When You Have a Fuzzer, Everything Looks Like a Reachability Problem

Alastair F. Donaldson[0000−1111−2222−3333], Cristian Cadar[0000−0002−3599−7264], Manuel Carrasco[0000−0003−2477−2163], Dan Iorga[0000−0002−2313−7910], Dan Liew[0009−0001−7602−7707], and John Wickerson[0000−0001−6735−5533]

Imperial College London, London, UK*

Abstract. We provide an overview of three projects that explore the idea of using coverage-guided fuzzing, a technique traditionally used for finding bugs in software, in unconventional domains: (1) efficiently solving SMT formulas that use floating-point constraints; (2) achieving fast SMT sampling for such formulas; and (3) simulating operational memory models. In each case, the idea is to reduce the problem at hand into a *reachability problem*: transforming a problem instance into a program equipped with a special error location, such that finding an input that reaches the error location equates to finding a solution to the problem instance. Coverage-guided fuzzing, which excels at mutating a corpus of inputs to achieve increasing statement coverage of a system under test, can then be used to search for an input that reaches the error location—i.e., for a solution to the problem instance. We hope this overview will inspire other researchers to consider recasting search problems into a reachability problem form where coverage-guided fuzzing may prove effective.

Keywords: Coverage-guided fuzzing, constraint solving, floating point, memory models

1 Introduction

Coverage-guided mutation-based fuzzing is a randomised testing technique for automatically finding software bugs, and has been widely adopted through tools such as AFL [36], AFL++ [14] and libFuzzer [25]. Building on the basic idea of fuzzing—testing a software system on randomised inputs [28]—coverage-guided mutation-based fuzzing is a search-based test case generation technique [3], using ideas from evolutionary algorithms [18] to guide the randomised testing process.

The technique starts with a corpus of *seed* inputs: inputs that already exercise the software under test (SUT) to some extent. Further inputs are then obtained by mutating and combining existing inputs drawn from the corpus. This is what makes the technique "mutation-based", and the hypothesis behind mutation-based fuzzing is that

* This invited paper is associated with Alastair Donaldson's invited talk at RP 2025. The other authors are the main authors of the three existing papers on which this overview paper is based [8, 21, 26] and are listed alphabetically. Imperial College London was the affiliation of all authors when they contributed to these papers.

inputs obtained via mutation are more likely to further exercise the SUT compared with inputs generated from scratch in a naïve manner.

To decide whether a mutated input is interesting enough to be added to the corpus (so that it will be considered for further mutation), a check is made to see whether the input covers code in the SUT that is not covered by any existing input in the corpus. Intuitively, favouring inputs that reach new parts of the SUT is a good strategy for guiding the fuzzing process towards finding bugs. This use of coverage information is what makes the technique "coverage-guided". In the context of evolutionary algorithms, code coverage is used as a measure of fitness.

The *raison d'être* of coverage-guided mutation-based fuzzing (henceforth referred to as coverage-guided fuzzing for brevity) is to find bugs that cause a program to crash. The technique has been very successful in this regard: ClusterFuzz, Google's continuous fuzzing infrastructure for Chrome [17], and OSS-Fuzz [35], a deployment of ClusterFuzz targeting open-source projects, are reported to have found tens of thousands of bugs [17].

However, from a more abstract viewpoint, coverage-guiding fuzzing can be seen as a technique for solving program reachability problems: a coverage-guided fuzzer demonstrates that a program can crash by synthesising an input that *reaches* an error location. With this viewpoint it is interesting to consider applications of coverage-guided fuzzing to reachability programs that go beyond finding bugs in programs, by *dressing up* said reachability problems as bug-finding problems.

In this invited paper we survey three pieces of work from the authors that leverage coverage-guided fuzzing for other kinds of reachability problems:

1. JFS, where coverage-guided fuzzing is used to find solutions to SMT formulas that feature floating-point constraints [26] (Section 2).
2. JFSAMPLER, an extension to JFS concerned with *SMT sampling*—finding a diverse range of solutions to an SMT formula [8] (Section 3).
3. A project on the use of program analysis tools—including coverage-guided fuzzing—for the simulation of operational memory models, to determine whether behaviours of concurrent programs characterised by litmus tests are allowed according to a given memory model [21] (Section 4).

