KLEEF: Symbolic Execution Engine (Competition Contribution)

Aleksandr Misonizhnik, Sergey Morozov, Yurii Kostyukov(✉), Vladislav Kalugin, Aleksei Babushkin, Dmitry Mordvinov, and Dmitry Ivanov

[1]RnD Toolchain Labs, Huawei, Shenzhen, China
`kostyukov.yurii@gmail.com`

Abstract. KLEEF is a complete overhaul of the KLEE symbolic execution engine for LLVM, fine-tuned for a robust analysis of industrial C/C++ code. KLEEF natively handles complex data structures, such as trees, linked lists, and dynamically allocated arrays, via lazy initialization and symcrete values. KLEEF has fine-tuned modes for both maximal test coverage generation and reproducing error traces, in particular reaching a specific point in the program. In the paper, we describe the above features and a competition configuration of KLEEF.

Keywords: Symbolic Execution · Lazy Initialization · KLEE Fork.

1 Test-Generation Approach

KLEEF is a complete overhaul of the KLEE [11,4] symbolic execution engine. We first describe how KLEE works, then we describe our enhancements over it.

1.1 Symbolic Execution in KLEE

As a *symbolic interpreter* [1], KLEE runs a program on a *symbolic memory*, which maps program locations to symbolic values, representing sets of concrete values. When it meets a branching instruction, it adds target instructions to a queue and after each executed instruction it decides which instruction execute next. Symbolic interpreter collects all conditions from branching instructions in a *path constraint*. It is a formula, which may be either unsatisfiable (if the path is infeasible) or satisfiable, and have multiple solutions. Each solution gives a concrete test, which would visit the corresponding path. A symbolic interpreter usually relies on an *SMT solver* (like Z3 [8]) to get solutions of path constraints.

The KLEE engine is split into two logical parts. The first part is a symbolic interpreter, which takes a symbolic state, executes one instruction, and produces new states. The second part is a *searcher*, which chooses the next symbolic state to execute according to a strategy, specified by input options, e.g., BFS or DFS.

[*]Y. Kostyukov—Jury member.

1.2 Our Enhancements over KLEE

We enhanced KLEE with *support for arbitrary data structures* such as trees and linked lists by implementing **lazy initialization** [7]. If KLEE dereferences a symbolic pointer, it forks the symbolic state into many: each one assumes that the pointer refers to one of the existing locations in the memory. In KLEEF we also fork one extra state, where the pointer refers to a fresh, lazy initialized symbolic object, which is distinct from all other object of the current symbolic memory. If there are not enough objects in the memory, KLEEF will create a new one and continue execution while KLEE will not. In the configuration used at the competition we only create lazy initialized symbolic objects for symbolic pointers without forking the state into existing locations beforehand.

We improve KLEE with **symcretes** [10], which help to support dynamically allocated arrays (with symbolic sizes) and external calls. KLEEF thus *supports detecting buffer overflows.* A symcrete is a pair of *sym*bolic value and its con*crete* instance valid in the current context. The concrete part of symcrete values is derived from the model of a path constraint. It stays the same if the solver can find a model for concretized constraints. Having failed, the concretization will be updated by values from the model for the original constraints. When a logical solver receives a query with a symcrete, an equality between the symbolic and concrete parts of the symcrete are added to the query. This helps the solver to solve the query, as a part of the model is already specified in the symcrete. KLEEF thus handles dynamically allocated arrays by making array size and address symcretes. KLEEF uses the solver to minimize possible array size and sparse storage for arrays, so that the entire process does not blow up.

We have implemented **searchers optimized** specifically for maximizing coverage and reaching the error target. That is, KLEEF has *targeted searcher* and *guided searcher* which maximize coverage and error reachability, similar to [3]. The targeted searcher uses the shortest path based algorithm to choose the nearest execution state to the target location. Each execution state carries a set of targets. A guided searcher manages a bunch of targeted searchers with different targets and chooses states from every targeted searcher in interleaved manner.

KLEEF improves over KLEE in **constraint solving** by caching unsatisfiability cores, interning symbolic expressions, tracking constraints during simplification to detect conflicts and using an SMT solver incrementally. In KLEEF we added support for Bitwuzla [9] SMT solver, which performs significantly better on Test-Comp benchmarks. For example, KLEEF with Z3 achieves 2430 points running for 30 seconds on Test-Comp 2023 benchmarks, while KLEEF with Bitwuzla achieves 2560 points within the same time limit.

2 Architecture

KLEEF has the same architecture as KLEE [4]. KLEEF is implemented in C/C++ and relies on the LLVM infrastructure. KLEEF supports STP [5], Z3 [8] and Bitwuzla [9] SMT solvers for checking constraint satisfiability.

3 Strengths and Weaknesses of the Approach

KLEEF took 3rd place in Test-Comp 2024 (Overall) [2], which is impressive as it is a pure symbolic execution engine. That is, it could get even better results if paired with fuzzing or other techniques.

The main reasons for our **advancement in coverage** category are as follows. First, it is a smart searcher which guides the symbolic execution towards uncovered branches. Second, it is fast constraint solving, incorporating a number of caching techniques and solver incrementality. Third, the engine handles allocations with a symbolic size without concretization by using symcrete values.

The main reasons for our **advancement in error reaching** category include a smart searcher guiding the execution towards an error and elimination of syntactically unreachable paths in CFG.

Note that KLEEF took less points than KLEE in error reaching category. KLEEF has more solved benchmarks, yet this number is normalized across subcategories. As KLEEF solves less benchmarks on SoftwareSystems-BusyBox-MemSafety and SoftwareSystems-OpenBSD-MemSafety subcategories than KLEE, we got less points in the error reaching category in total. Poor performance on these two subcategories is due to bugs in KLEEF: it generated a few tests which were not reproduced by the validation system.

