Legion: Best-First Concolic Testing
(Competition Contribution)

Dongge Liu[⋆1], Gidon Ernst[⋆⋆2], Toby Murray[1], and Benjamin I.P. Rubinstein[1]

[1] University of Melbourne, Australia
donggel@student.unimelb.edu.au
[2] LMU Munich, Germany
gidon.ernst@lmu.de

Abstract. Legion is a grey-box coverage-based concolic tool that aims to balance the complementary nature of fuzzing and symbolic execution to achieve the best of both worlds. It proposes a variation of *Monte Carlo tree search (MCTS)* that formulates program exploration as sequential decision-making under uncertainty guided by the best-first search strategy. It relies on *approximate path-preserving fuzzing*, a novel instance of constrained random testing, which quickly generates many diverse inputs that likely target program parts of interest. In Test-Comp 2020 [1], the prototype performed within 90% of the best score in 9 of 22 categories.

Keywords: Symbolic Execution, Fuzzing, Monte Carlo Search

1   Test-Generation Approach

Coverage testing aims to traverse all execution paths of the program under test to verify its correctness. Two traditional techniques for this task, *symbolic execution* [6] and *fuzzing* [7] are complementary in nature [5].

Consider exploring the program Ackermann02 in Fig. 1 from the Test-Comp benchmarks as an example. Symbolic execution can compute inputs to penetrate the choke point (line 10) to reach the "rare branch" (lines 14/15), but then becomes unnecessarily expensive in solving the exponentially growing constraints from repeatedly unfolding the recursive function ackermann. By comparison, even though very few random fuzzer-generated inputs pass the choke point, the high speed of fuzzing means the "rare branch" will be quickly reached.

The following research question arises when exploring the program space in a conditional branch: Will it be more efficient to focus on the space under the constraint, or to flood both branches with unconstrained inputs, to target the internals of log(m,n) in line 11 at the same time?

Legion[3] introduces *MCTS-guided program exploration* as a principled answer to this question, tailored to each program under test. For a program like

---

[3] The name Legion comes from the Marvel fictional character who changes personalities for different needs, to reflect the strategy adaption depending on the program.

```
1   int ackermann(int m, int n) {
2     if (m==0) return n+1;
3     if (n==0) return ackermann(m-1,1);
4     return ackermann(m-1,ackermann(m,n-1));
5   }
6
7   void main() {
8     int m = input(), n = input();
9     // choke point
10    if (m < 0 || m > 3) || (n < 0 || n > 23) {
11      log(n,m);                  // common branch
12      return;
13    } else {
14      int r = ackermann(m,n); // rare branch
15      assert(m < 2 || r >= 4);
16    }
17  }
```


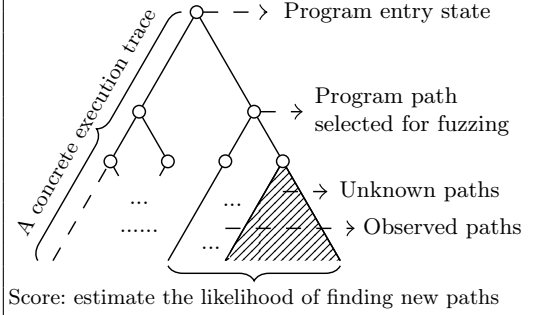Fig. 1: `Ackermann02.c`   Fig. 2: MCTS-guided fuzzing in LEGION

Fig. 2, LEGION estimates the expectation of finding new paths by the UCT score (upper confidence bound for trees), a successful approach for games [3], aiming to balance *exploration* of program space (where success is still uncertain) against *exploitation* of partial results (that appear promising already). Code behind rare branches is targeted by *approximate path-preserving fuzzing* to efficiently generate diverse inputs for a specific sub-part of the program.

LEGION's MCTS iteratively explores a tree-structured search space, whose nodes represent partial execution paths. On each iteration, LEGION selects a *target* node by recursively descending from the root along the highest scoring child, stopping when a parent's score exceeds its childrens'. A node's score is based on the ratio of the number of distinct vs. all paths observed passing through it, but nodes selected less often in the past are more likely to be chosen. Then, approximate path-preserving fuzzing is applied to explore the target node. The resulting execution traces are recorded and integrated into the tree.

Approximate path-preserving fuzzing (APPF) quickly generates inputs that likely follow the target program path, and therefore is crucial for LEGION's efficiency. LEGION's APPF implementation extends the QUICKSAMPLER [4] technique, which is a recent mutation-based algorithm that expands a small set of constraint solutions to a larger suite of likely solutions. LEGION extends QUICK-SAMPLER from propositional logic to bitvector path constraints.

2   Tool Description & Configuration

We implemented LEGION as a prototype in Python 3 on top of the symbolic execution engine angr [8]. We have extended its solver backend, claripy, by the approximate path-preserving fuzzing algorithm, relying on the optimizer component of Z3 [2]. Binaries are instrumented to record execution traces.

Installation. Download and unpack the competition archive (commit `b2fc8430`):
https://gitlab.com/sosy-lab/test-comp/archives-2020/blob/master/2020/legion.zip

LEGION requires Python 3 with python-setuptools installed, and gcc-multilib for the compilation of C sources. Necessary libraries compiled for Ubuntu 18.04

are included in the subfolder `lib` (modified versions of `angr`, `claripy` and their dependencies). The archive contains the main executable, `Legion.py`, and a wrapper script, `legion-sv` that includes `lib` into `PYTHONPATH`. The version tag is `0.1-testcomp2020`, options can be shown with `python3 ./Legion.py --help`.