We with a discussion of the conditions under which coverage-guided fuzzing may be an effective reachability analysis in these application domains, and the pros and cons of other program reachability analyses (Section 5). Key related work is discussed throughout, and we refer the reader to the related work sections of the original papers on these projects for a broader discussion of relevant literature [8, 21, 26].

Relationship to existing papers. The article draws on material from the original papers about these works [8, 21, 26]. The authors of the relevant material are also authors on this article and the material is reused here with their consent.

2 Just Fuzz It: Solving Floating-Point SMT Formulas

Satisfiability modulo theories (SMT) solvers have found application in many domains including software testing and software verification (see e.g. [5, 7, 9, 15, 16, 23, 24, 31]).

```
(declare-fun a () Float64)
(declare-fun b () Float64)
(define-fun div_rne () Float64 (fp.div RNE a b))
(define-fun div_rtp () Float64 (fp.div RTP a b))
(assert (not (fp.isNaN a)))
(assert (not (fp.isNaN b)))
(assert (not (fp.isNaN div_rne)))
(assert (not (fp.isNaN div_rtp)))
(assert (not (fp.eq div_rne div_rtp)))
(check-sat)
```

Fig. 1: Example QF_FP formula

```
int FuzzOneInput(const uint8_t* data, size_t size) {
  double a = makeFloatFrom(data, size, 0, 63);
  double b = makeFloatFrom(data, size, 64, 127);
  if (!isnan(a)) {} else return 0;
  if (!isnan(b)) {} else return 0;
  double a_b_rne = div_rne(a, b);
  double a_b_rtp = div_rtp(a, b);
  if (!isnan(a_b_rne)) {} else return 0;
  if (!isnan(a_b_rtp)) {} else return 0;
  if (a_b_rne != a_b_rtp) {} else return 0;
  abort(); // TARGET REACHED
}
```

Fig. 2: C++ program generated by JFS for Figure 1 using the *fail-fast* encoding

For example, symbolic execution techniques involve gathering the constraints on a program input that must hold for the program to reach a given location or trigger a particular error, and then using an SMT solver to solve for inputs that satisfy these constraints [7, 15, 16, 31].

A limitation to the utility of SMT solvers in these domains can be a lack of scalability; e.g. solver timeouts cause symbolic executors to grind to a halt and lead to inclusive results from program verifiers. Scalability is a particular problem when reasoning about formulas that use the floating-point SMT theory (QF_FP) [33] or the combination of floating-point and bitvector theories (QF_BVFP), to the extent that it can be impractical to apply SMT-based analysis methods to numeric applications that operate on floating-point numbers [27].

The limited scalability of floating-point SMT solvers inspired the first project that we survey here: the Just Fuzz It Solver (JFS).

Overview of JFS. The idea behind JFS is to transform the problem of finding a satisfying assignment to an SMT formula into a program reachability problem. Specifically, JFS transforms an SMT formula into a program such that (1) a program input corresponds to an assignment to the free variables of the formula, and (2) the program contains a special *target* statement that is reachable if and only if the input corresponds to a satisfying assignment to the formula.

A coverage-guided fuzzer aims to find inputs that maximise coverage, so when applied to this program it will search relentlessly for an input that reaches the target statement, i.e. for a satisfying assignment to the formula. The hypothesis behind JFS

is that this technique may sometimes be able to rapidly find satisfying assignments for formulas that are challenging for general-purpose solvers, such as floating-point formulas. However, JFS is incomplete: for an unsatisfiable formula the target statement is unreachable, thus coverage-guided fuzzing will run indefinitely, never finding an error-triggering input. We envision that JFS would be run in parallel with a complete solver as part of a portfolio.

JFS requires that the input formula is presented as a conjunction of assertions. Given the conjunction, JFS generates a C++ program that takes an assignment to the free variables of the formula as input. The program evaluates the formula on the assignment by evaluating the top-level conjuncts in turn. By construction, the program crashes if and only if all of the conjuncts are satisfied—i.e. if the input is a satisfying assignment. JFS then uses libFuzzer to automatically search for an input that triggers a crash—i.e. for a satisfying assignment.

