Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing

Daniel Liew
dan@su-root.co.uk
Imperial College London
United Kingdom

Cristian Cadar
c.cadar@imperial.ac.uk
Imperial College London
United Kingdom

Alastair F. Donaldson
afd@imperial.ac.uk
Imperial College London
United Kingdom

J. Ryan Stinnett
jryans@gmail.com
Mozilla
United States

ABSTRACT

We investigate the use of coverage-guided fuzzing as a means of proving satisfiability of SMT formulas over finite variable domains, with specific application to floating-point constraints. We show how an SMT formula can be encoded as a program containing a location that is reachable if and only if the program's input corresponds to a satisfying assignment to the formula. A coverage-guided fuzzer can then be used to search for an input that reaches the location, yielding a satisfying assignment. We have implemented this idea in a tool, **J**ust **F**uzz-it **S**olver (JFS), and we present a large experimental evaluation showing that JFS is both competitive with and complementary to state-of-the-art SMT solvers with respect to solving floating-point constraints, and that the coverage-guided approach of JFS provides significant benefit over naive fuzzing in the floating-point domain. Applied in a portfolio manner, the JFS approach thus has the potential to complement traditional SMT solvers for program analysis tasks that involve reasoning about floating-point constraints.

CCS CONCEPTS

• **Theory of computation → Constraint and logic programming**; • **Software and its engineering → Software testing and debugging**.

KEYWORDS

Constraint solving, feedback-directed fuzzing

ACM Reference Format:
Daniel Liew, Cristian Cadar, Alastair F. Donaldson, and J. Ryan Stinnett. 2019. Just Fuzz It: Solving Floating-Point Constraints using Coverage-Guided Fuzzing. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338921

1 INTRODUCTION

Satisfiability modulo theories (SMT) *solvers* have made tremendous progress over the last decade [25] and now underpin many important software engineering tools, including symbolic execution engines (e.g. [14, 28, 29, 50, 57]), program verifiers (e.g. [18, 39]) and program synthesis frameworks (e.g. [31, 32]).

Despite these advances, SMT solvers often exhibit limited scalability on large problems [48], and solving can be challenging for certain underlying theories. Scalable SMT solving in the theory of floating-point arithmetic is a particular challenge, and the subject of a lot of recent and ongoing work [3, 11, 35, 42].

An unrelated technology, *coverage-guided mutation-based fuzzing*, is widely used to automatically find inputs to a system under test (SUT) that expose crashes and potentially exploitable undefined behaviours [40, 45]. For an SUT that has been instrumented to record coverage information, a coverage-guided mutation-based fuzzer takes an initial corpus of inputs and uses genetic algorithms to synthesise further inputs by mutating and combining elements of the corpus. Inputs that cover new parts of the SUT are added to the corpus, under the hypothesis that via further manipulation they may yield inputs that provide even more coverage, and that aiming for high coverage is a good strategy for triggering bugs.

In this paper, we present an in-depth investigation into the potential for coverage-guided mutation-based fuzzing to be used to solve SMT formulas. Our idea is to transform an SMT formula into a *program* whose input corresponds to an assignment to the free variables of the formula, containing a statement, *target*, that is reachable if and only if the input corresponds to a satisfying assignment. 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 *target*, i.e. for a satisfying assignment to the formula. Our hypothesis is that this technique may sometimes be able to rapidly find satisfying assignments for formulas that are challenging for general-purpose solvers. The method we propose does *not* intend to help in proving *unsatisfiability* of formulas.

We present JFS (**J**ust **F**uzz it **S**olver), a prototype constraint solver based on coverage-guided mutation-based fuzzing. JFS is *sound*: a SAT result can be trusted. However, it is *incomplete*: JFS could time out and, as discussed above, unsatisfiability cannot be proven. We envision JFS would be run in parallel with a complete solver to form a portfolio solver. JFS was inspired by the limited scalability we have observed for state-of-the-art SMT solvers with respect to floating-point constraints, and currently supports the combination of boolean, bitvector, and floating-point theories, but our idea of SMT solving via coverage-guided fuzzing should be straightforward to adapt to any SMT theory over finite-domain variables.

We present a large experimental evaluation comparing JFS with seven floating-point-capable SMT solvers, over a set of 1344 benchmarks from three different SMT-COMP [58] suites. Our evaluation aims to answer the following research questions:

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338921

RQ1 To what extent is coverage-guided mutation-based fuzzing superior to naive random input generation for SMT solving?

RQ2 To what extent can JFS be accelerated via "smart seeds" derived from the formula under analysis and/or the associated SMT theory?

RQ3 How does the execution time of JFS compare with state-of-the-art SMT solvers when applied to satisfiable formulas over Boolean, bitvector, and floating-point variables?

Our main finding is that JFS is competitive with state-of-the-art solvers such as MathSAT5 and Z3 on floating-point constraints, complementing these solvers both in terms of number of solved benchmarks and execution time. By contrast, it is uncompetitive on bitvector-only constraints. In terms of design features, we found coverage-guided mutation-based fuzzing superior to naive random input generation, and the use of smart seeds to be beneficial.

We qualify the relative success of JFS with respect to floating-point constraints by acknowledging that solver support for floating point, as well as available evaluation benchmarks, are relatively recent, while traditional solvers are very mature for bitvector-only constraints, and the benchmarks available for this theory are known to be challenging.

In summary, our main contributions are:

1. The idea of leveraging coverage-guided mutation-based fuzzing to find satisfying assignments to SMT formulas (illustrated concretely via a worked example in §3);
2. JFS, a sound, incomplete solver for floating-point and bitvector constraints based on this idea (§4);
3. An evaluation comparing JFS with seven floating-point-capable SMT solvers over 1344 SMT-COMP benchmarks, addressing the above research questions (§5).

After covering relevant background (§2), we give an overview of JFS (§3), discuss the design and implementation of the approach and tool (§4), and present a detailed experimental evaluation (§5). We then discuss related work (§6), and ideas for future research directions (§7). Throughout the paper we discuss the limitations of JFS and threats to the validity of our approach.

2 BACKGROUND

We provide relevant background on coverage-guided fuzzing (§2.1) and some brief notes on floating-point arithmetic (§2.2).

2.1 Coverage-Guided Mutation-Based Fuzzing

Mutation-based fuzzing starts with a set of existing *seed* inputs, known to already exercise the SUT in some depth, and generates further inputs by mutating and combining seeds. Intuitively, the resulting inputs are much more likely to exercise the SUT in interesting ways compared with inputs generated in a purely random fashion. If code coverage data for the SUT can be obtained, through compile-time or binary instrumentation, a fuzzer can operate in a *coverage-guided* manner. Code covered by an input can be used as a proxy for measuring how interesting that input is, with an input that covers new code being deemed interesting.

Coverage-guided mutation-based fuzzing combines these ideas: starting from an initial corpus, SUT inputs are generated via mutation. An input that covers new code is added to the corpus to be considered as a seed for future mutation. Typical mutations include making small changes to an input in isolation, and performing "crossover", where multiple inputs are combined into one. This approach is essentially an *evolutionary algorithm* [33] where an input is considered *fit* if it covers new code. An evolutionary algorithm used in this context is part of a broader research area known as *search-based test case generation* [2].

Two popular coverage-guided mutation-based fuzzers, AFL [45] and LibFuzzer [40] (on which JFS is based) have found numerous bugs in real-world software [41, 44].

2.2 Floating-Point Arithmetic

A motivating use case for our work is SMT formulas that contain constraints over floating-point variables. We recap here a few terms and concepts that will be used later on.

Single- and double-precision floating-point numbers are represented in SMT-LIB [8] by the `Float32`/`Float64` types, which correspond to the IEEE-754 binary32/binary64 types [34]. The semantics of most floating-point operations match the process of performing the operation with real number semantics then rounding the result to a nearby floating-point number. Several rounding modes can be used, including rounding to the nearest floating-point number with ties favouring an even binary representation (RNE), and rounding towards positive infinity (RTP). The set of floating-point bit patterns includes special patterns to represent infinities, as well as "not a number" (NaN), which handles the results of computations for which no numerical representation makes sense (such as 0/0).