Configuration. In the competition, we ran `./legion-sv` with these parameters:

`--save-tests`	save test cases as `xml` files in Test-Comp format
`--persistent`	keep running when no more symbolic solutions are found (mitigates issue with dynamic memory allocations)
`--time-penalty 0`	do not penalise a node for expensive constraint-solving (experimental feature, not yet evaluated)
`--random-seed 0`	fix the random seed for deterministic result
`--symex-timeout 10`	limit symbolic execution and constraint solving to 10s
`--conex-timeout 10`	limit concrete binary execution to 10s

In the category `cover-branches`, we additionally use this flag:

`--coverage-only`	don't stop when finding an error

Finally, `-32` and `-64` indicate whether to use 32 or 64 bits (this affects binary compilation and the sizes for nondeterministic values of types `int`, . . .).

Participation. LEGION participates in all categories of Test-Comp 2020.

Software Project and Contributors. LEGION is principally developed by Dongge Liu, with technical and conceptual contributions by all authors of this paper. LEGION will be made available at https://github.com/Alan32Liu/Legion.

3   Discussion

LEGION is competitive in many categories of Test-Comp 2020, achieving within 90% of the best score in 2 of 9 error categories and 7 of 13 coverage categories.

```
1   void main( ) {
2     int N=100000, a1[N], a2[N], a3[N], i;
3     for (i=0; i<N; i++)
4       a1[i] = input(); a2[i] = input();
5     for(i=0; i<N; i++) a3[i] = a1[i];
6     for(i=0; i<N; i++) a3[i] = a2[i];
7     for(i=0; i<N; i++) assert(a1[i] == a3[i]);
8   }
```

Fig. 3: `standard_copy2_ground-1.c`

LEGION's instrumentation and exploration algorithm can accurately model the program. Consider the benchmark `standard_copy2_ground-1.c` in Fig. 3. With a single symbolic execution through the entire program over a trace found via initial random inputs, LEGION understands that all guards of the `for` loops can only evaluate in one way, and so omits them from the selection phase. It does discover that the assertion inside the last loop contributes interesting decisions, however, and will come up with two different ways to evaluate the comparison `a1[i] == a3[i]`, one of which triggers the error. With such an accurate model in combination with its principled MCTS search strategy, LEGION is particularly good at covering corner cases in deep loops: All other tools failed to score full marks in `standard_copy*_ground-*.c` benchmarks, but LEGION succeeded in 9 out of 18. We can furthermore solve benchmarks where pure constraint solving fails, e.g., when the solver times out on hard constraints of complex paths we label the respective branches for pure random exploration.

While instrumentation provides accurate information on the program, its currently naive implementation significantly slows down the concrete execution of programs with long execution traces. We mitigate this weakness by setting a time limit on the concrete executions. As a consequence, inputs that correspond to long concrete execution are not saved. In the future, we plan to explore Intel's PIN tool, which offloads binary tracing into the CPU with negligible overhead.

LEGION inherits some limitations from `angr` as a symbolic execution backend. Some benchmarks, such as `array-tiling/mbpr5.c`, dynamically allocate memory with a symbolic size that depends on the input. `angr` eagerly concretises this value, producing unsatisfiable path constraints for a feasible execution path. LEGION detects this inconsistency as soon as it encounters the feasible path and omits the erroneous node from selection. This helps e.g. on `bubblesort-alloca-1.c` where LEGION achieved full coverage (in contrast to most other participants) despite the dynamic allocations.

LEGION performed poorly on benchmark sets `bitvector` and `ssh-simplified`. These programs have long sequences of equality constraint that are hard to satisfy with fuzzing. This happens to be an extreme example of the parent-child trade-off that LEGION intends to balance where fuzzing the parent gives nearly no reward. This could potentially be mitigated by decreasing LEGION's exploration ratio in the UCT score, but we have not attempted such fine-tuning.

Another problem is allocations when loop counters or array sizes are randomly chosen very large in 64 bit mode, leading to excessively long concrete execution traces that cause timeouts or memory exhaustion. We plan to periodically prune the in-memory representation of the tree in the future.

References

1. Beyer, D.: Second competition on software testing: Test-comp 2020. In: Proc. of Fundamental Aspects of Software Engineering (FASE). LNCS, Springer (2020), https://www.sosy-lab.org/research/pub/2020-FASE.Second_Competition_on_Software_Testing_Test-Comp_2020.pdf
2. Bjørner, N., Phan, A.D., Fleckenstein, L.: νZ-an optimizing SMT solver. In: Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 9035, pp. 194–199. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_14
3. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4**(1), 1–43 (2012). https://doi.org/10.1109/TCIAIG.2012.2186810
4. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. In: Proc. of the International Conference on Software Engineering (ICSE). pp. 549–559. ACM (2018). https://doi.org/10.1145/3180155.3180248
5. Godefroid, P., Levin, M.Y., Molnar, D.A., et al.: Automated whitebox fuzz testing. In: Proc. of Network and Distributed Systems Security (NDSS). vol. 8, pp. 151–166. The Internet Society (2008)
6. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252

7. Takanen, A., Demott, J.D., Miller, C., Kettunen, A.: Fuzzing for software security testing and quality assurance. Artech House (2018)
8. Wang, F., Shoshitaishvili, Y.: Angr - the next generation of binary analysis. In: Proc. of Cybersecurity Development (SecDev). pp. 8–9. IEEE (2017). https://doi.org/10.1109/SecDev.2017.14