Deterministic State Space Exploration

Martin Nowack[1][0000−0002−1177−0233]

Imperial College London, United Kingdom
m.nowack@imperial.ac.uk

Abstract. Symbolic execution is a very active research area due to its ability of automatic test-case generation, bug finding, and many more applications. Despite the many recent proposals for improvements, we find it hard to quantify how state of the art progresses. Even if we only compare a single symbolic execution engine (A) and the same engine with some modifications ($A*$), running both implementations with the same benchmarks, concluding on the true cause of differences in behaviour is hard.
While artifacts (here benchmarks, measurements, data, and implementation) provide an invaluable base for reproducable research, the implementation itself is often treated as a blackbox. Changes in behaviour between two implementations are quantified based on coarse-grain difference in benchmark behaviour, i.e., changed code coverage or changed execution time. We propose a complementing fine-grain approach that helps to understand implementations better — not only supporting reproducable research but also the development of the implementation in the first place.
In this paper, we analyse KLEE, a well established dynamic symbolic execution engine for C and C++, and identify significant challenges that make it hard to evaluate and compare different implementations. We identify different reasons that are implementation-specific to KLEE but often can be transferred to other symbolic execution engines and propose ways to fix them. We propose *Deterministic State-Space Exploration* as one technique that helps to quantify and validate incremental improvements of symbolic execution engines.

Keywords: Determinisem · Symbolic Execution

1 Introduction

Scaling symbolic execution for test-case generation and bug finding got much traction in recent years. With a lot of new symbolic engines available and extensions proposed, evaluating their impact is hard. For example, KLEE [CDE08], a popular symbolic execution engine for C and C++ code based on LLVM, is available as open-source software and actively maintained. It is used as a baseline or extended in more than 200 publications [20]. Still, evaluating and comparing the effectiveness of different approaches and presented techniques that have been developed using KLEE is hard even if access to software and experiment artefacts is available.

In general, quantitatively measuring and comparing the efficiency of two approaches is hard. If two implementations behave differently, the actual cause can be either the implementations of the proposed research idea themselves, the system and context within which they are executed, the subjects they are tested with, or a mixture of those reasons. As a consequence, the measured effect cannot be necessarily attributed to the correct cause, i.e., the changed implementation.

To evaluate the performance of a system in general, we measure the *work,* W a system does and how much *time* T it took to finish it. The larger the ratio ($P = \frac{W}{T}$) is, the more performant is a system. However, a requirement for this is to define *work* that the tested system has to do. If two systems use the same time limits, the one that finished more work is more performant. Unfortunately, this assumes that a workload can be devided into equally-sized and equally-hard problems.

The example throughout this paper is measuring the effectiveness of different symbolic execution implementations, which is hard. The reason is the many control-flow paths a tested benchmark application can have. Instead of having a benchmark with a specific set of test input that triggers a well-defined path (e.g., SPEC benchmarks [Hen06]), the symbolic execution engine can potentially explore arbitrarily sized sets of paths through the application. This behaviour, combined with the state-space explosion problem, will often not allow the analysis of the whole applications in a given timeframe. As such, this impacts the following questions: Which of the paths are explored by symbolic execution? Has a path been fully explored? Or, in which order are they explored? The computational costs associated with a path can vastly vary for different paths. As a result, if two implementations analyse different parts of a tested application, the overall characteristics of the executed instructions can differ broadly. For a symbolic execution engine, exploring different code paths of a testd software will have an impact on the number of states being handled and the states' path constraints for the solver. Therefore, measuring the effectiveness of two different approaches to compare them needs to take these properties into account.

Listing 1.1: Example code with two paths of different costs. Taking the true branch in line 2 will generate more complex constraints utilising multiple symbolic variables. While taking the false branch, just one is required.

```
1 int a = symbolic;
2 if (a > 5) {
3   int b = symbolic;
4   a += a % b;
5 }
6 return a;
```

In the example code above (Listing 1.1), we have two variables (`a` and `b`) that have arbitrary symbolic values as an input. Symbolic execution can exercise two paths: one with the condition `a > 5` (Line 2) true and one with false. If true, the result of the return instruction will be more complex. And depending in which

context it is used, this will have an impact on the time spent in solving or the number of states forked.