Example. As an illustration of this idea, consider the example constraints in Listing 1, shown in SMT-LIB format. Free variables `a` and `b` of type `Float64` are declared on lines 1 and 2 respectively. On lines 3 and 4, variables `div_rne` and `div_rtp` are defined to be the division of `a` by `b` using the rounding to nearest, ties to even (RNE) and rounding toward positive infinity (RTP) rounding modes, respectively. The satisfiability problem captured by the example is the conjunction of the constraints specified in the five `assert` statements. The first four constraints state that none of `a`, `b`, `div_rne` and `div_rtp` are `NaN`; the last states that `div_rne` is not equal to `div_rtp`.

A possible translation of these constraints into a C++ program is shown in Listing 2. The program is a *fuzz target* for libFuzzer (with some details omitted for brevity). The guard of each `if` statement corresponds to a constraint. The fuzzer will repeatedly call `FuzzOneInput` (line 1), each time passing an input of `size` bytes via the `data` buffer. If the `abort()` statement is reached (line 11), causing the program to crash, the input corresponds to a satisfying assignment and JFS terminates and returns SAT. Otherwise, the fuzzer proceeds to try another input.

Further details and results. Before transforming a formula into a program, JFS performs a number of simplification and rewriting passes to make the formula more amenable to coverage-guided fuzzing. One such rewrite step involves splitting a top-level `and` constraint (appearing directly under **`assert`**) into two separate constraints (via two distinct **`assert`** commands). This leads to the resulting C++ program having a distinct location associated with the satisfaction of each conjunct, so that the coverage-guided fuzzer is rewarded separately for finding inputs that satisfy each conjunct.

We experimented with two different program encodings: *fail-fast*, where the fuzz target exits as soon as it is found that the current input does *not* satisfy some constraint of the formula, illustrated by Figure 2, and *try-all*, which evaluates all constraints of the formula even if some constraints are found not to hold, illustrated by Figure 3. We hypothesised that *try-all* would provide a stronger coverage signal that might outweigh the additional cost associated with redundantly evaluating further constraints once a constraint has been found not to hold. However, in practice we found that *fail-fast* was significantly more efficient.

Since coverage-guided fuzzing needs an input corpus, JFS provides *smart seeds* featuring special constant values such as infinities, zeros and NaNs [26].

```
1  int FuzzerTestOneInput(const uint8_t* data, size_t size) {
2    double a = makeFloatFrom(data, size, 0, 63);
3    double b = makeFloatFrom(data, size, 64, 127);
4    size_t counter = 0;
5    if (!isnan(a)) ++counter;
6    if (!isnan(b)) ++counter;
7    double a_b_rne = div_rne(a, b);
8    double a_b_rtp = div_rtp(a, b);
9    if (a_b_rne != a_b_rtp) ++counter;
10   if (!isnan(a_b_rne)) ++counter;
11   if (!isnan(a_b_rtp)) ++counter;
12   if (counter != 5)
13     return 0;
14   abort(); // TARGET REACHED
15 }
```

Fig. 3: C++ program generated by JFS for Figure 1 using the *try-all* encoding

An evaluation over benchmarks drawn SMT-COMP suites with respect to state-of-the-art solvers at the time of the project showed JFS to be highly competitive on QF_FP and QF_BVFP benchmarks (formulas that use the floating-point theory or the combination of floating-point and bitvector theories, respectively), but uncompetitive on QF_BV benchmarks (which only use the bitvector theory). The results support the idea that JFS could be used to help unblock the search for satisfying assignments in the presence of floating-point constraints. The evaluation also confirmed that coverage guidance is an important contributor to the success of JFS: results for a version of the tool where coverage guidance is disabled show markedly worse results.

JFS only supports the combination of bitvector and floating-point theories, but the idea of encoding SMT solving as a program reachability problem to be solved using fuzzing should be straightforward to adapt to other finite-domain SMT theories.