3 OVERVIEW OF JFS

In brief, JFS uses the following method to find a satisfying assignment to a formula Q presented as a conjunction of constraints:[1]

A program P is constructed such that:

- P takes a sequence of variables as input, with each variable corresponding to a free variable in Q.
- P contains a sequence of *constraint branches*, one per constraint in Q, each of which is an `if` statement whose condition corresponds exactly to the associated constraint.
- P contains a *target* statement that returns 1 if and only if *all* the `true` branches of the constraint branches are traversed.

JFS then passes the program P to a coverage-guided mutation-based fuzzer, which repeatedly runs P with different inputs until an input that reaches the target is found (corresponding to a satisfying assignment to Q), or the fuzzer reaches a given time limit. The intuition behind applying a coverage-guided fuzzer is that it will relentlessly try to generate inputs that cover *new* code. In particular, the program location that returns 1 is a target for the fuzzer.

As an illustration of this idea, consider the example constraints in Listing 1, shown in SMT-LIBv2.5 format [8]. Free variables `a` and `b`, of type `Float64` (see §2.2), 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

[1] Any formula can be transformed to an equisatisfiable formula in conjunctive form, e.g, by using the linear-time Tseytin transformation [65].

Listing 1: An example conjunction of floating-point constraints in the SMT-LIBv2.5 format.

```
1   (declare-fun a () Float64)
2   (declare-fun b () Float64)
3   (define-fun div_rne () Float64 (fp.div RNE a b))
4   (define-fun div_rtp () Float64 (fp.div RTP a b))
5   (assert (not (fp.isNaN a)))
6   (assert (not (fp.isNaN b)))
7   (assert (not (fp.isNaN div_rne)))
8   (assert (not (fp.isNaN div_rtp)))
9   (assert (not (fp.eq div_rne div_rtp)))
10  (check-sat)
```

Listing 2: A translation of the constraints in Listing 1 to a C++ program.

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

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.

These constraints are satisfiable. Using C++ hexfloat notation, one satisfying assignment has a set to 0x0.410815d750e65p-1022 ($\approx 5.65235 \times 10^{-309}$) and b to 0x1.021c1b000e7cp+28 ($\approx 2.70648 \times 10^{8}$). Dividing a by b rounding to nearest (ties to even) yields 0x0.0000000408001p-1022 ($\approx 2.088452 \times 10^{-317}$) and rounding toward positive infinity results in 0x0.0000000408002p-1022 ($\approx 2.088453 \times 10^{-317}$).

A possible translation of these constraints into a C++ program is shown in Listing 2, where 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 1 is returned, the input corresponds to a satisfying assignment, otherwise the fuzzer proceeds to try another input.

The program first constructs the free variables from the input buffer data. Variables a and b correspond directly to the free variables a and b in Listing 1 and are constructed on lines 2 and 3 from the data buffer using bits 0 to 63 (a), and bits 64 to 127 (b).

An if statement checks whether a is NaN (line 4), encoding the constraint on line 5 of Listing 1. Whether b is NaN is handled analogously (line 5). Variable a_b_rne corresponds to the div_rne macro on line 3 of Listing 1, and is set to the result of calling div_rne(a, b) (line 6). This performs floating-point division rounding the result to the nearest value (ties to even). The assignment to a_b_rtp is analogous, with rounding towards positive infinity.

The checks for whether a_b_rne and a_b_rtp and NaN are handled similarly to the checks for whether a and b are NaN (lines 8 and 9). The comparison of a_b_rne and a_b_rtp (line 10) corresponds to the constraint on line 9 of Listing 1.

Finally, on line 11 the function returns 1, which tells the fuzzer that a satisfying assignment has been found. Note that this line is only reachable if all previously evaluated constraints were true.

There are multiple ways of encoding constraints as a program. Listing 2 uses the *fail-fast* encoding, discussed further in §4.3.

4 DESIGN AND IMPLEMENTATION OF JFS

JFS is written in C++11 and builds on several existing projects: the constraint language and API of Z3 [24] is used for in-memory constraint representation, allowing reuse of Z3's parser and constraint simplification tactics; Clang and LLVM are used to compile generated C++ code [36]; and the coverage-guided mutation-based fuzzer LibFuzzer [40] is used to fuzz the resulting binary.

JFS accepts an SMT-LIBv2 [8] formula consisting of a conjunction of top-level constraints. Program analysis tools—such as those based on dynamic symbolic execution [15], but not only [66]—generate such conjunctions directly, and as mentioned in §3, any formula can be transformed to an equisatisfiable formula in conjunctive form, e.g, by using the linear-time Tseytin transformation [65].

The design of JFS in principle supports finding satisfying assignments to any theory using finite data types. Our current implementation supports combinations of the Core (i.e. Boolean), FixedSizeBitVectors, and FloatingPoint SMT-LIBv2 theories, over Float32 and Float64 floating-point variables, and bitvector variables of arbitrary widths up to 64 bits.

We now discuss practical issues related to the design of JFS, covering simplification of formulas pre-fuzzing (§4.1); the mapping of formula variables to the program input buffer (§4.2); choices for how to encode the formula as a program (§4.3); and the injection of "smart seeds" to guide the fuzzer (§4.4). We also briefly discuss JFS's runtime library (§4.5).

4.1 Formula Simplification

To make the C++ program that JFS will ultimately generate more friendly to LibFuzzer, JFS first applies the simplification passes detailed in Table 1, in order, to the input formula. The table indicates which passes were already available via calls into the Z3 library, vs. which we implemented using the Z3 API. These passes represent various cheap ways to simplify formulas that we observed to be useful useful during early prototyping of JFS. We remark briefly on the *And* hoisting pass: JFS uses Z3 to parse constraints, and parsing always returns a single conjunct; the AND hosting pass simply splits this into independent conjuncts.

If after simplification the formula is syntactically equivalent to *false*, JFS immediately reports UNSAT without invoking LibFuzzer.

4.2 Input Buffer Preparation

Having simplified the formula, JFS must decide how to represent free variables of the formula in the program's input buffer.

First, an *equality extraction* pass is used to partition the free variables and constants appearing in the formula into equivalence classes based on syntactic equalities, such that members of the same equivalence class are guaranteed to be constrained to be equal. Each resulting class contains at most one constant: if multiple distinct constants were constrained to be equal, JFS would have reported the formula as trivially UNSAT after formula simplification (§4.1).

Each equivalence class is then considered. If a class contains a constant c then there is no need to reserve space for variable of the class in the input buffer: each variable is declared in the

Table 1: The ordered set of simplifying passes run by JFS on a formula before program generation

Simplification	Description	Already in Z3?
And hoisting	Separates the constraint `(and a b)` into two separate constraints	No
Constant propagation	Apply Z3's `propagate-values` tactic to propagate constants	Yes
Duplicate constraint elimination	Removes duplicate constraints from the constraint set	No
Expression simplification	Invokes Z3's expression simplifier, which performs e.g. constant folding	Yes
Simplify contradictions	Replaces `(and a (not a))` with *false*	No
True elimination	Removes constraints of the form *true* from the constraint set	No

Listing 3: Example constraints used to illustrate equality extraction.

```
(declare-fun a () (_ FloatingPoint 11 53))
(declare-fun b () (_ FloatingPoint 11 53))
(declare-fun c () (_ FloatingPoint 11 53))
(declare-fun d () (_ FloatingPoint 11 53))
(assert (= a b))
(assert (= b c))
(assert (= d (_ +zero 11 53)))
(assert (not (fp.isNaN (fp.add RNE c d))))
(check-sat)
```

program and initialized to c. Otherwise, k bits of the input buffer are allocated to represent the common value of all free variables in the class, where k is the width of the associated data type (e.g. $k = 32$ for `Float32` variables). The variables are all declared locally in the program and initialized via the same k bits of the input buffer.

This process is illustrated by the formula of Listing 3 (where `(_ + zero 11 53)` denotes the 64-bit positive zero constant) and the associated program in Listing 4. The equivalence classes are $\{a, b, c\}$ and $\{d, 0.0\}$. As a result, the input buffer `data` requires 8 bytes in order to store the double-precision value common to a, b and c. The `makeFloatFromData` function initializes a via this buffer, and the value of a is then copied into b and c. Variable d does not require associated space in the buffer: it is initialized with the constant value 0.0. Because this process fully accounts for equality constraints between variables and constants, such constraints do not need to be modelled in the control flow of the generated program.

Equality extraction both reduces the size of the input buffer, and alleviates LibFuzzer from the onerous task of guessing equality between certain sets of variables.