While testing different search-space exploration strategies is important [HMZ12; ZJX18; Xie+09], measuring their effectiveness is possible if they are handled as randomized algorithms. Using the correct statistical tests to assess them boils down to using high numbers of iterations with different inputs and applications.

In the following section, we give an overview of the state-of-the-art of comparing symbolic execution engines (section 2) and the limitations. We detail multiple causes that make it harder to compare different implementations (section 4). We introduce *Deterministic State-Space Exploration* (section 3) as a new approach to compare different symbolic execution implementations and the non-deterministic sources. We give an overview of requirements and changes that have been necessary to support such a comparison strategy in KLEE.

For this paper, we explicitly do not cover non-deterministic behaviour resulting from the environment (like different OS scheduling decisions, ASLR or additional load on a testing machine) or non-deterministic behaviour inherent to the application itself (like the test applications using a random number generator, time, free disk space,...). Therefore, if we have a single implementation of a symbolic execution engine that analyses one application with only one set of arguments, running this setup multiple times should have the same deterministic result. We assume that we have such a testing setup and build on top of it, reducing the environmental impact to a minimum. Research in this area takes care of precisely those issues[BLW19]. Most importantly, we focus on comparing symbolic engines using a fixed, precise workload, targeting mainly incremental changes of symbolic execution engines, i.e., one implementation of a symbolic execution engine A is extended, resulting in a version $A*$ and both versions are directly compared.

2 State of the art: Comparing Improvements in Symbolic Execution

To measure the effectiveness of different symbolic execution approaches, two major methods of comparing them have been utilised: *time-based comparisons* and *coverage-based comparisons*[RED16].

For the *time-based comparisons*, a specific workload is given (e.g., execute n instructions, fork m times) and the time is measured how long it takes to finish it. Still, one implementation might take an unreasonably long time to finish. To avoid arbitrarily long experiment runs or discard those results, time-out based measurements have been proposed (Rizzi et al.[RED16]) that set a maximum time an experiment can take. Beyond this, the experiment is marked as timed-out.

For the *coverage-based comparison*, two methods A and B are compared on how much of the code could be covered by both approaches using the same execution time. The higher coverage should indicate the method that is better.

Unfortunately, those methods are vulnerable to multiple issues, especially in the context of symbolic execution. We will detail the issues in the following paragraphs.

Coverage Criterium Coverage describes what part of the code has been exercised. Different metrics focus on different units for describing this property. For example, line coverage describes how many lines of code have been executed. Branch coverage describes how many branch targets have been taken, or MCDC coverage which predicate combinations have been exercised for a branch condition.

Using coverage as a criterium can be insightful; still, often, coverage changes are only compared by quantifying their changes ($+/-X$ improvement in coverage or if coverage is similar) but not how they differ in detail. If two implementations are compared, how much of the covered code is common for both methods? Where do they differ? As different code paths will vary in costs, like time spent in solving path constraints or forking of states, comparing only the nominal values of coverage is not a valid approach to conclude which implementation is better.

Besides that, if we look at how coverage evolves over time, we see that often much coverage is reached at the beginning of the symbolic execution as it is easy for a symbolic engine to reach uncovered code paths — executing any instruction is essentially covering new instructions. Over time, reaching uncovered code becomes more challenging as path constraints become more complex and uncovered, reachable code is further away.

Another aspect is how coverage is calculated. Often, this happens in a two-stage process. First, a symbolic execution engine uses an internal tracker to remember which part of the code has been already covered. For example, KLEE has an internal coverage tracker and marks execution state with *covers new* if new code has been reached. A test case is generated with inputs that are able to reach this code. In a second step, an external tool is used (i.e. *gcov* [FSF17]) to use those test cases as input and execute the native version of the application to gather coverage information. The upside of this approach is that, for example, if the generated test case triggers a bug, its validity can be validated independently of the symbolic execution engine. Concerning achieved coverage, the downside is that it often claims more code to be covered, although symbolic execution was not able to reach that code in the first place. For example, if symbolic executes an application, stops the execution the moment the first conditional branch has been reached and generates a single test case, executing the native application will typically cover much more of the application than has been analysed in the first place.