3 JFSampler: Efficient Floating-Point SMT Sampling

Traditional SMT solvers aim to find a single satisfying assignment for a satisfiable formula, but in some application domains it can be useful to obtain a diverse range of satisfying assignments for a formula; e.g. in software testing there is evidence that symbolic execution can benefit from retrieving multiple solutions from the constraints instead of exploring multiple paths from a certain program point [19].

In large, configurable systems, such as operating systems or web development frameworks, it is often useful to generate a small but representative set of valid build or deployment configurations for testing [30]. In hardware design, it is often useful to generate multiple stimuli that meet the preconditions of a functional specification, and then compare the resulting outputs with those of the hardware's logic design before the design becomes silicon [13, 29].

In response to this, recent work has focused on SAT and SMT sampling techniques that find large sets of satisfying assignments for a formula, attempting to provide reasonable coverage of the formula's solution space [11–13].

The difficulty of scaling SMT solvers in the floating point domain (see Section 2) means that SMT sampling techniques such as SMTSampler [12], which rely on using

an existing SMT solver as a source of initial satisfying assignments, are also limited in this domain. In response to this we designed JFSAMPLER, an SMT sampling technique based on JFS geared towards efficient sampling for floating-point formulas [8].

Overview of JFSAMPLER. The basic idea behind JFSAMPLER is to take the JFS approach of encoding the search for a satisfying assignment as the problem of finding an input that reaches a special *target* statement in a program. However, instead of using coverage-guided fuzzing to search for a *single* input that reaches the target location, JFSAMPLER runs fuzzing continuously for a given time budget, collecting all distinct inputs that reach the target, which correspond to distinct satisfying assignments.

However, the basic idea of reusing JFS for this task, without modification, suffers from the problem that once a solution-inducing input has been found, further solution-inducing inputs will exercise the same path through the program: the unique path that satisfies all conjuncts and hence reaches the target location. With this setup the fuzzer is not rewarded (via coverage feedback) for finding diverse solutions.

To overcome this, JFSAMPLER uses a more sophisticated encoding based on an SMT-level coverage metric proposed in the SMTSAMPLER project [12]. The new encoding, which we call the *diversity encoding*, features additional code that, if covered, will correlate with an increase in the SMT-level coverage metric used for measuring diversity.

Example. Recall again the SMT formula of Figure 1 and its associated reachability problem encoding of Figure 2. The diversity encoding employed by JFSAMPLER involves the addition of extra conditional code right before the `abort()` call that corresponds to the target location. At this point, an input corresponding to a satisfying assignment has been found. The additional code is designed to test the value of every bit in every non-root and non-leaf subexpression of the formula under the current satisfying assignment.

For our running example this would involve adding 128 conditional statements between lines 10 and 11 of Figure 2 that test each of the 64 bits of each of the **`double`** variables `a_b_rne` and `a_b_rtp`. Each such test introduces a new program point that the fuzzer will be rewarded for reaching. This rewards the fuzzer for synthesising assignments that make the formula true in myriad different ways. It also allows the fuzzer to distinguish between them and keep them in the corpus for further refinement.

Further details and results. The SMTSAMPLER project proposed an effective method that takes existing satisfying assignments for a formula and combines them using a heuristic that generates candidate follow-on satisfying assignments. In JFSAMPLER we used this heuristic as the basis for a *custom mutator* for libFuzzer, the underlying fuzzer used by JFS. Our custom mutator has access to all satisfying assignments that have been shown so far. On encountering an input that achieves new coverage, the custom mutator combines the input with two randomly-selected previous satisfying assignments (if available), using a combination strategy based on the technique used by SMTSAMPLER.

We evaluated JFSAMPLER empirically, comparing it with SMTSAMPLER (which is based on the Z3 solver and its support for MaxSMT problems [6]), over the QF_FP and QF_BVFP formulas that were used in the evaluation of JFS. We found that JFSAMPLER significantly outperformed SMTSAMPLER on the QF_FP benchmarks. An ablation study confirmed that the diversity encoding and custom mutator make a substantial contribution to the performance of JFSAMPLER on these benchmarks when compared to

a naïve version of JFSAMPLER that simply searches for multiple crashing inputs with respect to the program encoding emitted by JFS with no further improvements.