The input buffer is tightly packed, so that the chunks of data associated with variables need not be aligned to word or even byte boundaries. Chunks are ordered by the order they appear while traversing the input formula. This makes the order deterministic (useful for reproducibility) but arbitrary. Non-aligned accesses make reading from the buffer sub-optimal, but avoids padding bits that have no impact on program behaviour. Such bits would be detrimental to LibFuzzer as it would waste time attempting to mutate those bits to increase coverage. With additional engineering effort we could adapt JFS to make LibFuzzer aware of padding bits and instruct it not to mutate them, allowing the performance benefits associated with better alignment.

4.3 Program Encodings

We have experimented with two ways to encode an SMT formula as a program: *fail-fast* and *try-all*.

Listing 4: A translation of the constraints in Listing 3 to a C++ program based on the equality extraction pass.

```
int FuzzerTestOneInput(const uint8_t* data, size_t size) {
  double a = makeFloatFrom(data, size, 0, 63);
  double b = a;
  double c = a;
  double d = 0.0;
  double c_plus_d = add_rne(c, d);
  if (!isnan(c_plus_d)) {} else return 0;
  return 1; // TARGET REACHED
}
```

Listing 5: A translation of the constraints in Listing 1 to a C++ program using the *try-all* encoding.

```
int FuzzerTestOneInput(const uint8_t* data, size_t size) {
  double a = makeFloatFrom(data, size, 0, 63);
  double b = makeFloatFrom(data, size, 64, 127);
  size_t counter = 0;
  if (!isnan(a)) ++counter;
  if (!isnan(b)) ++counter;
  double a_b_rne = div_rne(a, b);
  double a_b_rtp = div_rtp(a, b);
  if (a_b_rne != a_b_rtp) ++counter;
  if (!isnan(a_b_rne)) ++counter;
  if (!isnan(a_b_rtp)) ++counter;
  if (counter != 5)
    return 0;
  return 1; // TARGET REACHED
}
```

With the *fail-fast* encoding (Listings 2 and 4) the program exits as soon as an unsatisfied conjunct is found, without evaluating the remaining conjuncts. A satisfying assignment is found if and only if the end of the program is reached. With the *try-all* encoding (Listings 1 and 5) all n conjuncts of the input formula are evaluated, and a zero-initialised counter is incremented each time a conjunct is found to hold. A satisfying assignment is found if and only if the counter equals n at the end of the program.

The potential advantage of *try-all* is that evaluating *every* constraint provides rich coverage information: if an input satisfies some previously-unsatisfied conjunct, coverage will increase and the coverage-guided fuzzer will store the input to be considered for further mutation. The potential advantage of *fail-fast* is that it does not waste time further evaluating an input once it is known that it does not satisfy some constraint. Experimentally we have found that *fail-fast* enables JFS to solve significantly more benchmarks than *try-all*, thus we only consider the *fail-fast* encoding in our evaluation (§5).

4.4 Smart Seeds

As discussed in §2.1, a coverage-guided mutation-based fuzzer relies on a corpus of initial seed inputs, which in the case of JFS are initial valuations of the input buffer. We have experimented with two modes for selecting seeds.

In *naive seeds* mode, JFS generates two seeds: a buffer of all zeros and a buffer of all ones, which at least provide LibFuzzer's crossover mutator with a pair of diverse inputs to work with.

In *smart seeds* mode, seeds are generated as follows. For each distinct data type associated with a free variable (e.g. `Float64`, `bv32`, etc.), we construct a set consisting of (1) special values for that type, such as positive/negative zero, infinities, and NaN bit patterns for floating-point types (see §2.2), and bit patterns encoding 0, 1 and -1 for bitvector types; and (2) values of constants of the given type that appear in the input formula. We then construct a seed by randomly sampling from the space of possible input permutations that can be generated from these sets. The number of seeds selected is configurable and set to 100 by default.

Our hypotheses for why smart seeds may be valuable are that (1) special values are often important for particular data types (e.g. a floating-point formula that looks unsatisfiable on first sight often turns out to be satisfiable due to the subtle semantics of NaN values), and (2) the satisfiability of constraints is more likely to depend on values equal or similar to values appearing in the formula than on arbitrary values (with mutations of seeds being likely to yield said similar values). We evaluate the benefits of smart seeds experimentally in §5.

4.5 Runtime Library

The program that JFS generates calls into a runtime library that implements the semantics of relevant `FloatingPoint` and `BitVector` types from the SMT-LIBv2 standard, handling rounding modes that are natively supported by the `x86_64` architecture (all modes except round to nearest, ties to away from zero).

5 EVALUATION

We now turn to the evaluation of JFS, comparing it against seven state-of-the-art SMT solvers that support solving floating-point constraints. We discuss the benchmark selection process (§5.1), the solvers and how we configured them (§5.2), and our experimental setup (§5.3). We then present the results of the experiments (§5.4), in the context of the research questions identified in §1. We have made the source code of JFS and all our data sets publicly available.[2,3]

5.1 Benchmark Selection

Table 2 summarises the `QF_FP`, `QF_BVFP` and `QF_BV` SMT-LIB suites from which we have drawn benchmarks for our experiments. A subset of these suites are used in SMT-COMP, the annual SMT solver competition. All benchmarks are quantifier-free (`QF`), beyond which the suites are built over floating-point (`QF_FP`), bitvector (`QF_BV`), and a combination of bitvector and floating-point (`QF_BVFP`) types.

For each suite, the **Suite** column provides a reference to the git repository and SHA-1 hash associated with the version of the suite that we used. The **SAT** and **UNSAT** columns under **Unpruned** show the total number of benchmarks in each suite either already labelled SAT or UNSAT, or that were unlabelled but could be classified empirically as SAT or UNSAT by either MathSAT5 or Z3 within 900 seconds seconds on our test platform. The **UNKNOWN** and **Total** columns show the number of benchmarks that remained unlabelled, and the total number of benchmarks, respectively.

Since JFS is not designed to prove unsatisfiability, we pruned all benchmarks labelled UNSAT. We also pruned all benchmarks for which the pre-processing steps performed by JFS (§4.1) reduced the benchmark to contain only constants. We believe it was important to remove such trivial benchmarks to focus our evaluation on the effectiveness of fuzzing for constraint solving, rather than the effectiveness of these well-known pre-processing steps. The pruned benchmarks are summarized under **Non-trivial, no UNSAT** in Table 2. Notice that many SAT benchmarks were found to be trivial, including the vast majority of the `QF_FP` suite.

The large numbers of remaining `QF_BVFP` and `QF_BV` benchmarks would require prohibitive computation resources for our experiments. Therefore, in a final step, we sampled a subset of these benchmarks. To make sure we include benchmarks of varying difficulty, we performed *stratified random sampling* [4] based on the performance of both MathSAT5 and Z3. That is, for each benchmark suite, we computed two histograms (one for MathSAT5 and one for Z3) of solver execution time with five-second-wide bins. To select a benchmark, first a histogram is selected (round-robin), then a histogram bin is selected (random), and then a benchmark is selected from that bin (random). This process was repeated until the desired number of benchmarks were selected. We selected 5% of the benchmarks from each of the pruned `QF_BVFP` and `QF_BV` suites, using the pruned `QF_FP` suite in its entirety. Details of the final benchmark subsets are summarised under **Final subsets** in Table 2, which we refer to as $\texttt{QF\_BVFP}_{fs}$, $\texttt{QF\_FP}_{fs}$ and $\texttt{QF\_BV}_{fs}$, respectively (where *fs* stands for "final subset").

5.2 Solver Configurations

We compare JFS against seven state-of-the art constraint solvers for floating-point constraints. For each solver, Table 3 summarizes the version (v) or revision (r) used, and the main technique on which the solver is based. We also include a synthetic portfolio solver (JFS+MathSAT5) to aid discussions of using JFS in a portfolio setting. JFS+MathSAT5 models a complete portfolio solver that runs both JFS-LF-SS and MathSAT5 in parallel and returns the answer from which ever solver answers first. It is synthetic because solving time is computed as the minimum of the solving times of existing runs of JFS-LF-SS and MathSAT5. JFS-LF-SS and MathSAT5 are combined because they are the best performing JFS configuration (§5.4.1) and solver for $\texttt{QF\_FP}_{fs}$ (§5.4.2) respectively.

We acknowledge that some of these solvers are capable of proving UNSAT as well as SAT, while JFS is only capable of proving SAT. This might appear to give JFS an advantage, but we are not aware of any way to configure those solvers to only focus on SAT, hence we believe there is no fairer way of performing the comparison.

