Report on the KLEE workshop

Written by Julien Vanegue and Peter Martin
Bloomberg LP (New York, NY, USA)

The First International KLEE Workshop on Symbolic Execution
https://srg.doc.ic.ac.uk/klee18/
April 18-19 2018 in Imperial College, London, UK
82 participants

We attended the KLEE workshop for two full days of talks organized by Cristian Cadar's research group at Imperial College London. KLEE is open source software (http://klee.github.io) which performs symbolic execution of LLVM bytecode. KLEE has identified hundreds of security issues in coreutils and other open source software and can report on detailed coverage, unlike traditional black box testing tools (such as fuzz testers).

Roughly half of the talks at the workshop were delivered by senior figures from the industry and academia; these represented the work of entire research teams in one hour slots. The other half of the workshop was dedicated to 20 minutes talks about work-in-progress research projects. All talks were very interesting and covered a wide variety of topics in the field of symbolic execution, as well as related techniques.

Two major themes that were developed during the workshop were the interfacing of symbolic execution and fuzz testing, as well as symbolic execution on heap memory.

(A) Interface Symbolic execution with Fuzz testing

Several teams from Fujitsu, Samsung, and the National University of Singapore demonstrated their combinations of tools, such as AFL and KLEE, to increase coverage and bug count. A constant theme was the failure of KLEE to find new bugs on its own, while fuzzing shows very good rates of success. It seems obvious that fuzzing and SYMEX are good at different things: while SYMEX gives metrics and allows for enhanced coverage, the sole use of SYMEX is not adapted to bug finding. Most teams report no bugs or very small amount of bugs found with SYMEX only, while fuzzing can find more bugs, but coverage plateaus very fast (~ in 1h). The concrete benefit in combining the two was a big take away from the workshop.

(B) Symbolic Execution for Heap Memory

Several research students are working on improving heap support in KLEE. Most of the effort is being spent to reduce state space explosion. Most people focused on the problem of symbolic pointers. Nobody said they had solved the problem of symbolic size; yet, everybody noticed that problem and did not provide any compelling solution.

Talk summaries

All talks were greatly informative in their own way and made the workshop a success. We apologize if we forgot to stress a particular contribution of any of the talks or if we misunderstood or misrepresented the work of the authors in some way. At this moment, the KLEE workshop presentations are not available online and these notes are compiled from our recollections after attending in person. Feel free to contact us if you perceive something to be outrageous that should absolutely be corrected. We write this report in our free time and appreciate your understanding in the limitations of the format.

(1) KLEE annotations case study on open source software

(Oxford University)

John Galea and Sean Heelan presented an interesting case study where they added manual annotations using klee_assume() to cut down on state space explosion. They used between 2 and 10 annotations per program to help KLEE drive symbolic execution on the interesting part of the code. While both unsound and incomplete, this approach found ~40 new bugs in tcpdump (which were duplicate finds from AFL, and already patched). This is quite similar to a contract-based approach in static analysis except the annotations were not used to define properties, but assume constraint to avoid black-hole paths where KLEE fails to get enough coverage. A given example was an image decoding library which uses loops to populate some internal data structures; watching a concrete execution and adapting the library to use a precomputed version of the same data (at larger cost in storage) allowed SYMEX to continue beyond what would otherwise have been a sink that would have prevented other more interesting branches from being reached. This could have been more convincing if the survey had produced some more coherent categories of issues / suggested strategies for more automated fixes.

(2) ConcFuzzer: sanitizer-guided hybrid fuzzing

(Baidu, USA)

AFL's code-coverage-guided fuzzing produced concrete results, which were then used to guide concolic execution in KLEE. Results from KLEE are then fed back to AFL to provide the basis for further fuzzing. One particular feature of interest is that they leverage clang sanitizer guards as part of instrumentation. Clang can introduce guards which will catch and warn on particular bad situations (Address Sanitizer, Memory Sanitizer, Undefined Behavior Sanitizer). Their approach was to invert those guards and guide execution down paths which would have been caught by these sanitizers. This finds problem inputs that would trigger those cases: specifically, concrete inputs that lead to undefined behavior, memory corruption, integer overflow, etc.

(3) KLOVER: Symbolic execution for C++

(National University of Singapore)

KLOVER is a tool developed by Fujitsu in the last 7 years. It is a fork of KLEE that supports two major features: 1) 30+ LLVM instructions for C++ and 2) the use of a String Solver "PASS" (Parameterized Array Based String Solver) to make strings first class citizens in the solver (similar to Z3str). Unfortunately KLOVER is not public and will not be any time soon, according to the director of research who presented the work.

(4) Symbolic Execution for Directed Search and Specification Inference

(National University of Singapore)

Abhik Roychoudhury presented on a range of different research done by his research group. Not particularly relevant to SYMEX, but one project he mentioned which was interesting was https://github.com/aflgo/aflgo. This can get significantly better results than vanilla AFL by guiding fuzzing to sections of the program nominated by the user. The way this works under the hood: the program is instrumented at compile-time not only to report back the code paths covered, but also for each branch, the distance (in the control flow graph) to a code location of interest. The modified AFL can then use the distance metric to prioritize the seeds that are getting the program closest to "interesting" locations.

(5) Ranged symbolic execution

(University of Texas at Austin, USA)

Sarfraz Khurshid presented what could be worth exploring more for distributing SYMEX among cloud machines. The idea is to find ways to encode "ranges" of code paths to be explored. A benefit of this is that it allows workloads to be parallelized and each range explored by an independent worker. He described some of the difficulty in doing this efficiently (i.e., selecting independent non-overlapping workloads so that one worker explore branches which another has pruned out of the tree).

