Scalable SMT Sampling for Floating-Point Formulas via Coverage-Guided Fuzzing

Manuel Carrasco
Imperial College London
London, United Kingdom
m.carrasco@imperial.ac.uk

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

Alastair F. Donaldson
Imperial College London
London, United Kingdom
alastair.donaldson@imperial.ac.uk

***Abstract*—SMT sampling involves finding numerous satisfying assignments (samples), for an SMT formula, and is increasingly finding applications in software testing. An effective SMT sampler should achieve high *throughput*, yielding a large number of samples in a given time budget, and high *diversity*, yielding samples that cover disparate parts of the solution space.**

Most SMT samplers rely on off-the-shelf SMT solvers and thus inherit those solvers' scalability issues. Because SMT solvers tend to scale poorly when applied to floating-point constraints, the scalability and diversity of SMT sampling is correspondingly limited in the floating-point domain.

We propose JFSAMPLER, the first SMT sampling technique built on top of coverage-guided fuzzing. JFSAMPLER extends Just Fuzz-it Solver (JFS), a scalable but incomplete SMT solver that is effective at finding solutions to floating-point formulas by encoding satisfiability as a reachability problem that is then offloaded to a fuzzer. By building on JFS, JFSAMPLER has an advantage over other SMT samplers in the floating-point domain. Further, we propose two novel strategies that increase both throughput and diversity of sampled solutions. First, JFSAMPLER enhances the fuzzer's code coverage feedback signal by measuring coverage of the formula's solution space. Second, JFSAMPLER incorporates a custom mutator tailored for SMT sampling. By design, these two novel techniques can be combined, having a positive synergy on throughput and diversity.

We present a large evaluation over QF_FP and QF_BVFP formulas from the SMT-LIB benchmark. Our results show that JFSAMPLER achieves substantial improvements over SMTSAMPLER, a state-of-the-art SMT sampling technique, when applied to floating-point formulas.

***Index Terms*—fuzzing, SMT, sampling.**

I. INTRODUCTION

Various software testing techniques leverage SMT solvers to produce inputs that meet specific constraints: if the constraints can be satisfied, the solver provides a satisfying assignment that can be used as a new input for testing. A prime example is symbolic execution [1]–[4], where constraints arising from the code of the software under test are gathered and solved to find new inputs that cover previously-unexplored paths. Specifically, symbolic execution uses an SMT solver to find an input that corresponds to an execution path *prefix*.

SMT-based testing techniques are subject to two key limitations of typical off-the-shelf SMT solvers. First, most off-the-shelf solvers provide only one satisfying assignment for a satisfiable set of constraints. Second, the difficulty of SMT solving leads to scalability problems where solvers time out, so that no satisfying assignment is provided even if the constraints are actually satisfiable. Regarding the first limitation: traditional symbolic execution techniques only expect one satisfying assignment for each path prefix's constraints, although multiple inputs in the same path prefix can take different paths afterwards and unlock new coverage. 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 [5].

To overcome this limitation, researchers have been working on techniques which find large sets of satisfying assignments and that provide reasonable coverage of the formula's solution space. This field of research is referred to as *SMT sampling*. The benefits of SMT sampling have started to gain attention in the field of software testing [5]–[8].

The second limitation—scalability—leads to solver timeouts, in which case *no* new input is generated. This may prevent critical bugs from being revealed. Scalability is particularly problematic for the floating-point SMT theory (QF_FP). As a result, symbolic execution implementations offer limited or no support for floating-point arithmetic [9]–[11].

Various methods have been investigated to improve the scalability of floating-point SMT solving. These methods can be categorized into *complete* and *incomplete* SMT solvers. Complete solvers [12]–[15] can determine whether the formula is satisfiable or not (assuming no timeouts). On the other hand, incomplete solvers [16]–[19] use heuristics that can only prove a formula is satisfiable but cannot determine if it is unsatisfiable.