Time-based comparison For the time-based comparison, evaluating different implementations is hard. They are reasonable if the tested application can be fully explored. If not, like for many non-trivial applications, the two implementations can take different control-flow paths in the tested applications. Therefore, the code that is explored is different with different costs associated.

To summarise, the different measurement approaches are not enough to conclude a new technique's impact over a given baseline as they do not imply that the same work has been done.

Statistical Valid Comparison Both comparisons could be improved by running experiments on a broader range of applications or many more execution runs [AB14]. While, we clearly agree that this would be the ideal way forward, we want to emphasise that it is often not feasible to have such a setup. With limited budget, time, and computational resources — especially in the academic context — this is often not achievable. Even if those would be available, the environmental impact is much higher than our suggested strategy.

3 Deterministic Exploration of the State Space

To improve the validity of measurements of two or more symbolic execution engines, we propose *Deterministic Exploration of State Space* that allows us to validate different symbolic execution engines exercising the same precise workload.

Definition 1 (Instruction). *An instruction is the smallest unit of work a symbolic execution engine can execute.*

For example, in the case of KLEE, instruction is an LLVM IR [LA04] instruction.

Definition 2 (State). *A state represents the allocated memory and the collected path constraints along one execution path for software under test.*

Definition 3 (Deterministic Exploration of State Space). *If two symbolic executions compromise the exact same instructions by the same states in the same order.*

With these definitions in place, a workload is an ordered list of instructions generated via the deterministic exploration of the state space. We can compare runs of different symbolic execution engines or the same engine and quantify them, for example, with respect to execution time or resource utilisation like memory consumption or caching solver efficiency.

These definitions are simple, but achieving them is hard and poses several non-trivial challenges.

3.1 Example of the application of deterministic state space exploration

Due to space restrictions, this is a very short version of our use case.

We compared two versions of KLEE that differed only by the SMT solver used for query solving (A: STP [GD07] vs. A*: Z3 [MB08]). We ran different CoreUtils experiments similar to [CDE08] for 30min each application with different

search strategies (Depth-First-Search (DFS), Breadth-First-Search (BFS),Random+Coverage-guided search (Rnd+Cov)).

While we first concluded that Z3 had an edge over STP, we imposed deterministic state space exploration for all the experiments, i.e., for the same application and the same search strategy, the same instructions were executed for both implementations (A:STP, A*:Z3). We noticed that execution time of executing instructions differed and so did solving time. While different solving time would be explicable and expected, different time spent in the actual instruction execution was not due to the deterministic state space exploration.

We found that the cause of the different time spent in executing instructions, was due to KLEE's handling of the solver invocation of STP by forking its process that was highly influenced by KLEE's utilised memory slowing down the actual instruction execution. We fixed this behaviour by mitigating the forking costs. We re-ran all experiments, and as a result, both implementations showed the same time in instruction execution as expected in the first place, changing the result which solver is better in favour for STP.

We used that technique in muliple papers, e.g., [Now19; NTF15; BNC20] not only to show valid performance improvements but also demonstrating correctness with respect to the baseline behaviour.

4 Sources of Non-deterministic Behaviour - KLEE as an Example

We have seen the power of deterministic state-space exploration as a methodology. Unfortunately, applying this approach is not straightforward as one would expect. For our use case, we used KLEE, which shows several non-deterministic cases of behaviour even in the context of deterministic testing environments. To support deterministic state-space exploration in KLEE, we had to extend it in several ways. We started with adding support to track executed instructions. Based on this, we debugged differing executions and applied fixes mainly origin in internal datastructures and behaviour that lead to to different search beahviour even for static search strategies like DFS.

We will detail the many building blocks of this approach and the obstacles we had to deal with in a longer version of this paper.

We are convinced that similar behaviour exists in other symbolic execution engines as well.

5 Limitations

This approach's major goal is to specify a workload by using a trace that describes the exact instructions and states that executed them.

