# A tight integration of symbolic execution and fuzzing

Yaëlle Vinçont, Sébastien Bardin, Michaël Marcozzi

# A Tight Integration of Symbolic Execution and Fuzzing (short paper)

Yaëlle Vinçont[1,2], Sébastien Bardin[2], and Michaël Marcozzi[2]

[1] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, Gif-sur-Yvette, France
yaelle.vincont@universite-paris-saclay.fr
[2] Université Paris-Saclay, CEA, List, Saclay, France
first.last@cea.fr

**Abstract.** Most bug finding tools rely on either fuzzing or symbolic execution. While they both work well in some situations, fuzzing struggles with complex conditions and symbolic execution suffers from path explosion and high constraint solving costs. In order to enjoy the advantages from both techniques, we propose a new approach called *Lightweight Symbolic Execution* (LSE) that integrates well with fuzzing. Especially, LSE does not require any call to a constraint solver and allows for quickly enumerating inputs. In this short paper, we present the basic concepts of LSE together with promising preliminary experiments.

**Keywords:** Software Testing · Symbolic Execution · Fuzzing

## 1 Introduction

**Context.** *Automatic test generation* is a major topic in software engineering and security. Currently, most test generation techniques and tools studied by researchers and applied in industry rely on some form of either *symbolic execution* [2, 9, 11] or *fuzzing* [12, 13]. Symbolic execution generates so-called *seeds* (test inputs) covering as many execution paths as possible, by analyzing each of them symbolically, in order to infer a corresponding path constraints that is then solved by an off-the-shelf solver. Fuzzing relies on massive and cheap seeds generation. While the first fuzzers were akin to blackbox random testing, *greybox (mutation-based) fuzzing* [14, 16, 18] takes the technique one step further by adding a feedback loop, where new seeds are produced by randomly mutating previous seeds deemed as interesting (e.g. covering new parts of code).

**Problem.** Symbolic execution can explore arbitrarily deep parts of the program, thanks to its powerful constraint derivation and solving machinery. Yet, it scales badly as soon as the number of paths in the program is large and the constraints are difficult to solve. On the contrary, the randomness of fuzzing enables quick and easy seed generation, independent of program size or complexity. Yet, fuzzing will usually fail to explore (in acceptable time) parts of the code protected by

complex conditions (e.g. deeply nested conditions or hard-coded "magic bytes" checks). Symbolic execution and fuzzing exhibit rather complementary strengths and weaknesses, calling for a proper integration between the two techniques.

**Goal and challenges.** *Our objective is precisely to develop a mixed test generation technique reaching a sweet spot between the power of symbolic execution and the lightness of greybox fuzzing.* More precisely, we want to build an *efficient* approach able to reason about complex code, while generating seeds much more quickly and easily than symbolic execution would.

**Related work.** Several recent works [3, 4, 10, 15, 17] follow roughly the same goal. Many of these approaches [15, 17] combine an off-the-shelf fuzzer together with an off-the-shelf symbolic executor, i.e. they do not integrate the two techniques at the *conceptual* level. We aim at introducing a correct seed generation technique which genuinely *integrates* the concepts from symbolic execution with those of fuzzing.

**Proposal.** We introduce two novel ideas to tackle this problem: *Lightweight Symbolic Execution* and *Constrained Fuzzing*. Lightweight Symbolic Execution (LSE) is a variant of Symbolic Execution where the target constraint language is restricted to an *easily-enumerable* fragment. As a consequence, deriving (correct) path constraints in this language is more complicated but seeds exercising a given path are then easy to enumerate, and do not require any SMT solver. Second, a Constrained Fuzzer operates over a seed and an easy-to-enumerate constraint in order to massively generate seeds exercising the intended path. Overall, LSE will lead the exploration past specific conditions and towards interesting parts of the code, while the constrained fuzzer will efficiently create seeds, including solutions to the constraints. This allows us to explore the program without systematically relying on symbolic analysis, and removes the need for an SMT solver to create seeds satisfying the constraints.