On the QF_BVFP benchmarks we found that SMTSAMPLER significantly outperformed the naive version of JFSAMPLER. The diversity encoding and custom mutator, when used individually in isolation, improved the performance of JFSAMPLER, but performance remained below that of SMTSAMPLER. However, these two features in combination ultimately led to a slight performance improvement over SMTSAMPLER.

4 Simulating Concurrency Memory Models

Our final case study considers the use of coverage-guided fuzzing and other reachability analysis techniques to aid in the simulation of *operational memory models* [21]. This project is distinct from the JFS and JFSAMPLER projects described in Sections 2 and 3, but was in part inspired by the success of JFS.

Overview. The memory model of a shared memory concurrent system formally describes the potential interactions between threads that can arise through communication via shared memory locations. For reasons of efficiency, modern multi-core CPUs, accelerators and heterogeneous systems feature memory models that are *weaker* than the appealingly simple *sequentially consistent* memory model [22], as demonstrated by vendor documentation and various academic studies [1, 2, 20, 32, 34].

An operational memory model characterises the allowed memory behaviours of a system using a state machine, where *states* represent components such as store buffers, caches and queues, and *transitions* define the legal changes between these states, triggered by memory operations such as reads, writes and flushes. The memory model can then be applied to *litmus tests*—small programs that capture potential interactions between threads, where a litmus test is *allowed* if the behaviour that it captures is permitted by the memory model.

Constructing operational memory models and applying them to litmus tests is facilitated by *simulators* that reveal which behaviours of a given program are allowed. While extensive work has been done on simulating *axiomatic* memory models [2, 4], there has been less work on simulation of operational models [32, 34], despite the fact that operational models are arguably more intuitive than their axiomatic counterparts.

Part of the reason for this is the overhead associated with engineering and maintaining a full-blown simulator for an operational memory model. For example the RMEM [32] state-of-the-art memory model simulator, which supports the ARM, Power, RISC-V and x86 memory models, comprises more than 60k lines of OCaml code in part because the developers had to build custom efficient reachability analyses.

An appealing idea to reduce this engineering overhead is to implement the logic of the memory model as a program that takes a particular test scenario as input. Determining whether the test scenario is allowed would then boil down to determining whether a particular state of the program that encodes the memory model is reachable when executed on an input describing the scenario of interest, and off-the-shelf reachability analysis tools for the language of interest could be leveraged to answer this question. Subsequent detailed examination of traces would then be possible by stepping through the simulator code using a standard debugger.

```
1  // Choose a number of steps of the simulation to run
2  int sim_steps = choose(SIMULATION_STEPS);
3  for (int i = 0; i < sim_steps; i++) {
4    // Choose a thread to take a step
5    int thread = choose(NUM_THREADS);
6    // Choose whether the CPU or the environment takes a step
7    Action action = choose(NUM_ACTIONS);
8    switch (action) {
9      case CPU_THREAD: // the CPU is to take a step
10       // Check that the thread still has work to do
11       if (!thread_ops[thread].empty()) {
12         // Pop the next instruction from the thread's list
13         Operation op = thread_ops[thread].pop();
14         // Carry out a write by appending to the CPU's store buffer
15         if (op.type == WRITE) {
16           write_to_buffer(thread, op.var, op.val);
17         }
18         // Carry out a read from the CPU's store buffer or from memory
19         if (op.type == READ) {
20           read_buffer_or_memory(thread, op.var);
21         }
22         break;
23       }
24     case FLUSH_BUFFER: // the environment is to take a step
25       // Check that the CPU's store buffer is not empty
26       if (!buffer[thread].empty()) {
27         // Flush an entry from the CPU's store buffer into memory
28         flush_buffer(thread);
29         break;
30       }
31   }
32 }
33 // Check whether the litmus test's postcondition has been reached
34 check_litmus_test();
```

Fig. 4: The pseudocode of the mechanised x86 memory model.

In recent work [21] we put this idea into practice for two different operational memory models: the x86 memory model, which allows for a comparison with the RMEM simulator, and the X+F memory model [20] associated with a system that combines an Intel Xeon CPU with a field-programmable gate array (FPGA), for which no bespoke memory model simulator exists.

