International Journal on Software Tools for Technology Transfer (2021) 23:867–870
https://doi.org/10.1007/s10009-020-00570-3

COMPETITIONS AND CHALLENGES

Special Issue: TestComp 2019

# KLEE symbolic execution engine in 2019

Cristian Cadar[1] · Martin Nowack[1]

Published online: 2 June 2020


**Abstract**
KLEE is a popular dynamic symbolic execution engine, initially designed at Stanford University and now primarily developed and maintained by the Software Reliability Group at Imperial College London. KLEE has a large community spanning both academia and industry, with over 60 contributors on GitHub, over 350 subscribers on its mailing list, and over 80 participants to a recent dedicated workshop. KLEE has been used and extended by groups from many universities and companies in a variety of different areas such as high-coverage test generation, automated debugging, exploit generation, wireless sensor networks, and online gaming, among many others.

**Keywords** Dynamic symbolic execution · Bug finding · Test generation

## 1 Short history and impact

KLEE is a popular testing and analysis platform, initially developed at Stanford University by Daniel Dunbar, Dawson Engler, and the first author of this paper [5], drawing inspiration from the design of EXE [7], another symbolic execution system developed at Stanford.

KLEE is based on dynamic symbolic execution [8], a variant of symbolic execution [2,9,15] which was initially introduced in 2005 by DART [13] and EGT [6]. Dynamic symbolic execution (DSE) provides the ability to automatically explore paths in a program, using a constraint solver to reason about path feasibility. It comes in two main flavours, concolic or offline DSE [13] and EGT or online DSE [6], with both variants based on mixed concrete-symbolic execution. KLEE implements the EGT flavour of DSE. We refer the reader to the original KLEE paper [5] for a detailed description of KLEE and DSE.

Since 2009, KLEE has been developed and maintained primarily by the Software Reliability Group (SRG) at Imperial College London, but with important contributions from outside developers. In the last ten years, KLEE has seen important improvements, both algorithmic and engineering in nature—to give just a couple of examples, the array acceleration technique proposed by Perry, Mattavelli, Zhang, and Cadar [21] illustrates the first category, while the work by Jiri Slaby and others of updating KLEE to work with recent LLVM versions (currently up to 8.0) illustrates the second category.

KLEE has a large user base, in both industry and academia. Examples of its popularity include a large number of citations to the original KLEE paper (over 2000 currently according to Google Scholar), the number of subscribers to KLEE's mailing list (over 350), and the number of stars on GitHub (over 1000). The original KLEE paper [5] was recently elected to the prestigious ACM SIGOPS Hall of Fame.[1]

KLEE is also a key component in many projects, such as Cloud9 [4], GKLEE [18], KLEENet [22], and Klover [17], to name just a few. The KLEE website currently lists over 140 papers that extend or use KLEE.[2]

The First International KLEE Workshop on Symbolic Execution took place in 2018 at Imperial College London.[3] It attracted over 80 participants from academia, industry, and government, with registration having to close early due to

Cristian Cadar: Jury member.

✉ Cristian Cadar
c.cadar@imperial.ac.uk

Martin Nowack
m.nowack@imperial.ac.uk

[1] Department of Computing, Imperial College London, London, UK

[1] https://www.sigops.org/awards/hof/.

[2] http://klee.github.io/publications/.

[3] https://srg.doc.ic.ac.uk/klee18/.

reaching capacity. The workshop was sponsored by the UK EPSRC, Baidu, Bloomberg, Fujitsu, Huawei, and Imperial College London.

Talks covered a wide range of topics related to KLEE and symbolic execution, such as scalability, usability, memory models, constraint solving, and new application domains. The schedule included both academic and industry speakers, with academic keynotes from Sarfraz Khurshid (UT Austin), Alessandro Orso (Georgia Tech), and Abhik Roychoudhury (NUS) and industry keynotes from Indradeep Ghosh (Fujitsu) and Peng Li (Baidu).

## 2 Software project and contributors

KLEE is an open-source tool, released under a liberal UIUC license and hosted on GitHub.[4] It is the work of many different contributors, over 60 of whom are listed on GitHub.[5] Since KLEE was moved to GitHub only in 2013, not all contributors are listed there. KLEE would not have been such a successful tool without its open-source contributors. Special thanks go to Daniel Dunbar, the main author of the original tool, and the maintainers of KLEE over the years, who, in addition to the authors of this paper, included in the past Daniel Dunbar, Dan Liew, and Andrea Mattavelli.