While this allows comparing different aspects of a symbolic execution engine, like state representation, solver efficiency and instruction execution efficiency, it does not allow comparing different state-space exploration strategies.

6 Conclusion

Testing symbolic execution is hard. The potentially unlimited number of paths a program can take and the different computational expenses each path has, make it often impossible to explore the whole program fully under a specific time budget. As a result, comparing different implementations of a symbolic execution engine might lead to incorrect conclusions as they execute different workloads if different parts of the search space are explored.

We demonstrated such behaviour using a showcase and propose *deterministic space exploration* as one way to improve it. We detailed sources of non-determinism in general and KLEE specifically that had to be addressed to achieve deterministic exploration of states. Many of those changes have been already made available in upstream KLEE to support the wider research community.

This approach's applicability is only for certain incremental changes. However, for cases where it can be applied, it provides an accurate way for researchers to explore new ideas and evaluate existing, concluding on performance and correctness. Moreover, we are convinced that the general approach can be applied beyond symbolic execution for better research replicability.

References

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[Hen06] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Comput. Archit. News* 34.4 (Sept. 2006), pp. 1–17. ISSN: 0163-5964. DOI: 10.1145/1186736.1186737. URL: http://doi.acm.org/10.1145/1186736.1186737.

[GD07] Vijay Ganesh and David L. Dill. "A Decision Procedure for Bit-Vectors and Arrays". In: *Computer Aided Verification*. Ed. by Werner Damm and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 519–531. ISBN: 978-3-540-73368-3. DOI: 10.1007/978-3-540-73368-3_52.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[Xie+09] T. Xie et al. “Fitness-guided path exploration in dynamic symbolic execution”. In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. June 2009, pp. 359–368. DOI: 10.1109/DSN.2009.5270315.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. “Search-Based Software Engineering: Trends, Techniques and Applications”. In: *ACM Comput. Surv.* 45.1 (Dec. 2012). ISSN: 0360-0300. DOI: 10.1145/2379776.2379787. URL: https://doi.org/10.1145/2379776.2379787.

[AB14] Andrea Arcuri and Lionel Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering”. In: *Software Testing, Verification and Reliability* 24.3 (2014), pp. 219–250. DOI: 10.1002/stvr.1486. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1486. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486.

[NTF15] Martin Nowack, Katja Tietze, and Christof Fetzer. “Parallel Symbolic Execution: Merging In-Flight Requests”. In: *Hardware and Software: Verification and Testing*. Ed. by Nir Piterman. Cham: Springer International Publishing, 2015, pp. 120–135. ISBN: 978-3-319-26287-1.

[RED16] E. F. Rizzi, S. Elbaum, and M. B. Dwyer. “On the Techniques We Create, the Tools We Build, and Their Misalignments: A Study of KLEE”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. May 2016, pp. 132–143. DOI: 10.1145/2884781.2884835.

[FSF17] FSF. *GCOV (GNU Compiler Collection)*. July 2017. URL: https://gcc.gnu.org/ (visited on 07/21/2017).

[ZJX18] Z. Zhu, L. Jiao, and X. Xu. “Combining Search-Based Testing and Dynamic Symbolic Execution by Evolvability Metric”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Sept. 2018, pp. 59–68. DOI: 10.1109/ICSME.2018.00015.

[BLW19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. “Reliable benchmarking: requirements and solutions”. In: *International Journal on Software Tools for Technology Transfer* 21.1 (Feb. 2019), pp. 1–29. ISSN: 1433-2787. DOI: 10.1007/s10009-017-0469-y. URL: https://doi.org/10.1007/s10009-017-0469-y.

[Now19] Martin Nowack. “Fine-Grain Memory Object Representation in Symbolic Execution”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2019, pp. 912–923. DOI: 10.1109/ASE.2019.00089.

[BNC20] Frank Busse, Martin Nowack, and Cristian Cadar. “Running symbolic execution forever”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 63–74.

[20] *Publications and Systems Using KLEE*. 2020. URL: https://klee.github.io/publications/ (visited on 06/22/2020).