At the time experiments were run, XSat had not been officially released; we use a version of the solver uploaded to STAR-EXEC[4] for the 2017 SMT-COMP competition.

[2] https://github.com/mc-imperial/jfs

[3] https://github.com/mc-imperial/jfs-fse-2019-artifact

[4] https://www.starexec.org/

Table 2: Summary of the SMT-LIB benchmark suites we use as a basis for our experiments.

	Unpruned				**Non-trivial, no UNSAT**			**Final subset (fs)**		
Suite	**SAT**	**UNSAT**	**UNKNOWN**	**Total**	**SAT**	**UNKNOWN**	**Total**	**SAT**	**UNKNOWN**	**Total**
`QF_FP` [61]	20,125	20,142	35	40,302	125	35	160	125	35	160
`QF_BVFP` [60]	14,033	3179	3	17,215	14,033	3	14,036	699	3	702
`QF_BV` [59]	11,283	20,991	133	32,407	9495	133	9628	466	16	482

Table 3: The solvers compared in our experiments.

Solver	**Version**	**Technique**
COLIBRI [13]	r1572	Interval solving
CORAL [62]	v0.7	Meta-heuristic search
CVC4 [7]	v1.6	Bit-blasting
goSAT [9]	rb5a423c	Mathematical optimisation
JFS (this paper)	r5ceecd1	Coverage-guided fuzzing
MathSAT5 [17]	v5.5.1	Bit-blasting
XSat [26]	See text	Mathematical optimisation
Z3 [24]	v4.6.0	Bit-blasting

The CORAL, goSAT and XSat solvers do not support bitvector reasoning, thus we can only apply them to the `QF_FP`$_{fs}$ benchmark suite. Instead of the SMT-LIBv2.5 format, CORAL uses its own constraint language that only supports a subset of the semantics of the `QF_FP` theory. To allow a best-effort comparison with CORAL, we have implemented a tool to convert SMT-LIBv2.5 constraints into this language.

We run each solver using its default configuration, edited if necessary to enable floating-point reasoning and to enforce SMT-LIB compliance. Exceptions are CORAL, which we run using options suggested by the developers as we were unsure how to best invoke the solver, and MathSAT5, which comes with a file describing preferred options for each benchmark suite (`smtcomp2015_main.txt`).

We run CORAL in two distinct modes: alternating variable method (CORAL-AVM) and particle swarm optimisation (CORAL-PSO). We run JFS in three modes: using LibFuzzer with naive seeds (JFS-LF-NS), using LibFuzzer with smart seeds (JFS-LF-SS), and using purely naive random input generation, i.e. without LibFuzzer (JFS-NR). In all cases the *fail-fast* encoding is used (see §4.3).

Where solvers support setting a random seed, we use a fixed per-solver seed to try to ensure reproducible results.

5.3 Experimental Setup

We ran the 11 configurations (eight solvers, with CORAL in two and JFS in three configurations respectively) on a machine with two Intel® Xeon® E5-2450 v2 CPUs (8 physical cores each) with 256GiB of RAM running Ubuntu `16.04LTS`. Each solver was run five times per benchmark with a timeout of 900 seconds per run and with a fixed random seed (if supported). The repeat runs of a solver are used to compute average execution time and observe non-deterministic behaviour. To allow experiments to complete within a reasonable time-frame, each solver was executed in parallel over the set of benchmarks, with at most 13 benchmarks running in parallel.

Each time a solver is run on a benchmark we record a result label. If solver reports UNKNOWN, crashes, or hits the memory or time limit, the result is labelled as UNKNOWN. If the solver reports SAT (UNSAT) and that matches the expected satisfiability of the benchmark or the expected satisfiability is UNKNOWN then the result is labelled as SAT (UNSAT). If the solver reports SAT (UNSAT) and the expected satisfiability of the benchmark is UNSAT (SAT) then the result is labelled as WRONG.

We combined results labels for repeat runs of a solver on a benchmark as follows: If at least one label is SAT (UNSAT) and all labels are either SAT (UNSAT) or UNKNOWN, the combined label is SAT (UNSAT). If at least one label is WRONG or the labels include a mixture of SAT and UNSAT, the combined label is WRONG. Otherwise, in the case where all labels are UNKNOWN, the combined label is UNKNOWN.

To combine the execution times (wall clock time), the arithmetic mean and confidence intervals (99.9%) are computed. Mean execution times are only considered distinguishable between solvers if their confidence intervals do not overlap.

5.4 Results

We now present and discuss our experimental results, relating them to the research questions identified in §1. In §5.4.1 we address RQ1 and RQ2 by comparing different JFS configurations. Then in §5.4.2 we compare the overall best JFS configuration found in §5.4.1 against other solvers in order to address RQ3.

To visualise solver performance we use quantile plots (e.g. Figure 1). Each curve on a plot corresponds to a solver configuration. A curve is plotted by computing a score for each run on a benchmark (1 for *correct*, −1 for *wrong*, and 0 for *unknown*), sorting *correct* results by solver execution time and then plotting accumulating score against solver execution time. An extra leftmost point is then added to the curve and all other points are offset along the x-axis by this value. The x value of this point is the sum of negative scores.

The resulting quantile plot has the following properties: (1) the x-value of the leftmost point on a curve indicates the number of incorrect solver answers (e.g. a value of −5 on the x-axis indicates the solver incorrectly reported satisfiability on five benchmarks); therefore, the x-value of the leftmost points can be compared between curves (ranked by least number of wrong answers); (2) the x-value of the right-most point on a curve is the difference between the number of correct vs. incorrect solver answers; therefore, the x-value of the rightmost points can be compared between curves (ranked by total solver score); (3) the total execution time of a solver on correctly solved benchmarks is equal to the area under the curve. We cannot compare the points with the same y-value between curves because the points do not necessarily refer to the same benchmark. However, we can compare the general shapes of curves. The quantile plots that follow are best viewed in colour.

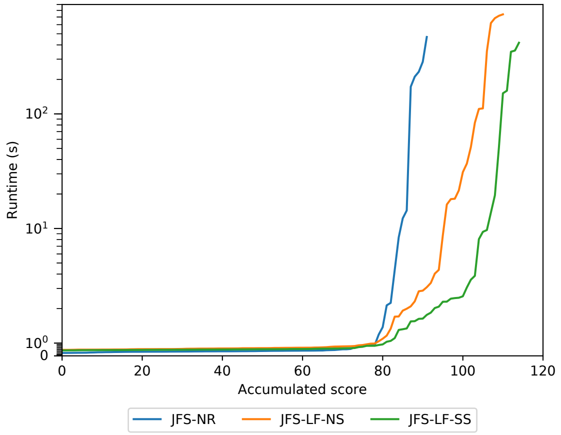

Figure 1: Comparing JFS configurations over QF_FP$_{fs}$.

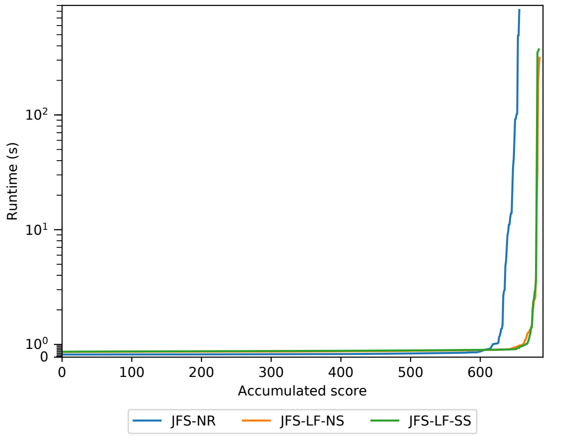

Figure 2: Comparing JFS configurations over QF_BVFP$_{fs}$.

5.4.1 JFS Configuration Comparison. We compare JFS in three different configurations JFS-LF-NS, JFS-LF-SS, JFS-NR (§5.2) on the three benchmark suites.

On the QF_BV$_{fs}$ suite, all JFS configurations performed poorly: 95.44% of the benchmarks could not be solved by any configuration, with very little difference in performance between the configurations. We discuss the poor performance of JFS on this suite in §5.4.2, restricting our attention to QF_FP$_{fs}$ and QF_BVFP$_{fs}$ for the remainder of this subsection.