## 3 Software architecture

KLEE works at the level of LLVM bitcode, the intermediate language of the widely used LLVM compiler infrastructure [16]. It provides an interpreter that can execute almost arbitrary code represented in LLVM IR, both concretely and symbolically.

One of the main strengths of KLEE is its modular and extensible architecture. For example, while KLEE already provides a variety of different search heuristics to explore the program state space, it can be easily extended with new ones. A similar approach is taken for constraint solving, with constraint solving activities (such as optimisations and caching) structured as a series of stages. Existing stages can be readily enabled or disabled, and new ones easily added. Moreover, a variety of different SMT solvers are supported by different back ends, such as STP [12] (the default solver), Boolector [3], CVC4 [1], Yices 2 [11], and Z3 [10]. Some of these solvers are supported via the MetaSMT framework [14].

A customised C standard library, based on uClibc,[6] and a model for many POSIX library calls allow testing a variety of software systems that interact with their environment.

### 3.1 Modifications for Test-Comp 2019

The version of KLEE submitted to Test-Comp 2019 is based on commit `b845877a` in mainline KLEE (from January 2019). It uses the default options at that time, except for a few changes discussed below. While KLEE supports multiple solvers, we decided to configure it with STP, which in prior experiments had the best overall performance [20]. The use of other solvers could be explored in future editions of the competition.

We made several modifications to KLEE based on the nature of the Test-Comp benchmarks. For instance, we configured it differently for the bug-finding and the coverage categories, e.g. with the former configuration stopping KLEE as soon as an error is found, and the latter generating tests on the fly as soon as new basic blocks are covered.

As another example, we extended KLEE to better handle large numbers of symbolic variables, which is atypical for the benchmarks on which KLEE is usually run. We also extended KLEE to support the generation of XML-based test cases, a requirement for Test-Comp.

The Test-Comp benchmarks also helped reveal a series of bugs in KLEE, such as one affecting the handling of arrays of symbolic sizes and one concerning the debug information used in a coverage-based search heuristic. Besides that, we reduced the memory footprint of KLEE to handle more execution states simultaneously by sharing more information between states (e.g. sharing of common stack frames).

### 3.2 Set-up and configuration for Test-Comp 2019

The binary artefact is publicly available from the Test-Comp 2019 archives.[7] It runs on Ubuntu 18.04 and uses LLVM 6.0. Details related to the Test-Comp specific invocation can be found as part of the Python script `bin/klee` inside the artefact directory. This script compiles the given C file into LLVM bitcode and invokes KLEE (`klee_build/bin/klee`) on it. For detailed information on available options, see the documentation provided via the integrated help (`klee -help`) and at http://klee.github.io/.

We opted for all Test-Comp categories, even though support for symbolic floating-point arithmetic is not part of the artefact. We note that there are two extensions of KLEE for floating point [19], but they have not been integrated into the mainline yet.

### 3.3 Test-Comp 2019 results

KLEE placed second in both the bug-finding and coverage categories. In the coverage category, the tool was particu-

[4] https://github.com/klee/klee/.

[5] https://github.com/klee/klee/graphs/contributors.

[6] https://www.uclibc.org/.

[7] https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/master/2019/klee.zip.

larly close to the winning tool, VeriFuzz, gaining 1226 points vs 1238 points for VeriFuzz. In the bug-finding category, it gained 499 points vs 595 points.

One notable observation is that while KLEE found much fewer bugs in the bug-finding category than the winning tool (437 vs 592), it found the vast majority of the common bugs faster: out of the 433 bugs that both KLEE and the winning tool found, KLEE found 345 of the bugs quicker.

KLEE gained points in every sub-category, with the exception of the bug-finding sub-category involving floats, as the submitted version of KLEE does not have support for symbolic floating-point values (see Sect. 3.2).

More details on the results, including per-task results and score-based quantile plots, can be obtained at https://test-comp.sosy-lab.org/2019/results/results-verified/.