**Contribution.** As a summary, our contribution is three-fold:

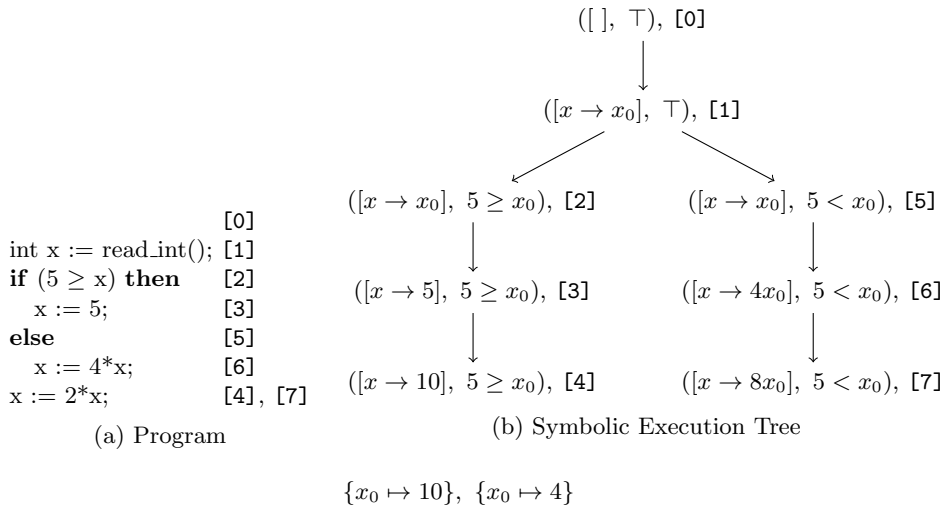
- We introduce *Lightweight Symbolic Execution* (LSE), a flavor of symbolic execution tailored for tight integration with fuzzing. LSE relies on the novel notion of *easily-enumerable* path predicates, and avoids the need for any external constraint solver;
- We show how Lightweight Symbolic Execution can be smoothly integrated with fuzzing, through the novel idea of Constrained Fuzzing, communicating through easily-enumerable path predicates, yielding fast (solver-less) seed enumeration together with targeted symbolic reasoning;
- Finally, we have implemented these ideas in an early prototype named ConFuzz, built on top of Binsec [5, 6] and AFL [18], and provide promising preliminary experiments against standard tools.

We believe that these preliminary results show the potential of LSE and Constrained Fuzzing. Still, the experimental evaluation needs to be consolidated on larger benchmarks and compared to the latest advances in fuzzing. This is left as future work.

## 2 Symbolic execution

Symbolic execution [2, 9, 11] runs the program over symbolic input instead of concrete values. Along the execution, symbolic execution maintains two pieces of information about the state of the program: a *symbolic state* $\Sigma$ – a map binding variables to their symbolic value – and a *path constraint* $\varphi$ – a predicate over the input symbols, describing the condition for a seed to reach the current instruction. On branching instructions, symbolic execution forks in order to explore all possible paths (up to a given bound). When one of the forked analyses reaches the program end, the resulting path constraint is a predicate over the input, so that executing any of its solutions follows the path of all the branching choices made in this analysis. That constraint can be tackled by an off-the-shelf solver. If the constraint has a solution, the solver will return a seed which covers the path. If there is no solution, it means that the path is unfeasible.

The execution tree on the right of Figure 1 shows the symbolic state and path predicate for each of the (numbered) instructions in the program on the left. In this tree, $x_0$ is the symbol corresponding to the program input returned by the `read_int` function, and forking happens due to the condition **if** (x >=5).