Example. Figure 4 gives a flavour of our encoding—full details are in our full article about the project [21]. The program makes several nondeterministic choices – how many simulation steps to run (line 2), which thread to activate for each step (line 5), and whether each step corresponds to the thread's next instruction being executed (line 9) or to the 'environment' making a transition by flushing an x86 store buffer (line 24). If executed directly, this program would not be very useful because it would be unlikely to make the right sequence of decisions to get interesting behaviours. However, it becomes useful when presented for reachability analysis, because then the question becomes: is it *possible* to resolve all of these choices so as to make the interesting behaviour emerge.

Further details and results. We investigated the effectiveness of three different C-analysis tools for analysis of: CBMC [10], which encodes the reachability problem as a monolithic SAT query, KLEE [7], which uses dynamic symbolic execution to explore

the program in a path-by-path manner, generating an SMT query per path, and using three different coverage-guided fuzzers including libFuzzer (the fuzzer behind JFS and JFSAMPLER). A common feature of all these tools is that they avoid false positives: if the tool reports 'reachable' then the error-state really is reachable. This is because, unlike many static analysis tools, they do not employ any abstraction.

The CBMC and KLEE tools can, in principle, prove that error states are *not* reachable, assuming the litmus test provided as input is loop-free. In the case of CBMC this is via the use of *unwinding assertions*, while for KLEE it involves exhaustive exploration of all program paths. While the fuzzers cannot prove unreachability of error states, when the error states *are* reachable, the fuzzers tend to discover this much more quickly than the other two tools. However, large litmus tests feature error states that can only be reached via lengthy paths that depend on an intricate schedule of thread and memory subsystem events. The fuzzers struggled on these larger litmus tests, while CBMC/KLEE performed somewhat better due to their more systematic approaches.

Our results also show that the *coverage-guidedness* of the fuzzers is valuable: when coverage guidance is disabled, the fuzzers did not perform at all well for the task of memory model simulation.

The X+F memory model to which we applied this technique is rather complicated, so it is a testimony to the generality of our approach that we were able to obtain a simulator for it with little additional effort, by encoding its logic as a C program.

5 Discussion

We conclude with a discussion of the strengths and weaknesses of coverage-guided fuzzing as a reachability analysis technique in the context of these three case studies, and how the dynamic vs. symbolic and under-approximating vs. over-approximating dimensions of a reachability analysis affect its suitability in these domains.

Effectiveness of coverage-guided fuzzing. Our experience with JFS is that a fuzzing-based approach to constraint solving often works better than traditional approaches for formulas involving floating-point constraints, but that the performance of JFS was poor when applied to formulas involving only bitvector constraints, where traditional methods excel. When applied to the problem of memory model simulation, fuzzing excelled in comparison to symbolic analysis techniques for simpler litmus tests, but fared less well on larger tests where the behaviour under consideration relied on a very specific interleaving of threads and memory subsystem events. Our hypothesis is that fuzzing has the potential to outperform symbolic techniques for reachability analysis on problem instances where (a) many solutions exist, so that the probability of finding a solution via guided random search is reasonably high, and (b) the constraints that a solution must satisfy are nevertheless complex enough that symbolic solving algorithms have a difficult time navigating the underlying search tree. Investigating this hypothesis in more detail, e.g. by using model counting to see whether there is a correlation between the number of solutions to a floating-point SMT query and the ease with which JFS can solve this query, would be an interesting avenue for future work.

In both the JFS work and the memory model simulation project, experimental results confirmed that the "coverage-guided" part of coverage-guided fuzzing is an important

contributor: disabling coverage guidance led to less effective solving of floating-point formulas by JFS (and thus would also negatively impact JFSAMPLER), and to markedly worse results for memory model analysis.

The JFSAMPLER approach is a particularly good match for fuzzing because in SMT sampling one is always concerned with satisfiable formulas, and typically formulas that have many different solutions.

Dynamic vs. symbolic reachability analyses. In the JFS and JFSAMPLER projects we focused on a purely-dynamic reachability analysis—coverage-guided fuzzing—to analyse the program associated with an SMT formula, while in the memory model simulation work we also considered the use of techniques that perform full or partial symbolic reasoning (CBMC and KLEE).