Widely-used complete solvers such as Z3 [14] support floating-point constraints by translating them into bitvector constraints and solving them using a SAT solver [20]–[22]. However, this approach does not scale well because floating-point expressions are translated into very large, complex Boolean circuits that are difficult for the underlying SAT solver to reason about [22], [23]. In contrast, JFS [16], an incomplete SMT solver, has been shown to yield competitive and complementary results when compared with a wide range of complete and incomplete solvers in the floating-point domain. The key idea behind JFS is to leverage *coverage-guided fuzzing* [24] to solve constraints: an SMT formula is turned into a C++ program whose inputs correspond to free variables of the formula, and that contains a special error location that is reachable if and only if the inputs comprise a satisfying assignment to the formula. Coverage-guided fuzzing can then

be used to find bugs in this program, where finding a bug amounts to finding a satisfying assignment.

Although JFS has been shown to be successful in improving the scalability of floating-point reasoning, it does not address the limitation of not being able to generate multiple solutions: JFS returns a *single* satisfying assignment. It thus cannot aid in software testing problems requiring SMT *sampling*.

Our contribution. In this work, we focus on the problem of achieving *scalable SMT sampling* in the domain of floating-point arithmetic, contributing to addressing both of the above limitations. We propose a new technique, called JFSAMPLER, for SMT sampling, which is more scalable than the current state-of-the-art technique, SMTSAMPLER [25], when dealing with floating-point constraints.

Our JFSAMPLER technique lifts the coverage-guided fuzzing approach of JFS from the domain of SMT solving to SMT sampling. We draw on our insight that the incomplete nature of JFS is unproblematic in the context of SMT sampling, where the assumption is that the formula under analysis *is* satisfiable and the aim is to extract a large number of diverse satisfying assignments. Our key innovations in JFSAMPLER are (a) a novel program encoding that plays to the strengths of coverage-guided fuzzing, and (b) a custom mutation strategy inspired by SMTSAMPLER that equips the underlying coverage-guided fuzzer with a mutation operator explicitly tailored to meet the needs of SMT sampling.

We validate our contributions using formulas from the extensive SMT-LIB benchmark [26] that target the QF_FP (quantifier-free formulas over floating-point arithmetic) and QF_BVFP (quantifier-free formulas over bitvector and floating-point arithmetic) theories. This benchmark has a wide range of constraints, including (but not limited to) constraints arising from various software testing and verification techniques, such as symbolic execution, bounded model checking, and static analysis. In this way, we aim to push the advancement of scalable SMT-sampling-based testing methods for software with floating-point operations.

In our evaluation, we compare JFSAMPLER to two baselines. The first baseline is SMTSAMPLER [25], which is based on the SAT sampling technique QUICKSAMPLER [27]. SMTSAMPLER has previously shown outstanding scalability compared to other samplers. Additionally, SMTSAMPLER has shown competitiveness with respect to *SMT coverage*, a metric designed to compute how well the generated samples cover the solution space. The second baseline is an SMT sampler based on the original JFS tool that does not use our novel encoding and mutation strategies, allowing us to validate the effectiveness of these strategies.

A large experimental evaluation against these baselines shows that JFSAMPLER excels in the QF_FP domain. First, the ability of JFSAMPLER to quickly solve many floating-point queries, thanks to its use of coverage-guided fuzzing, makes the technique practical for SMT sampling where SMTSAMPLER is often inapplicable (because the traditional SMT solver that backs SMTSAMPLER cannot find any satisfying assignments). Secondly, our novel encoding and mutation strategy dramatically enhance the throughout and diversity of sampled formulas compared with the naive JFS-based baseline. In the QF_BVFP domain, our strategies allow JFSAMPLER to dramatically reduce an initial performance gap with respect to SMTSAMPLER and win by a tight margin. Our experiments evaluate a total of 862 formulas from the SMT-LIB benchmark in the QF_FP and QF_BVFP suites and required 212 CPU days of compute time to complete.

In summary, our contributions are as follow:

1) The first coverage-guided SMT sampler for floating-point constraints.
2) A novel program encoding for fuzzing which increases sampling throughput and the solution space coverage.
3) A coverage-guided custom mutator tailored for the SMT sampling domain and designed to generate new satisfying assignments with high probability.
4) The implementation of these ideas in a new SMT sampling tool, JFSAMPLER, and an evaluation across 862 QF_FP and QF_BVFP formulas from the SMT-LIB benchmark, showing a superior performance compared to the state-of-the-art SMTSAMPLER technique.