The quantile plots of Figures 1 and 2 summarise the performance of the JFS configurations over the QF_FP$_{fs}$ and QF_BVFP$_{fs}$ benchmarks, respectively. The zero leftmost x-values of all curves indicates that no incorrect results were produced (this also holds for QF_BV$_{fs}$).

For QF_FP$_{fs}$ (Figure 1), the right-most x-value of each curve shows that JFS-LF-SS solved the most benchmarks (114), followed by JFS-LF-NS (110), and finally by JFS-NR (91), providing positive support for RQ1 and RQ2. The shape of the curves shows that JFS-LF-SS is generally faster than both JFS-LF-NS and JFS-NR (smaller area under curve if curve widths are normalised), further supporting RQ2. However, upon investigation we noticed that JFS-LF-SS was the fastest configuration for 22 benchmarks, JFS-LF-NS for 6, and JFS-NR for 24. For the remaining benchmarks, it was not possible to determine which configuration was fastest, either because the solver execution time confidence intervals overlapped or because none of the solvers reported SAT. It is expected that JFS-NR might sometimes be faster because it has lower overhead than the other configurations (e.g. no coverage instrumentation, no seeds to read).

Table 4: JFS-LF-SS vs. other JFS configurations over QF_FP$_{fs}$.

Solver	Both	Only LF-SS	Only other	Neither
JFS-LF-NS	108 (67.50%)	6 (3.75%)	2 (1.25%)	44 (27.50%)
JFS-NR	90 (56.25%)	24 (15.0%)	1 (0.62%)	45 (28.12%)
All above	108 (67.50%)	6 (3.75%)	3 (1.88%)	43 (26.88%)

For QF_BVFP$_{fs}$, Figure 2 shows that JFS-LF-NS solved the most benchmarks (685), followed by JFS-LF-SS (684) and finally JFS-NR (656). We can see that the shape of the curves for the LibFuzzer configurations are similar, suggesting little difference in overall performance between them. However, the naive random configuration is clearly worse. These results provide positive support for RQ1, and are inconclusive with respect to RQ2.

Quantile plots do not tell the complete story. Tables 4 and 5 show JFS-LF-SS similarity, complementarity, and limitations for the QF_FP$_{fs}$ and QF_BVFP$_{fs}$ benchmarks, compared to the other JFS configurations. In each table, the **Both** column states the number of benchmarks shown to be satisfiable by both JFS-LF-SS and the other JFS configuration. The **Only LF-SS** (**Only other**) column shows the number of benchmarks that were shown to be satisfiable by JFS-LF-SS (other configuration) and not by the other configuration (JFS-LF-SS). The **Neither** column shows the number of benchmarks that were shown to be satisfiable by neither JFS-LF-SS nor the other configuration. Each row of the table corresponds to the *other* solver (specified by the **Solver** column). The "All above" row has a special meaning and is a combination of all the above results. For the "All above" row: the **Both** table cell is the union of all benchmarks that both JFS-LF-SS and another JFS configuration managed to solve (i.e. it is a union of intersections, not an intersection of intersections); the **Only LF-SS** table cell is the number of benchmarks found satisfiable by JFS-LF-SS and none of the other configurations; the **Only other** table cell is the union of all benchmarks found to be satisfiable by another configuration and not JFS-LF-SS; and the **Neither** table cell is the number of benchmarks not found satisfiable by any JFS configuration.

For QF_FP$_{fs}$, Table 4 shows that JFS-LF-SS and JFS-LF-NS are quite similar (67.50% of the benchmarks solved by both and 27.50% by neither); perhaps unsurprising given that they only differ in the seeds fed to LibFuzzer. By contrast, JFS-NR is less similar, with 15.0% of benchmarks solved only by JFS-LF-SS.

In terms of complementarity, JFS-LF-SS always solved benchmarks that the other configurations did not. Although the converse is true (other configurations solving benchmarks that JFS-LF-SS did not) it is less frequent. Looking at limitations, 26.88% of the benchmarks were not solved by any JFS configuration.

For QF_BVFP$_{fs}$, while the quantile plot of Figure 2 suggests that JFS-LF-NS performs slightly better than JFS-LF-SS due to the number of benchmark solved, Table 5 shows that there are two benchmarks that only JFS-LF-SS solved and three that only JFS-LF-NS

Table 5: JFS-LF-SS vs. other JFS configurations over QF_BVFP$_{fs}$.

Solver	Both	Only LF-SS	Only other	Neither
JFS-LF-NS	682 (97.15%)	2 (0.28%)	3 (0.43%)	15 (2.14%)
JFS-NR	655 (93.30%)	29 (4.13%)	1 (0.14%)	17 (2.42%)
All above	682 (97.15%)	2 (0.28%)	3 (0.43%)	15 (2.14%)

solved, showing that neither configuration is strictly superior to the other on this benchmark suite. Regarding limitations, the JFS configurations performed collectively well on this suite, with only 2.14% not solved by any JFS configuration.

Overall, for formulas involving floating-point constraints, the results of this subsection show that using coverage-guided fuzzing over naive random input generation offers benefit, supporting RQ1. The results also partially support RQ2 in this domain, showing that smart seeds improve the performance of JFS over QF_FP$_{fs}$. While the performance results for JFS-LF-SS and JFS-LF-NS over QF_BVFP$_{fs}$ do not reveal a clear winner, we use JFS-LF-SS as the JFS configuration for comparison against other solvers in §5.4.2 due to its superior performance on the QF_FP$_{fs}$ suite.

5.4.2 *JFS Compared with Other Solvers.*

We now address RQ3 by comparing the JFS-LF-SS configuration of JFS against seven solvers on the QF_FP$_{fs}$ benchmarks and four on the QF_BVFP$_{fs}$ and QF_BV$_{fs}$ benchmarks.

Comparison over QF_FP$_{fs}$. The quantile plot of Figure 3 summarises performance results for the eight non-portfolio solvers plus JFS+MathSAT5 over QF_FP$_{fs}$ benchmarks. The leftmost points for XSat and COLIBRI indicate that they gave 34 and 5 wrong answers, respectively. In all cases this was due to UNSAT being reported for a SAT-labelled benchmark.

In terms of the number of benchmarks found to be satisfiable, JFS+MathSAT5 was the most successful (126) followed by MathSAT5 (125), JFS (114), CVC4 (110), COLIBRI (104), Z3 (102), goSAT (91), XSat (69), CORAL-PSO (60), and finally CORAL-AVM (31). Even though JFS does not rank the highest by number of benchmarks solved, we can see from the shape of the curves that JFS's total solving time is significantly smaller than MathSAT5's which solved the most benchmarks out of the non-portfolio solvers. The JFS+MathSAT5 synthetic portfolio solver illustrates that a portfolio combination of JFS-LF-SS and MathSAT5 would perform well because it would solve the most benchmarks and in less time on average.

Table 6 shows JFS's capability, complementarity, and limitations for the QF_FP$_{fs}$ benchmarks. The columns and special **All above** row have the same meaning as discussed for Table 4 in §5.4.1. Table 6 shows great deal of similarity (**Both** column) between MathSAT5 and JFS, followed by COLIBRI and CVC4, and then Z3. The similarity with the other search-based solvers (CORAL-AVM, CORAL-PSO, goSAT, and XSat) is somewhat lower.

JFS complements every other non-portfolio solver, i.e. there is at least one benchmark that JFS can solve and the other solver cannot. However, every benchmark solved by JFS can be solved by at least one other solver. For the search-based solvers (CORAL, goSAT, JFS, and XSat) JFS finds many benchmarks to be satisfiable that the other solver does not. This shows that out of the all the search-based solvers, JFS is the most competitive on the QF_FP$_{fs}$ benchmark suite.

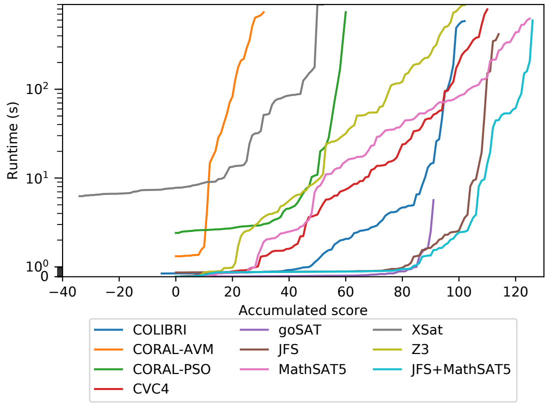

Figure 3: Quantile plot comparing the performance of solvers on the QF_FP$_{fs}$ benchmarks

Table 6: JFS compared to other solvers over QF_FP$_{fs}$.