In principle, any under-approximating analysis could be used to find satisfying assignments through analysis of the program emitted by JFS, and it would certainly be possible to try applying CBMC or KLEE to a JFS-generated program. However, given that the purpose of JFS is to provide an alternative means for solving a problem that is known to be hard for symbolic methods (namely reasoning about floating-point arithmetic), applying SAT or SMT-based analysis techniques such as CBMC or KLEE to the programs generated by JFS would make little sense. In contrast, memory model simulation involves the exploration of the possible behaviours of a nondeterministic system, something at which symbolic program analysis tools excel.

Under-approximating vs. over-approximating analyses. In the projects discussed here, we have focused almost entirely on the use of under-approximating program analyses, geared towards bug finding, except that in the memory model simulation work we considered the use of CBMC and KLEE for "brute force" verification, by fully unrolling program loops (CBMC) or exhaustively exploring program paths (KLEE).

In the context of JFS it is necessary to use an under-approximating analysis to find solutions to a formula from its associated program with confidence. Furthermore, the under-approximating analysis must yield inputs that triggers the bug found by the analysis if one wishes to obtain a satisfying assignment to the formula of interest rather than merely knowing that it is satisfiable. An over-approximating analysis *might* still be useful for finding solutions to a formula if there is a way to obtain a candidate input from an alarm raised by the analysis, because one could simply run the program on the candidate input to check whether it indeed reaches the program's target location.

In principle it would be possible to use a JFS-generated program to prove unsatisfiability of a given formula by using an over-approximating static verification technique to prove that the program is correct. However, similar to the discussion above regarding the limited value of applying CBMC or KLEE to JFS-generated programs, many static verification techniques are based on symbolic analysis so using them for this task would again seem somewhat circular.

For JFSAMPLER, bug finding is the *only* meaningful way to analyse the program associated with a formula, because SMT sampling involves mining a formula for a diverse range of satisfying assignments and is not concerned with proving unsatisfiability.

In the context of memory model analysis, applying an under-approximating analysis to a program obtained from a (memory model, litmus test) pair facilitates establishing that the behaviour characterised by a litmus test is *allowed*: finding a bug in the resulting

program equates to confirming that a behaviour is possible. A bug report from an over-approximating analysis would merely indicate that the behaviour *might* be allowed, because the bug report could be a false alarm. In contrast, showing that a memory model behaviour is *disallowed* would require verifying that the associated program is correct. We have only investigated performing such verification via brute force methods, as discussed above, and it would be interesting to assess the utility of other verification techniques for this task.

References

1. Alglave, J., Batty, M., Donaldson, A.F., Gopalakrishnan, G., Ketema, J., Poetzl, D., Sorensen, T., Wickerson, J.: GPU concurrency: Weak behaviours and programming assumptions. ASPLOS'15, ACM (2015). https://doi.org/10.1145/2694344.2694391
2. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 1–74 (2014). https://doi.org/10.1145/2627752
3. Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.: A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Software Eng. **36**(6), 742–762 (2010). https://doi.org/10.1109/TSE.2009.52
4. Armstrong, A., Campbell, B., Simner, B., Pulte, C., Sewell, P.: Isla: integrating full-scale ISA semantics and axiomatic concurrency models (extended version). Formal Methods Syst. Des. **63**(1), 110–133 (2024). https://doi.org/10.1007/S10703-023-00409-Y
5. Ball, T., Bounimova, E., Levin, V., Kumar, R., Lichtenberg, J.: The static driver verifier research platform. CAV'10, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_11
6. Bjørner, N., Phan, A.D., Fleckenstein, L.: νz - an optimizing SMT solver. TACAS'15, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_14
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, USENIX (2008)
8. Carrasco, M., Cadar, C., Donaldson, A.F.: Scalable SMT sampling for floating-point formulas via coverage-guided fuzzing. ICST'25, IEEE (2025). https://doi.org/10.1109/ICST62969.2025.10989031
9. Carter, M., He, S., Whitaker, J., Rakamaric, Z., Emmi, M.: SMACK software verification toolchain. In: Dillon, L.K., Visser, W., Williams, L.A. (eds.) ICSE'16 Companion Volume, ACM (2016). https://doi.org/10.1145/2889160.2889163
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. TACAS'04, Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
11. Delannoy, R., Meel, K.S.: On almost-uniform generation of SAT solutions: The power of 3-wise independent hashing. LICS'22, ACM (2022). https://doi.org/10.1145/3531130.3533338
12. Dutra, R., Bachrach, J., Sen, K.: SMTSampler: Efficient stimulus generation from complex SMT constraints. ICCAD'18, ACM (2018). https://doi.org/10.1145/3240765.3240848
13. Dutra, R., Laeufer, K., Bachrach, J., Sen, K.: Efficient sampling of SAT solutions for testing. ICSE'18, ACM (2018). https://doi.org/10.1145/3180155.3180248
14. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++: Combining incremental steps of fuzzing research. WOOT'20, USENIX (2020)
15. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. PLDI'05, ACM (2005). https://doi.org/10.1145/1065010.1065036
16. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. NDSS'08, The Internet Society (2008)
17. Google: ClusterFuzz, https://github.com/google/clusterfuzz (2025)