## 4 KLEE and the Test-Comp 2019 benchmarks

The main strength of KLEE, and dynamic symbolic execution in general, is that by modelling paths using mathematical formulas, it has applications to a wide variety of problems, beyond test generation and bug finding. Examples include debugging, specification inference, program and input repair, fault reproduction, and patch analysis.

In the context of the tasks relevant to Test-Comp, test generation and bug finding, KLEE benefits from a systematic exploration of the program search space and its ability to reason about all possible values on each path explored.

The weaknesses of KLEE and symbolic execution more generally are well documented and are mainly related to path explosion and constraint solving [8].

The benchmarks used in Test-Comp are inherited from SV-COMP, the software verification competition. As a result, they have a verification twist and are quite dissimilar to the benchmarks currently used to evaluate software testing tools. First, they are quite small: using the `cloc` tool, we measured a range of 10 to 184,969 executable lines of code (ELOC), but with a median of only 1409 ELOC. By contrast, the benchmarks currently used in software testing research are considerably larger.

Another particularity of the current Test-Comp benchmarks is that they involve huge numbers of symbolic bytes, as well as memory objects with arbitrary sizes. Furthermore, there are many hand-crafted examples to make analysis difficult. These present interesting challenges for testing tools, but they are quite atypical for the way testing tools are usually used.

We think that new types of benchmarks and challenges are needed to encourage more robust software testing tools. We believe it is important for future editions of Test-Comp to incorporate larger real-world applications like the ones used in current evaluations of testing tools. The difficulty is, of course, to incorporate such benchmarks in the current evaluation infrastructure, construct appropriate drivers for the competition environment, and deal with the complexity brought by real-world applications, such as complex build systems and interactions with the environment.

We are looking forward to continuing discussing these challenges with members of the community.

**Acknowledgements** We would like to thank once again the wonderful KLEE community; Dirk Beyer, the organiser of Test-Comp, for this excellent initiative that has the potential to bring important benefits to software testing research and practice; and Frank Busse and Tomasz Kuchta for proofreading our paper.



## References

1. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Proceedings of the 23rd International Conference on Computer-Aided Verification (CAV'11) (2011)
2. Boyer, R.S., Elspas, B., Levitt, K.N.: Select—a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the International Conference on Reliable Software (ICRS'75) (1975)
3. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09) (2009)
4. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: Proceedings of the 6th European Conference on Computer Systems (EuroSys'11) (2011)
5. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08) (2008)
6. Cadar, C., Engler, D.: Execution generated test cases: how to make systems code crash itself. In: Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN'05) (2005)
7. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06) (2006)
8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. Assoc. Comput. Mach. (CACM) **56**(2), 82–90 (2013)
9. Clarke, L.A.: A program testing system. In: Proceedings of the 1976 Annual Conference (ACM'76) (1976)

10. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08) (2008)
11. Dutertre, B.: Yices 2.2. In: Proceedings of the 26th International Conference on Computer-Aided Verification (CAV'14) (2014)
12. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Proceedings of the 19th International Conference on Computer-Aided Verification (CAV'07) (2007)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'05) (2005)
14. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R.: metaSMT: focus on your application not on solver integration. In: Proceedings of the International Workshop on Design and Implementation of Formal Tools and Systems (DIFTS'12) (2011)
15. King, J.C.: A new approach to program testing. In: Proceedings of the International Conference on Reliable Software (ICRS'75) (1975)
16. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04) (2004)
17. Li, G., Ghosh, I., Rajan, S.P.: KLOVER: a symbolic execution and automatic test generation tool for C++ programs. In: Proceedings of the 23rd International Conference on Computer-Aided Verification (CAV'11) (2011)
18. Li, G., Li, P., Sawaga, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'12) (2012)
19. Liew, D., Schemmel, D., Cadar, C., Donaldson, A., Zähl, R., Wehrle, K.: Floating-point symbolic execution: a case study in n-version programming. In: Proceedings of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17) (2017)
20. Palikareva, H., Cadar, C.: Multi-solver support in symbolic execution. In: Proceedings of the 25th International Conference on Computer-Aided Verification (CAV'13) (2013)
21. Perry, D.M., Mattavelli, A., Zhang, X., Cadar, C.: Accelerating array constraints in symbolic execution. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'17) (2017)
22. Sasnauskas, R., Landsiedel, O., Alizai, M.H., Weise, C., Kowalewski, S., Wehrle, K.: Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'10) (2010)