Solver	Both	Only JFS	Only other	Neither
COLIBRI	98 (61.25%)	16 (10.00%)	6 (3.75%)	40 (25.00%)
CORAL-AVM	31 (19.38%)	83 (51.88%)	0 (0.00%)	46 (28.75%)
CORAL-PSO	59 (36.88%)	55 (34.38%)	1 (0.62%)	45 (28.12%)
CVC4	98 (61.25%)	16 (10.00%)	12 (7.50%)	34 (21.25%)
goSAT	86 (53.75%)	28 (17.50%)	5 (3.12%)	41 (25.62%)
MathSAT5	113 (70.62%)	1 (0.62%)	12 (7.50%)	34 (21.25%)
XSat	62 (38.75%)	52 (32.50%)	7 (4.38%)	39 (24.38%)
Z3	96 (60.00%)	18 (11.25%)	6 (3.75%)	40 (25.00%)
All above	114 (71.25%)	0 (0.00%)	21 (13.12%)	25 (15.62%)

In terms of limitations, every solver except CORAL-AVM finds some benchmarks to be satisfiable that JFS does not (i.e. most solvers are able to complement JFS). There are also some benchmarks that neither JFS, nor another solver manage to show as satisfiable.

JFS is also complementary in terms of execution time. Figures 4 and 5 show scatter plots comparing the execution time of JFS against MathSAT5 and CVC4 respectively. We show MathSAT5 and CVC4 here because these are the solvers that found the highest number of benchmarks to be satisfiable that JFS did not. On these plots, each point represents a benchmark. A diagonal line ($y = x$) is drawn, upon which a benchmark would lie if both solvers solved the benchmark in an identical amount of time. Points that appear below the diagonal are cases where JFS was faster, and points above the line are cases where the other solver was faster. The number of points where this is the case (and where confidence intervals do not overlap) are shown on the figures along with annotation indicating how many points are cases where both solvers reached a timeout. These plots only show cases where both solvers either reported SAT or reached a timeout because it does not make sense to compare execution times if one of the solvers crashed. The plots show that the solvers are highly complementary, with JFS being faster for 86 benchmarks in each case, while MathSAT5 was faster for 27 benchmarks and CVC4 for 24.

In relation to RQ3, these results show that JFS is very competitive with other solvers on the QF_FP$_{fs}$ benchmarks and is able to complement every solver considered.

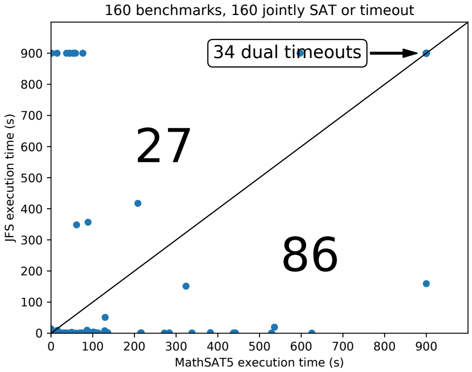

Figure 4: Scatter plot comparing the execution time of MathSAT5 and JFS on the `QF_FP`$_{fs}$ benchmark.

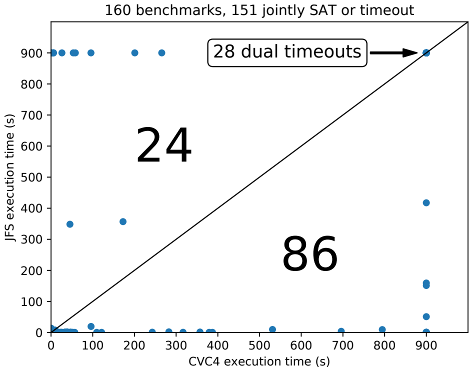

Figure 5: Scatter plot comparing the execution time of CVC4 and JFS on the `QF_FP`$_{fs}$ benchmark.

Comparison over QF_BVFP$_{fs}$. Figure 6 shows a quantile plot comparing JFS against JFS+MathSAT5 and the other three non-portfolio solvers that support the `QF_BVFP`$_{fs}$ suite. The plot shows that the none of the solvers report incorrect answers and that they all report a similar number of benchmarks as satisfiable. JFS+MathSAT5, CVC4, MathSAT5 and Z3 report 699 benchmarks as satisfiable, followed by JFS with 684, and COLIBRI with 666. The figure also shows that for every solver, over 600 benchmarks were solved in under a second. This suggests that the benchmark suite (despite our best efforts during stratified sampling) is not well balanced in terms of difficulty and may not accurately reflect the kind of constraints that might be encountered in practice. Table 7 shows the similarity, complementarity, and limitations of JFS on this benchmark compared to other non-portfolio solvers. The table shows a high degree of similarity between the solvers and that JFS is only able to complement COLIBRI. Every solver is able to solve benchmarks that JFS is unable to solve. However, if we make scatter plots comparing the execution time of JFS with that of other solvers, we find in each case a significant number of benchmarks where JFS solves the constraints faster (56 faster than CVC4, 27 faster than COLIBRI, 55 faster than MathSAT5, and 69 faster than Z3). We omit these plots for brevity but they look very similar to Figures 4 and 5.

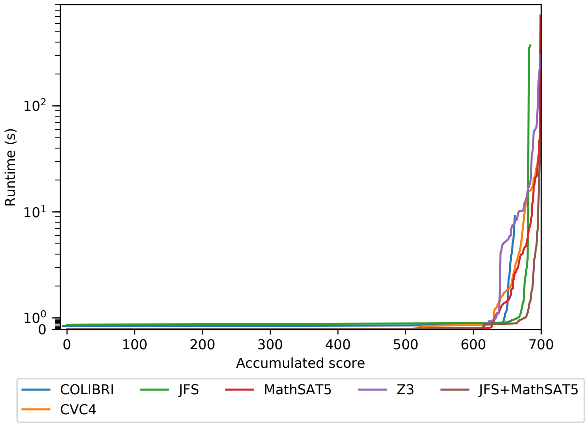

Figure 6: Quantile plot comparing the performance of solvers on the `QF_BVFP`$_{fs}$ benchmarks.

Table 7: JFS compared to other solvers over `QF_BVFP`$_{fs}$.

Solver	Both	Only JFS	Only other	Neither
COLIBRI	661 (94.16%)	23 (3.28%)	5 (0.71%)	13 (1.85%)
CVC4	684 (97.44%)	0 (0.00%)	15 (2.14%)	3 (0.43%)
MathSAT5	684 (97.44%)	0 (0.00%)	15 (2.14%)	3 (0.43%)
Z3	684 (97.44%)	0 (0.00%)	15 (2.14%)	3 (0.43%)
All above	684 (97.44%)	0 (0.00%)	15 (2.14%)	3 (0.43%)

With reference to RQ3, these results show that JFS is competitive over `QF_BVFP`$_{fs}$, complementing COLIBRI in the number of benchmarks solved, and all other solvers in terms of execution time. However, as discussed the results across all solvers suggest that `QF_BVFP`$_{fs}$ may not be an especially challenging suite.

Comparison over QF_BV$_{fs}$. JFS is not competitive on the `QF_BV`$_{fs}$ suite, finding only 22 benchmarks satisfiable, compared to e.g. 419 for Z3 and 344 for MathSAT5. (We omit the associated quantile plot for space reasons.) However, for each solver except CVC4, there are always cases where JFS is able to solve some benchmarks faster.

We suspect two main reasons for the poor performance of JFS on the bitvector-only theory, compared to the theories involving floating point. First, floating-point constraints result in much more complex circuits, which often blow-up the underlying SAT solvers used by state-of-the-art SMT solvers. As a result, a more lightweight approach like the one used by JFS is competitive on these theories.

Second, bitvector solvers have been available for over a decade, which has allowed a set of difficult and challenging benchmarks to be developed over a long period of time. These benchmarks likely evolved in difficulty as bitvector solvers gradually increased their capability. On the other hand, solvers for floating-point constraints are comparatively new and have had much less time to develop. As a consequence, the available floating-point benchmarks are a reflection of the relatively immature floating-point constraint solvers currently available.