18. Holland, J.: Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. University of Michigan Press (1975)
19. Huang, H., Yao, P., Wu, R., Shi, Q., Zhang, C.: Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. S&P'20, IEEE (2020). https://doi.org/10.1109/SP40000.2020.00063
20. Iorga, D., Donaldson, A.F., Sorensen, T., Wickerson, J.: The semantics of shared memory in Intel CPU/FPGA systems. Proc. ACM Program. Lang. **5**(OOPSLA), 1–28 (2021). https://doi.org/10.1145/3485497
21. Iorga, D., Wickerson, J., Donaldson, A.F.: Simulating operational memory models using off-the-shelf program analysis tools. IEEE Trans. Software Eng. **49**(12), 5084–5102 (2023). https://doi.org/10.1109/TSE.2023.3326056
22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **28**(9), 690–691 (1979). https://doi.org/10.1109/TC.1979.1675439
23. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. LPAR'10, Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
24. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. TACAS'10, Springer (2010). https://doi.org/10.1007/978-3-642-12002-2_26
25. LibFuzzer website. http://llvm.org/docs/LibFuzzer.html (2025).
26. Liew, D., Cadar, C., Donaldson, A., Stinnett, J.R.: Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing. ESEC/FSE'19, ACM (2019). https://doi.org/10.1145/3338906.3338921
27. Liew, D., Schemmel, D., Cadar, C., Donaldson, A., Zähl, R., Wehrle, K.: Floating-point symbolic execution: A case study in N-version programming. ASE'17, IEEE (2017). https://doi.org/10.1109/ASE.2017.8115670
28. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Communications of the Association for Computing Machinery (CACM) **33**(12), 32–44 (1990). https://doi.org/10.1145/96267.96279
29. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Mag. **28**(3), 13–30 (2007). https://doi.org/10.1609/AIMAG.V28I3.2052
30. Plazar, Q., Acher, M., Perrouin, G., Devroey, X., Cordy, M.: Uniform sampling of SAT solutions for configurable systems: Are we there yet? ICST'19, IEEE (2019). https://doi.org/10.1109/ICST.2019.00032
31. Poeplau, S., Francillon, A.: Symbolic execution with SymCC: Don't interpret, compile! USENIX Security'20, USENIX (2020)
32. Pulte, C., Flur, S., Deacon, W., French, J., Sarkar, S., Sewell, P.: Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. Proc. ACM Program. Lang. **2**(POPL), 19:1–19:29 (2018). https://doi.org/10.1145/3158107
33. Rümmer, P., Wahl, T.: An SMT-LIB theory of binary floating-point arithmetic. SMT'10 (2010). http://www.cprover.org/SMT-LIB-Float/smt-fpa.pdf
34. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding POWER multiprocessors. PLDI'11, ACM (2011). https://doi.org/10.1145/1993498.1993520
35. Serebryany, K.: OSS-Fuzz – Google's continuous fuzzing service for open source software. Invited talk at USENIX Security'17, USENIX (2017).
36. Zalewski, M.: Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt (2025).