(a) Program

(b) Symbolic Execution Tree

$\{x_0 \mapsto 10\},\ \{x_0 \mapsto 4\}$

(c) Possible seeds returned by a constraint solver

**Fig. 1.** Symbolic execution of a sample program

## 3 Coverage-based fuzzing

Fuzzing [12, 13] is a brute-force software testing technique aimed at triggering faults and vulnerabilities by running the program on a very large number of

quickly-generated random seeds. In coverage-based greybox fuzzing [16, 18], the seed generation process (detailed in Figure 2) is lightly directed in order to maximize the code coverage of the produced seeds. The fuzzing tool — or fuzzer — maintains a seed database, which can be initialized by the user. The fuzzing procedure is then basically a loop, executed for as long as possible, where every new iteration selects a seed within the test database, applies a slight syntactical modification to it (a.k.a. *mutation*) and runs the program on the mutated seed. If the program run fails, the fuzzer successfully found a bug-triggering seed. If not, code coverage data is collected and analyzed. If the mutated seed covered parts of the code that had not been explored by previous iterations, it is considered to be "*interesting*" and added to the seed database. Otherwise, it is discarded. The coverage data is also used during the seed selection phase, in order to bias the picking towards seeds that recently increased coverage. The rationale behind this heuristic is that by mutating such seeds, there is a higher chance of exploring the newly uncovered parts of the program.

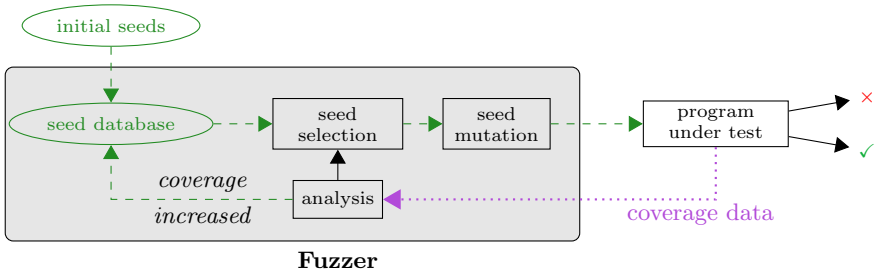



**Fig. 2.** Coverage-based fuzzing process

## 4 Lightweight symbolic execution and constrained fuzzing

We first present an example showing the potential issues faced by fuzzing and symbolic execution (Section 4.1). Then we provide an overview of our approach (Section 4.2) and finally we describe promising preliminary experimental results (Section 4.3).

### 4.1 Motivating example

We describe the issues behind fuzzing and symbolic execution and the benefits of our approach by discussing how the KLEE symbolic execution engine [1] and the AFL fuzzer [18], two popular and representative tools, struggle at generating seeds for the sample program in Figure 3, while our implementation of lightweight symbolic execution and constrained fuzzing (ConFuzz) performs well. In a nutshell, the sample program contains (lines 9-12) a loop which dramatically increases the number of paths considered by KLEE, as well as (lines

16-18) a set of nested equality conditions over the inputs, which might take AFL a long time to cover.

```
1  int main(int argc, char** argv) {
2
3    char buf[64];
4    int x, y;
5
6    read(0, buf, 64);
7
8    int cpt;
9    for (cpt = 10; cpt < 30; cpt++) {
10     if (buf[cpt] == cpt % 20)
11       y += 1;
12   }
13
14   printf("%i\n", y);
15
16   if (buf[0] == 'a')
17     if (buf[4] == 'F')
18       if (buf[7] == '6')
19         x = 1;
20       else
21         x = 2;
22     else
23       x = 3;
24   else
25     x = 4;
26
27   printf("%i\n", x);
28
29   return 0;
30 }
```

**Fig. 3.** Sample program

**AFL.** The two main issues that will prevent AFL from quickly finding seeds penetrating the three nested conditionals are the following. First, the fuzzer does not know how to mutate the seeds in order to enter the conditionals, meaning that it will typically have to try a large number of mutations before succeeding. Second, since it does not understand why a given seed increases coverage, the fuzzer may apply mutations that will destroy this ability. For example, it may mutate "a42" into "042", which does not satisfy the first condition anymore. Meanwhile, the loop does not cause any problem to AFL, as it focuses on branch rather than path coverage.

**KLEE.** For KLEE, solving the specific conditions from lines 16 to 18 is not an issue, as it will simply infer the corresponding path predicates – such as $i[0] = \texttt{'a'}$, and then create a seed using a constraint solver. On the other hand, KLEE will actively try to explore every possible path of the loop, yielding path explosion.