It is also worth drawing an analogy with coverage-guided fuzzers applied to bug finding (their usual domain). These fuzzers are typically good at finding shallow bugs, and can only excel at finding deep bugs with a large amount of compute time, good seeds, or domain-specific knowledge. It could be the case that the floating-point benchmarks currently available in SMT-LIB are the equivalent of shallow programs, where bugs are easy for a fuzzer to find.

In summary, with respect to RQ3: the results across all three benchmark suites show that JFS is highly competitive on two suites (both involving floating point), and uncompetitive on the bitvector benchmark suite.

6 RELATED WORK

There is a large body of existing work that seeks to improve solving floating-point constraints. The CORAL [62] and FloPSy [35] solvers apply meta-heuristic search techniques to try to find satisfying assignments to floating-point constraints. Like JFS, these methods are incomplete because they can only show satisfiability. All solvers construct a fitness function which they attempt to maximise. JFS's fitness function is coarse—the number of new branches covered—in contrast to CORAL's and FloPSy's fitness functions, which gradually change as candidate solutions get closer to a satisfying assignment. Despite the coarseness of JFS's fitness function, our results show that JFS performs better overall than CORAL, both in terms of the number of benchmarks it can show to be satisfiable, and in execution time. We could not easily compare with FloPSy due to its tight integration with Pex [64], the symbolic execution tool it is designed to work with.

CORAL supports using an interval solver to improve the quality of its initial candidate inputs. It's likely we could apply a similar approach in JFS to generate higher quality seeds for the fuzzer.

The goSAT [9] and XSat [26] solvers both reformulate finding a satisfying assignment as a mathematical optimisation problem and apply existing mathematical optimisation algorithms to try to find a global minimum. This is similar in spirit to JFS, FloPSy and CORAL in that the functions that goSAT and XSat seek to minimise are essentially fitness functions. The difference is in the algorithms used to perform the search. Like JFS, this strategy is incomplete. Again, despite JFS's coarser fitness functions, the experimental evaluation found JFS to perform better on those benchmarks.

CVC4 [6], MathSAT5 [17], SONOLAR [51] and Z3 [24] solve floating-point constraints by transforming floating-point operations into bitvector circuits and then bit-blasting these into a SAT problem. This problem is then solved using a SAT solver. Unlike JFS, these solvers are complete, but they can end up generating very large SAT problems, which are difficult to solve. Like JFS, these solvers support a combination of the bitvector and floating-point SMT-LIBv2.5 theories. Our comparison with CVC4, MathSAT5 and Z3 indicates that the approaches are complementary, particularly for the floating-point benchmarks, suggesting these solvers would likely benefit from incorporating a JFS-style search-based strategy with their existing strategies, to form a portfolio solver. We did not compare JFS with SONOLAR, but given that its design is similar to that of SAT based solvers, we do not expect such experiments to change our main conclusions. A prior study comparing SAT-based solving with random and heuristic solvers also found that a portfolio approach performs best [63].

COLIBRI [13] and FPCS [46] use interval solving as a complete method for solving floating-point constraints. As for the comparison with SAT based solvers, our comparison with COLIBRI showed complementarity, suggesting that these solvers could also benefit from incorporating a search-based strategy. We did not compare against FPCS because it is not publicly available.

REALIZER [38] tries to solve floating-point constraints by transforming (in an equisatisfiable manner) floating-point constraints into constraints over reals, using Z3 as a back-end to solve these constraints. REALIZER's strategy is particularly suitable for working with constraints that check the accuracy of floating-point expressions compared to their real counterparts. JFS cannot do this because it cannot handle constraints over reals. We have not yet had time to compare JFS with REALIZER.

More generally, floating-point constraint solving has gathered a lot of attention from the research community, with several tools based on symbolic execution, model checking, abstract interpretation, etc. using it to perform test-case generation, precision tuning, verification, equivalence checking, peephole optimizations, branch instability assessment, etc. involving floating-point code [1, 3, 5, 10–12, 16, 19–23, 27, 30, 35, 37, 42, 43, 47, 49, 52–56].

7 CONCLUSION

We have investigated using coverage-guided mutation-based fuzzing to prove satisfiability of SMT formulas over finite variable domains, and floating-point constraints in particular, via a prototype solver, JFS. Our main experimental findings are that in the domain of floating-point constraints, solving via coverage-guided fuzzing outperforms solving via naive fuzzing, and performance can be further improved by generating initial seeds in a smart manner; JFS is highly competitive with and complementary to all solvers we compared with in the floating-point domain; and JFS is much less effective when applied to the domain of bitvectors. Our synthetic portfolio solving results indicate that JFS's complementary nature would make it a useful component in a portfolio solver.

In future work, we would like to better understand the properties of benchmarks that dictate whether JFS performs well, with a view to developing heuristics to help decide when it would be beneficial to apply JFS. A first step in this direction would be to use model counting solvers to understand whether suitability for solving via fuzzing relates to number of solutions. A practical problem here is that model counting suffers from limited scalability.

Regarding our *smart seeds*, *smarter* seeds could be generated based on domain-specific knowledge about the context in which JFS is being used. For example, if JFS were integrated with a symbolic execution engine, seeds encoding knowledge about feasible paths (and thus feasible inputs) could be communicated from the symbolic execution engine to JFS. We also envisage several improvements to the fuzzing component of JFS: designing mutators tailored to the context of SMT formulas would likely be beneficial; the fuzzer could be made aware of data flow, using information about the bytes that caused a constraint to become satisfied to guide mutations; and candidates for mutation could be prioritised according to the number of constraints they satisfy, which we hypothesise would lead to faster synthesis of satisfying assignments.

ACKNOWLEDGEMENTS

This research was generously sponsored by the UK EPSRC through grants EP/N007166/1, EP/P010040/1 and EP/R006865/1.

REFERENCES

[1] Merav Aharoni, Sigal Asaf, Laurent Fournier, Anatoly Koyfman, and Raviv Nagel. 2003. FPgen - a test generation framework for datapath floating-point verification. In *Eighth IEEE International High-Level Design Validation and Test Workshop 2003, San Francisco, CA, USA, November 12-14, 2003.* 17–22.

[2] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. 2010. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation. *IEEE Transactions on Software Engineering* 36, 6 (Nov. 2010), 742–762. https://doi.org/10.1109/TSE.2009.52

[3] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. 2013. Symbolic Path-Oriented Test Data Generation for Floating-Point Programs. In *Proc. of the IEEE International Conference on Software Testing, Verification, and Validation (ICST'13).*

[4] Vic Barnett. 2009. *Sample Survey Principles and Methods* (3 ed.). John Wiley & Sons, Chapter 4.

[5] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. 2013. Automatic detection of floating-point exceptions. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013.* 549–560.

[6] Clark Barrett, Haniel Barbosa, Martin Brain, Duligur Ibeling, Tim King, Paul Meng, Aina Niemetz, Andres Nötzli, Mathias Preiner, Andrew Reynolds, and Cesare Tinelli. 2018. CVC4 at the SMT Competition 2018. *CoRR* abs/1806.08775 (2018). arXiv:1806.08775

[7] Clark Barrett, Christopher Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11).*

[8] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2015. *The SMT-LIB Standard: Version 2.5.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[9] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2017. goSAT: Floating-point Satisfiability as Global Optimization. In *Proceedings of Formal Methods in Computer-Aided Design (FMCAD'17).* 11–14. https://doi.org/10.23919/FMCAD.2017.8102235

[10] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03).* ACM, New York, NY, USA, 196–207. https://doi.org/10.1145/781131.781153

[11] Mateus Borges, Marcelo d'Amorim, Saswat Anand, David Bushnell, and Corina S. Pasareanu. 2012. Symbolic Execution with Interval Solving and Meta-heuristic Search. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12).* IEEE Computer Society, Washington, DC, USA, 111–120. https://doi.org/10.1109/ICST.2012.91

[12] Bernard Botella, Arnaud Gotlieb, and Claude Michel. 2006. Symbolic Execution of Floating-point Computations. *Softw. Test. Verif. Reliab.* 16, 2 (June 2006), 97–121. https://doi.org/10.1002/stvr.v16:2

[13] Bruno Marre and FranÃğois Bobot and Zakaria Chihani. 2017. Real Behavior of Floating Point Numbers. In *Proc. of the 15th International Workshop on Satisfiability Modulo Theories (SMT'17).* http://smt-workshop.cs.uiowa.edu/2017/papers/SMT2017_paper_21.pdf

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08).*

[15] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the Association for Computing Machinery (CACM)* 56, 2 (2013), 82–90.

