik.uni-freiburg.de/smtinterpol/
[8] http://mathsat.fbk.eu/
[9] https://github.com/Z3Prover/z3
[10] http://www.philipp.ruemmer.org/princess.shtml

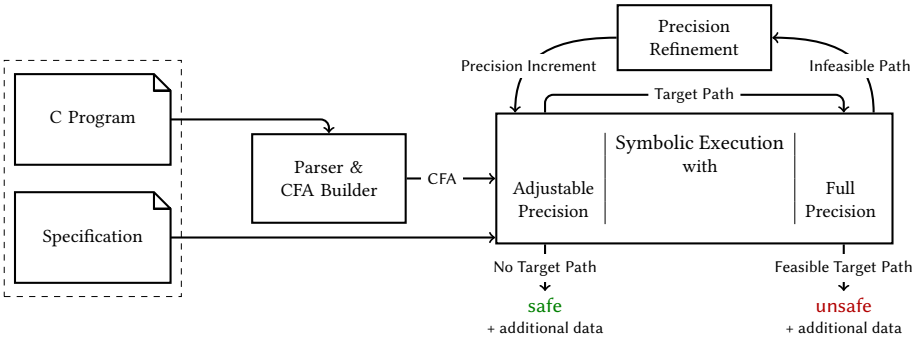


Figure 3: Architecture of CPA-SymExec

Table 1: Comparison of CPA-SymExec with the other state-of-the-art tools Klee and Symbiotic; best value highlighted

Tool	correct TRUE	correct FALSE	incorrect TRUE	incorrect FALSE	unsolved
CPA-SymExec	**2137**	545	**0**	22	**2865**
Klee	446	**899**	6	33	4 206
Symbiotic	1 201	848	3	**9**	3 529

4 COMPARISON

To illustrate the competitiveness of CPA-SymExec, we performed a thorough comparison with Klee and Symbiotic on 5 590 tasks of the largest available benchmark set for verification tasks in C [11]. Table 1 reports for each of the approaches the number of (1) correct answers that no path exists, (2) correct answers that a path is found, (3) missed paths (i.e., a path exists but the tool did not find it), (4) wrong paths (i.e., paths that cannot be executed in the program semantics), and (5) unsolved tasks. The different approaches of the three tools show in the results: Klee is the strongest tool for finding paths for a user query; Symbiotic is the most accurate tool: it reports the lowest number of wrong paths, i.e., paths that are actually not paths for which the user was querying; and CPA-SymExec is the best tool for exhaustively exploring the state space: the tool correctly reports 2 137 programs that do not contain a path for which the user was querying, and it did not miss any such path. This outperforms both Klee and Symbiotic.

5 CONCLUSION

We explained the architecture and basic concepts of CPA-SymExec, as well as the process of running CPA-SymExec for verification and test-case generation. A more detailed tutorial video on YouTube supports these explanations. CPA-SymExec in CPAchecker is a competitive symbolic-execution engine that allows users to combine symbolic execution with abstraction and CEGAR. In addition, the matureness of CPAchecker and the multitude of different abstract domains, algorithms, and utilities that are implemented in the framework allow an efficient implementation of new ideas and a quick adaption of existing techniques that rely on symbolic execution.

Acknowledgment. We thank the contributors of the CPAchecker project (http://cpachecker.sosy-lab.org/acknow.php); CPA-SymExec is based on many standard components of this framework.

[11] https://github.com/sosy-lab/sv-benchmarks

REFERENCES

[1] D. Beyer. 2017. Software Verification with Validation of Results (Report on SV-COMP 2017). In *Proc. TACAS (LNCS 10206)*. Springer, 331–349. https://doi.org/10.1007/978-3-662-54580-5_20
[2] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. 2004. Generating Tests from Counterexamples. In *Proc. ICSE*. IEEE, 326–335. https://doi.org/10.1109/ICSE.2004.1317455
[3] D. Beyer and M. Dangl. 2016. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses. In *Proc. CAV (2) (LNCS 9780)*. Springer, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28
[4] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351
[5] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. https://doi.org/10.1145/2786805.2786867
[6] D. Beyer, M. Dangl, T. Lemberger, and M. Tautschnig. 2018. Tests from Witnesses: Execution-Based Validation of Verification Results. In *Proc. TAP (LNCS 10889)*. Springer, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1
[7] D. Beyer and T. Lemberger. 2016. Symbolic Execution with CEGAR. In *Proc. ISoLA (LNCS 9952)*. Springer, 195–211. https://doi.org/10.1007/978-3-319-47166-2_14
[8] D. Beyer and T. Lemberger. 2018. Replication Package for Article "CPA-SymExec: Efficient Symbolic Execution in CPAchecker" in Proc. ASE'18. https://doi.org/10.5281/zenodo.1321181
[9] C. Cadar, D. Dunbar, and D. R. Engler. 2009. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. OSDI*. USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
[10] M. Chalupa, M. Vitovská, and J. Strejcek. 2018. SYMBIOTIC 5: Boosted Instrumentation - (Competition Contribution). In *Proc. TACAS (LNCS 10806)*. Springer, 442–446. https://doi.org/10.1007/978-3-319-89963-3_29
[11] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794. https://doi.org/10.1145/876638.876643
[12] W. Craig. 1957. Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem. *J. Symb. Log.* 22, 3 (1957), 250–268. https://doi.org/10.2307/2963593
[13] E. Ermis, M. Schäf, and T. Wies. 2012. Error Invariants. In *Proc. FM (LNCS 7436)*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17
[14] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Proc. CAV (LNCS 7358)*. Springer, 758–766. https://doi.org/10.1007/978-3-642-31424-7_61
[15] R. Jhala and R. Majumdar. 2005. Path Slicing. In *Proc. PLDI*. ACM, 38–47. https://doi.org/10.1145/1065010.1065016
[16] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal. 2008. Path projection for user-centered static analysis tools. In *Proc. PASTE*. ACM, 57–63. https://doi.org/10.1145/1512475.1512488
[17] D. Kim, Yonghwi Kwon, P. Liu, I. L. Kim, D. M. Perry, X. Zhang, and G. Rodriguez-Rivera. 2016. Apex: automatic programming assignment error explanation. In *Proc. OOPSLA*. ACM, 311–327. https://doi.org/10.1145/2983990.2984031
[18] J. C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394. https://doi.org/10.1145/360248.360252
[19] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: program repair via semantic analysis. In *Proc. ICSE*. IEEE, 772–781. https://doi.org/10.1109/ICSE.2013.6606623
[20] A. Roychoudhury. 2016. SemFix and beyond: semantic techniques for program repair. In *Proc. ForMABS*. ACM, 2. https://doi.org/10.1145/2975941.2990288