### 4.2 Our approach: ConFuzz

ConFuzz relies on two key components: *Lightweight Symbolic Execution* (LSE) and *Constrained Fuzzing* (CF). These two components communicate through the key notion of *easily-enumerable path predicate*. CF identifies interesting runs (like a classical fuzzer) and derives targets to be sent to LSE from these runs. LSE is in charge of deriving easily-enumerable constraints for these targets in the code. CF is then back in charge, to quickly enumerate solutions of such constraints.

**Easily-enumerable path predicates.** We want LSE to create *path predicates* in order to produce seeds reaching targets in the code. For CF to solve such predicates, we need them to be *easily-enumerable*, i.e. creating $n$ solutions is linear w.r.t. the number of inputs and $n$. For that, we restrict our constraint language for path predicates to conjunctions of interval constraints ($k \leq x \leq k'$) and equality constraints between variables ($x = y$). Then, we rely on backward domain propagation to translate actual path constraints to our language, together with concretization (forcing a symbolic variable to take an observed

concrete runtime value) for some hard-to-handle constraints, such as disequality. Figure 4 shows an example of the path predicate created by LSE ($\varphi_2$), i.e. translated from the actual predicate $\varphi_1$ to our constraint language. While not complete, $\varphi_2$ is correct: all its satisfying seeds follow the path from $\varphi_1$.

```
t = {x : 0 ; y : 1 ; z : 2 ;
     t : 4 ; v : 5}
a := x + 3;
if (a ≤ 4) then
  b := y;
  c := t;
else
  b := 2;
if (b != z) then
  d := 4;
else if (c != v)
  d := 3
```

(a) program $P$

```
declare x, y, z, t, v;
define a = x + 3;
assert (a ≤ 4);
define b = y;
define c = t;
assert (b == z);
assert (c ≠ v);
define d = 3;
```

(b) trace ($\sigma$)

$$x \leq 1 \\ \wedge\ y = z \\ \wedge\ t \neq v$$

(c) path predicate ($\varphi_1$)

$$x \leq 1 \\ \wedge\ y = z \\ \wedge\ t = 4 \\ \wedge\ v = 5$$

(d) easily-enumerable path predicate ($\varphi_2$)

**Fig. 4.** Example of an easily-enumerable path predicate

**Integrating Lightweight Symbolic Execution and Constrained Fuzzing.** Figure 5 illustrates how the two techniques communicate. In practice, communication is asynchronous, as both techniques run in parallel. When the fuzzer finds an interesting seed, it sends the trace as well as the target (a branch condition to be inverted) to LSE. LSE will analyze such information and infer constraints, which will be sent back to the fuzzer, to be associated to the seed in the database.

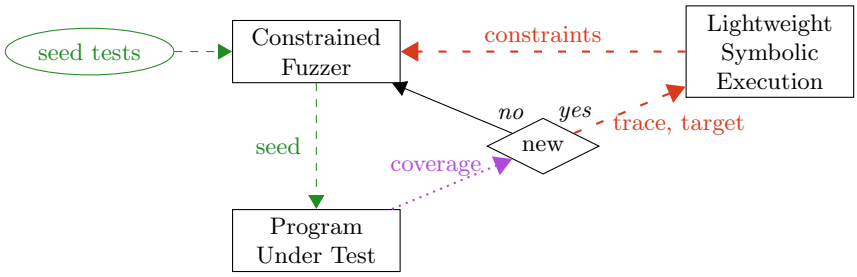



**Fig. 5.** Overview of ConFuzz

### 4.3 Preliminary experiments

**Sample program from Figure 3.** We consider two settings, depending on whether the loop can be unrolled at most 20 iterations, or at most 0 iterations. We run KLEE, AFL and AFL++[8] (a popular fork of AFL) 10 times each

over the sample program with a timeout of 20 minutes, and compare the time necessary for each tool to reach 100% branch coverage (if a tool reaches less, we count 20 minutes). We also carry the experiment with our ConFuzz prototype.