[16] Wei-Fan Chiang, Mark Baranowski, Ian Briggs, Alexey Solovyev, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2017. Rigorous Floating-point Mixed-precision Tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 300–315. https://doi.org/10.1145/3009837.3009846

[17] Alessandro Cimatti, Alberto Griggio, Bastiaan Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of TACAS (LNCS)*, Nir Piterman and Scott Smolka (Eds.), Vol. 7795. Springer.

[18] Edmund Clarke and Daniel Kroening. 2003. Hardware Verification using ANSI-C Programs as a Reference. In *Proc. of the 8th Asia and South Pacific Design Automation Conference (ASP-DAC'03).*

[19] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. A Tool for Checking ANSI-C Programs. In *Proc. of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'04).*

[20] Hélène Collavizza, Claude Michel, Olivier Ponsini, and Michel Rueher. 2014. Generating test cases inside suspicious intervals for floating-point number programs. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, May 31, 2014.* 7–11.

[21] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Crosschecking of Floating-Point and SIMD Code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11).*

[22] Peter Collingbourne, Cristian Cadar, and Paul H.J. Kelly. 2011. Symbolic Testing of OpenCL Code. In *Proc. of the Haifa Verification Conference (HVC'11).*

[23] Marc Daumas, Laurence Rideau, and Laurent Théry. 2001. A Generic Library for Floating-Point Numbers and Its Application to Exact Computing. In *Theorem Proving in Higher Order Logics*, Richard J. Boulton and Paul B. Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–184.

[24] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08).*

[25] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: introduction and applications. *Communications of the Association for Computing Machinery (CACM)* 54, 9 (Sept. 2011), 69–77.

[26] Zhoulai Fu and Zhendong Su. 2016. *XSat: A Fast Floating-Point Satisfiability Solver.* Springer International Publishing, Cham, 187–209. https://doi.org/10.1007/978-3-319-41540-6_11

[27] Zhoulai Fu and Zhendong Su. 2017. Achieving High Coverage for Floating-point Code via Unconstrained Programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).* ACM, New York, NY, USA, 306–319. https://doi.org/10.1145/3062341.3062383

[28] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05).*

[29] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08).*

[30] Yijia Gu, Thomas Wahl, Mahsa Bayati, and Miriam Leeser. 2015. Behavioral Non-portability in Scientific Numeric Computing. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings.* 558–569.

[31] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'11).*

[32] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[33] J.H. Holland. 1975. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* University of Michigan Press.

[34] IEEE 754-2008 2008. *IEEE Standard for Floating-Point Arithmetic.* Standard. Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/IEEESTD.2008.4610935

[35] Kiran Lakhotia, Nikolai Tillmann, Mark Harman, and Jonathan de Halleux. 2010. FloPSy - Search-Based Floating Point Constraint Solving for Symbolic Execution. In *Testing Software and Systems: 22nd IFIP WG 6.1 International Conference, ICTSS 2010, Natal, Brazil, November 8-10, 2010. Proceedings*, Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 142–157. https://doi.org/10.1007/978-3-642-16573-3_11

[36] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04).*

[37] Wen-Chuan Lee, Tao Bao, Yunhui Zheng, Xiangyu Zhang, Keval Vora, and Rajiv Gupta. 2015. RAIVE: Runtime Assessment of Floating-point Instability by Vectorization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015).* ACM, New York, NY, USA, 623–638. https://doi.org/10.1145/2814270.2814299

[38] M. Leeser, S. Mukherjee, J. Ramachandran, and T. Wahl. 2014. Make it real: Effective floating-point reasoning via exact arithmetic. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE).* 1–4. https://doi.org/10.7873/DATE.2014.130

[39] K. Rustan M. Leino. 2009. Dafny: An Automatic Program Verifier for Functional Correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'10).* 348–370.

[40] libfuzzer [n.d.]. LibFuzzer. http://llvm.org/docs/LibFuzzer.html.

[41] libfuzzerbugs [n.d.]. LibFuzzer trophies. http://llvm.org/docs/LibFuzzer.html#trophies.

[42] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair Donaldson, Rafael ZÃďhl, and Klaus Wehrle. 2017. Floating-Point Symbolic Execution: A Case Study in N-version Programming. In *Proc. of the 32nd IEEE International Conference on Automated Software Engineering (ASE'17).*

[43] David Menendez, Santosh Nagarakatte, and Aarti Gupta. 2016. *Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM.* Springer Berlin Heidelberg, Berlin, Heidelberg, 317–337. https://doi.org/10.1007/978-3-662-53413-7_16

[44] Michal Zalewski. [n.d.]. AFL "bug-o-rama" trophy case. http://lcamtuf.coredump.cx/afl/#bugs.

[45] Michal Zalewski. [n.d.]. Technical "whitepaper" for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[46] C. Michel, M. Rueher, and Y. Lebbah. 2001. Solving Constraints over Floating-Point Numbers. In *Principles and Practice of Constraint Programming — CP 2001*, Toby Walsh (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 524–538.

[47] Andres Nötzli and Fraser Brown. 2016. LifeJacket: Verifying Precise Floating-point Optimizations in LLVM. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP 2016)*. ACM, New York, NY, USA, 24–29. https://doi.org/10.1145/2931021.2931024

[48] Hristina Palikareva and Cristian Cadar. 2013. Multi-solver Support in Symbolic Execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*. http://srg.doc.ic.ac.uk/files/papers/klee-multisolver-cav-13.pdf

[49] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. 2015. Automatically Improving Accuracy for Floating Point Expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2737924.2737959

[50] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. 2013. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 20, 3 (01 Sept. 2013), 391–425.

[51] Jan Peleska, Elena Vorobev, and Florian Lapschies. 2011. Automated Test Case Generation with SMT-Solving and Abstract Interpretation. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 298–312.

[52] Sylvie Putot, Eric Goubault, and Matthieu Martel. 2003. Static Analysis-Based Validation of Floating-Point Computations. In *Numerical Software with Result Verification, International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 19-24, 2003, Revised Papers*. 306–313.

[53] Minghui Quan. 2016. Hotspot Symbolic Execution of Floating-Point Programs. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 1112–1114. https://doi.org/10.1145/2950290.2983966

[54] Jaideep Ramachandran, Corina S. Pasareanu, and Thomas Wahl. 2015. Symbolic Execution for Checking the Accuracy of Floating-Point Programs. *ACM SIGSOFT Software Engineering Notes* 40, 1 (2015), 1–5.

[55] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H. Bailey, and David Hough. 2016. Floating-point Precision Tuning Using Blame Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 1074–1085. https://doi.org/10.1145/2884781.2884850

[56] C. Rubio-GonzÃąlez, Cuong Nguyen, Hong Diep Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–12. https://doi.org/10.1145/2503210.2503296

[57] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*.

[58] SMT-COMP Competition 2006 2006. SMT-COMP Competition 2006. http://smtcomp.sourceforge.net/2006/.

[59] SMT-LIB. 2018. QF_BV benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV.git, revision `f7e691bf`.

[60] SMT-LIB. 2018. QF_BV_FP benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BVFP.git, revision `57d0c730`.

[61] SMT-LIB. 2018. QF_FP benchmarks. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_FP.git, revision `3346ad7a`.

[62] Matheus Souza, Mateus Borges, Marcelo d'Amorim, and Corina S. Păsăreanu. 2011. CORAL: Solving Complex Constraints for Symbolic Pathfinder. In *Proceedings of the Third International Conference on NASA Formal Methods (NFM'11)*. Springer-Verlag, Berlin, Heidelberg, 359–374. http://dl.acm.org/citation.cfm?id=1986308.1986337

[63] Mitsuo Takaki, Diego Cavalcanti, Rohit Gheyi, Juliano Iyoda, Marcelo d'Amorim, and Ricardo B. C. Prudêncio. 2010. Randomized constraint solvers: a comparative study. *Innovations in Systems and Software Engineering* 6, 3 (01 Sept. 2010), 243–253. https://doi.org/10.1007/s11334-010-0124-1

[64] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*.

[65] G. S. Tseytin. 1970. On the complexity of derivation in propositional calculus. *Constructive Mathematics and Mathematical Logic* (1970), 115–125.

[66] Xi Wang, Nickolai Zeldovich, Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proc. of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.