Paper: "Using Test Ranges to Improve Symbolic Execution" https://link.springer.com/chapter/10.1007%2F978-3-319-77935-5_28

(6) Chopper: Chopped symbolic execution

(UTelAviv and Imperial College)

Paper published at ICSE'18:
https://srg.doc.ic.ac.uk/files/papers/chopper-icse-18.pdf

An interesting talk: the basic idea is to find ways to skip expensive/uninteresting code paths which would otherwise be in-lined with cheaper paths (making those more cheaply explored paths harder to reach). They introduce recovery checkpoints, skip over exploration of some branches and refer back to those checkpoints later (only executing the branches from the checkpoint which are known to affect state of the code after the skipped section).

(7) Symbolic Execution to reproduce field failures

(Georgia Tech, USA)

First talk (Bugredux/f3): This was an interesting use of SYMEX~~ that we could also try~~. They take field reports (i.e., stack traces from crashes) and symbolically execute (constraining to the stack path) to produce input which can reproduce the bug. This is not necessarily an exact reproduction, but is input which produces a crash with the same stack. They got better results in some cases by removing stack frame data (unconstraining the search for better performance).

(8) Symbolic execution for probabilistic analysis

(Imperial College, London, UK)

The goal of this work is to approximate the satisfiability of a given first order logic formula by sampling variable values and figuring out which values make the formula true and which do not. The ratio of cases where the formula is true vs. false is a probabilistic value of the truthness of the formula. The author used Monte-Carlo methods to approximate the truth values for complex formula. The niche application is when programs can act probabilistically, such as in the presence of random numbers or for aerospace software which can probabilistically fail based on cosmic rays and the like.

(9) Symbolic Execution of Stateful Programs

(Samsung kNOX team, Palo Alto, USA)

The Samsung team uses KLEE to perform symbolic execution of each function individually and store state across procedures. They mix KLEE and AFL to find security bugs. While the tool is not yet released, I was told they have obtained clearance to release it as open source, so we will be watching GitHub to give the tool a try soon.

(10) Advanced use of Symbolic Execution: JavaScript

(Royal Holloway, London, UK)

Blake described a framework for symbolic execution on JS. He talked through some challenges involved in attempting SYMEX on JavaScript. Mainly: syntax for complexes regexes with backward references, Dynamic types, async calls, use of APIs like eval() to dynamically evaluate JS on the fly. While much of those remain unsolved, the tool has still uncovered bugs in various JS libraries. The author acknowledged that a lot of work is still ahead to get a larger level of support.

GitHub: https://github.com/ExpoSEJS/ExpoSE

(11) KLEE constraint solvers optimization

(Various talks)

At least 3 short talks were dedicated to improving performance of solving. While useful, these talks did not necessarily contain any creative insights and relied on heuristics or interpolants to do the job. It was also demonstrated that the use of several solvers (e.g., Z3 + STP) was more efficient than using a single solver, as some solvers are better for smaller formulas while others are better for more complex ones. One of the techniques mentioned was launching two solvers in parallel and using only the results of the solver that returned faster (e.g., run 2 processes in parallel and discard the solver that was slower). One group had developed a good heuristic which chooses from 20 cached, previously solved results. This lets them improve performance by reusing earlier results where possible, in place of an expensive solver execution (where earlier results related

to the same constraints). Another talk advocated using machine learning to select the right solver for the query. The analysis would extract features of the input to the SMT solver, then train a model (beforehand or on a proportion of queries) which can later select a solver back-end that is likely to be quickest given the properties of the query.

(12) Hardware-assisted Symbolic Execution

(UEdinburgh, UWash, and UMichigan)

This rather interesting talk demonstrated the use of a recent CPU feature which can record all executed instructions in a ring buffer, which is useful for time travel debugging and the like. This is still quite early work. However, the project is open source and could be useful for performance improvements as well as replay.

Mentioned:
* Mozilla RR (record and replay framework
https://github.com/mozilla/rr
* Intel Processor Trace (PT) on Andi Kleen's blog
http://halobates.de/blog/ (experimenting with this on Linux)

(13) Debugging P4 programs with VERA

(Polytechnic University of Bucharest, Romania)

This work by students at the Polytechnic University of Bucharest presented techniques to test equivalences of P4 programs (firewall / router logic) for medium size routers (10K lookup tables was the biggest router they tried it on). Equivalence is interesting as this allows filtering to happen when a packet matches a rule. This was a refreshing talk on using SYMEX for networking.

(14) Tracer-X: a new symbolic execution engine

(National University of Singapore)

Tracer-X is a KLEE-like tool where techniques such as weakest precondition calculus (backward analysis) is used and combined with traditional forward analysis to "meet in the middle." The performance and coverage were reportedly better than KLEE. However, it looks like the tool was possibly less scalable than KLEE as it seems to perform more explicit state space enumeration.

(15) Advanced coverage criteria

(Center for Atomic Energy (CEA), France)

This was an interesting talk. While not related to symbolic execution, it introduced interesting metrics for testing, namely context-based coverage (rather than path-based or instruction-based coverage). The author's work on x86 binary code and allow for reporting on coverage across contexts for a given function (e.g., count the function as additional coverage if that function was called from a different caller, with different parameters, etc.).