Results are presented in Table 1 and fulfill our expectations: fuzzers are not impacted by the loop but struggle on nested constraints (showing here poor performance and significant variability), while KLEE has a hard time going from 0 iteration to 20. On the other hand, we can observe that ConFuzz performs very well here (quick time for full coverage, low variability) and is not impacted by the loop. Interestingly, ConFuzz generates here roughly 6x more seed than KLEE, but only 4 of them come from a symbolic reasoning, highlighting the capacity of ConFuzz to trigger symbolic reasoning only when needed.

**Table 1.** Comparison of KLEE, AFL and ConFuzz on our sample program

| | | | AFL | AFL++ | KLEE | ConFuzz |
|---|---|---|---|---|---|---|
| **0 iterations - 20min** | **Nb success/Nb tries** | | 9/10 | 10/10 | 10/10 | 10/10 |
| | **Time (s) to cover all branches** | **Avg** | 247 | 14 | 0.3 | 1.0 |
| | | **Min** | 15 | 0.5 | 0.2 | 0.7 |
| | | **Max** | TO | 92 | 0.5 | 1.4 |
| | | **Dev** ($\sigma$) | 348 | 26 | 0.1 | 0.2 |
| **20 iterations - 20min** | **Nb success/Nb tries** | | 9/10 | 10/10 | 10/10 | 10/10 |
| | **Time (s) to cover all branches** | **Avg** | 246 | 96 | 133 | 1.4 |
| | | **Min** | 2.2 | 14 | 121 | 1.2 |
| | | **Max** | TO | 627 | 155 | 1.9 |
| | | **Dev** ($\sigma$) | 355 | 177 | 9.5 | 0.2 |

| | |
|---|---|
| AFL - average number of executions | 10,433,816 |
| AFL++ - average number of executions | 14,239,200 |
| KLEE - average number of generated seeds | 1,101,764 |
| ConFuzz - average number of executions | 6,131,172 |
| ConFuzz - average number of traces sent to LSE | 4 |

**LAVA-M.** We report the performance of ConFuzz on 3/4 programs from the standard LAVA-M fuzzing benchmark [7] (5 runs of 1h) – our prototype crashes on the last example. On `base64` (3kloc, 44 injected faults), ConFuzz reports on average 38.8 fault per run (min: 38, max: 39), while KLEE finds 10 (min:8, max: 11), AFL++ finds 0.2 (min:0, max:1) and AFL reports 0 fault. On `md5sum` (3kloc, 57 injected faults), ConFuzz reports on average 9 bugs (min: 8, max: 11), where KLEE, AFL++ and AFL do not find any bug. On `uniq` (3kloc, 28 inbjected faults), ConFuzz reports 26.9 faults on average (min:15, max:29), better than KLEE (avg: 5, min: 5, max: 5), AFL++ (avg: 0.4, min: 0 min, max: 1) and AFL (avg: 0, min: 0, max: 0).

## 5 Conclusion

We have introduced and discussed Lightweight Symbolic Execution (LSE), a variant of Symbolic Execution tailored to tight integration with fuzzing thanks to its focus on fast solution enumeration – yielding Constrained Fuzzing. We

report promising early experiments against standard tools, demonstrating the potential of these novel ideas. Future work includes consolidating the experimental evaluation with larger benchmarks and the latest advanced fuzzers as competitors, as well as providing a full formalization of the approach.

## References


1. C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
2. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.
3. P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.
4. P. Chen, J. Liu, and H. Chen. Matryoshka: fuzzing deeply nested branches. In *Conference on Computer and Communications Security*, 2019.
5. R. David, S. Bardin, T. D. Ta, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. A dynamic symbolic execution toolkit for binary-level analysis. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, SANER 2016. IEEE, 2016.
6. A. Djoudi and S. Bardin. Binsec: Binary code analysis with low-level regions. In *TACAS*. Springer, 2015.
7. B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *IEEE Symposium on Security and Privacy (SP)*, 2016.
8. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++: Combining incremental steps of fuzzing research. In *WOOT*, 2020.
9. P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.
10. H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *Security & Privacy*, 2020.
11. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
12. V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 2019.
13. B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
14. S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.
15. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.
16. Website. Libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2021.
17. I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 2018.
18. M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2021.