4 Tool Setup and Configuration

4.1 How to Use KLEEF

In order to **run the competition version** from the command line, one should get the archive with binaries from Zenodo[1] and follow the `README` inside.

In order to **generate a test coverage** for a project **without configuring** KLEEF manually, one should use a user-friendly wrapper UnitTestBot C/C++ [6,12]. It allows KLEEF to be run in VS Code and JetBrains CLion.

In order to **build KLEEF from sources**, one should install LLVM, clone KLEEF from GitHub[2] and run `build.sh` script in the repository root.

4.2 Competition Configuration

KLEEF participates in both Cover-Error and Cover-Branches categories.

Common Parameters. Parameters `--strip-unwanted-calls`, `--delete-dead-loops=false`, `--mock-all-externals` are used to (de)activate necessary LLVM passes to simplify bitcode for a symbolic execution. A parameter `--external-calls=all` allows function calls with symbolic arguments. An option `--libc=klee` makes KLEEF support an extended number of external functions.

Parameters `--cex-cache-validity-cores`, `--use-forked-solver=false`, `--solver-backend=bitwuzla-tree`, `--max-solvers-approx-tree-inc=16` are used to cache unsatisfiability cores and call a Bitwuzla solver incrementally.

[1] https://doi.org/10.5281/zenodo.10202734
[2] https://github.com/UnitTestBot/klee

Parameters `--symbolic-allocation-threshold=8192, --skip-not-lazy-initialized, --use-sym-size-alloc` are used to tune lazy initialization and dynamically allocated arrays.

A parameter `--fp-runtime` adds a floating point support. Parameters starting with `--allocate-determ` activate X86 support. An option `--x86FP-as-x87FP80` adds emulation of X86 floating points as extended 80 bit floating points.

Finally, `--max-memory` and `--max-time` fix memory and time limit.

Parameters for Cover-Error. An option `--optimize=true` simplifies code before execution, e.g., it joins some branches to multiple blocks into selection instructions. Options `--search=dfs --search=bfs` make KLEEF interleave between DFS and BFS. Options `--function-call-reproduce=reach_error, --exit-on-error-type=Assert` make KLEEF run towards `reach_error` function and fail only there. An option `--dump-states-on-halt=unreached` permits KLEEF to generate tests for unfinished paths.

Parameters for Cover-Branches. A parameter `--track-coverage=all` makes KLEEF track coverage by both branches and instructions. Options `--optimize=false` and `--optimize-aggressive=false` disable optimizations which decrease coverage. Options `--use-iterative-deepening-search=max-cycles, --max-cycles-before-stuck=15` activate an iterative-deepening mode of execution on a number of executed loop cycles. A parameter `--max-solver-time=10s` fixes a time limit for an SMT solver. An option `--only-output-states-covering-new` makes KLEEF only generate tests which increase coverage. Options `--search=dfs, --search=random-state` make KLEEF interleave between DFS and taking a random state. A parameter `--dump-states-on-halt=all` makes KLEEF generate tests for the symbolic states remaining in the end. Options `--cover-on-the-fly, --delay-cover-on-the-fly, --mem-trigger-cof` start on the fly test generation after approaching memory cap.

5 Software Project and Contributors

More information about KLEEF is available on its website[3]. KLEEF is an open-source piece of software which you could contribute to at GitHub[4].

The key developers are the authors of this paper affiliated with RnD Toolchain Labs, Huawei, Shenzhen, China. The authors have decent experience in the implementation of research and industrial symbolic execution engines.

6 Data-Availability Statement

A binary version of KLEEF participating in the competition is publicly available[5]. Also, its source code is available on GitHub[6].

[3] https://toolchain-labs.com/projects/kleef.html
[4] https://github.com/UnitTestBot/klee
[5] https://doi.org/10.5281/zenodo.10202734
[6] https://github.com/UnitTestBot/klee/releases/tag/testcomp24

References

1. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A Survey of Symbolic Execution Techniques. ACM Comput. Surv. **51**(3) (2018)
2. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. Springer (2024)
3. Burnim, J., Sen, K.: Heuristics for Scalable Dynamic Test Generation. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. pp. 443–446 (2008). https://doi.org/10.1109/ASE.2008.69
4. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings. pp. 209–224. USENIX Association (2008), http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
5. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification. pp. 519–531. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
6. Ivanov, D., Babushkin, A., Grigoryev, S., Iatchenii, P., Kalugin, V., Kichin, E., Kulikov, E., Misonizhnik, A., Mordvinov, D., Morozov, S., Naumenko, O., Pleshakov, A., Ponomarev, P., Shmidt, S., Utkin, A., Volodin, V., Volynets, A.: UnitTestBot: Automated Unit Test Generation for C Code in Integrated Development Environments. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 380–384 (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00107
7. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 553–568. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
8. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
9. Niemetz, A., Preiner, M.: Bitwuzla. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II. Lecture Notes in Computer Science, vol. 13965, pp. 3–17. Springer (2023). https://doi.org/10.1007/978-3-031-37703-7_1, https://doi.org/10.1007/978-3-031-37703-7_1
10. Pandey, A., Kotcharlakota, P.R.G., Roy, S.: Deferred Concretization in Symbolic Execution via Fuzzing. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 228–238. ISSTA 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3293882.3330554, https://doi.org/10.1145/3293882.3330554
11. The KLEE Team: KLEE Symbolic Execution Engine (2009), http://klee.github.io/
12. The UnitTestBot C/C++ Team: UnitTestBot C/C++ (2021), https://www.utbot.org/cpp