The rest of the paper is structured as follows. We introduce the necessary background in §II. Then, we explain the design of JFSAMPLER, which includes our key contributions, in §III. Next, we evaluate and compare our contributions to the baselines in §IV. Lastly, we present related work in §V and conclude in §VI.

II. BACKGROUND

A. Coverage-guided Fuzzing

Coverage-guided fuzzing (also known as greybox fuzzing) aims to find bugs by randomly mutating and combining inputs drawn from a corpus and feeding them to a program, with the aim of finding inputs that cause the program to crash. The fuzzer instruments the program to track edge code coverage. When an input is found to achieve new coverage, it is added to the corpus for further mutation. The idea is that prioritising inputs that were found to achieve new coverage for further mutation and combination may guide the fuzzing process towards bug-triggering inputs. Popular coverage-guided fuzzers include LIBFUZZER [24] (used in this work) and AFL [28].

B. Just Fuzz-it Solver (JFS)

As discussed in the introduction, satisfiability modulo theories (SMT) solvers underpin many formal testing methods, and limitations of SMT solvers correspondingly limit the effectiveness of testing. Solvers often perform so poorly when dealing with *floating-point* constraints that SMT-based testing methods become impractical for analysing software operating on floating-point numbers [29].

Just Fuzz-it Solver (JFS) [16] is an incomplete SMT solver for the theories of bitvectors (QF_BV), floating point (QF_FP) and their combination (QF_BVFP). It is built on top of the coverage-guided fuzzer LIBFUZZER [24] (see §II-A). JFS can only prove that a given formula is satisfiable but not that

```
1  ; Declare floating-point variable x
2  (declare-fun x () (_ FloatingPoint 8 24))
3  (assert
4   (let
5     ((?x9 ; Alias for (x + 0)
6       (fp.add RNE
7         x (_ +zero 8 24))))
8   ; Assert (x + 0) >= x
9   (fp.geq ?x9 x)))
10 (check-sat)
```

Listing 1: Example QF_FP formula.

```
1  int LLVMFuzzerTestOneInput(uint8_t* data)
2  {
3    Float32 x = makeFloatFrom(data, 0, 31);
4    Float32 ssa0 = getPositiveZero();
5    Float32 ssa1 = ssa0.add(RM_RNE, x);
6    bool ssa2 = ssa1.fpgeq(x);
7    if (!ssa2) return 0;
8    abort();
9  }
```

Listing 2: C++ program generated by JFS for Listing 1.

it is unsatisfiable. In its evaluation, JFS is competitive and complements off-the-shelf SMT solvers in the floating-point domain (but not effective in the bitvector domain).

JFS requires that the input formula is presented as a conjunction of assertions (the SMT-LIB format guarantees it). 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.

To illustrate this, consider the simple formula of Listing 1. In SMT-LIB notation, this formula represents the constraint $(x + 0) \geq x$. First, a floating-point free variable `x` is declared with an exponent and mantissa of 8 and 24 bits, respectively. Then `x9` is declared as an alias for an addition expression `fp.add`. The addition expression takes as operands the rounding mode *RNE* (round to nearest, with ties to even [30]), the free variable *x*, and `(_ +zero 8 24)`, which represents the constant value positive zero.

Listing 2 shows the C++ program that JFS generates for the formula of Listing 1. The program is a fuzz target for LIBFUZZER, and some details are omitted for conciseness, such as the included headers (defining types and functions) and the `data` buffer's size. The fuzzing target receives the input byte buffer `data` (line 1), which the fuzzer fully controls. JFS uses the buffer to map its bits to the ones in the free variables occurring in the SMT formula; the buffer's bits correspond to a full candidate assignment's bits. JFS parses the buffer to instantiate the floating-point free-variable `x` that matches the semantics of SMT-LIB, in the process checking that the buffer size is correct (line 3).[1] This corresponds to the declaration of free variable `x` at line 2 of Listing 1. The floating-point positive zero is constructed (line 4) and added to `x` (line 5); these steps correspond to lines 6–7 of Listing 1.

Finally, the program performs a comparison matching the input constraint (line 6). If the comparison returns true, the current assignment satisfies the constraint, and LIBFUZZER is signaled with an abort; JFS terminates and returns SAT. In other words, if the current input satifies the constraints, the program crashes and the fuzzer saves the byte buffer as a crash-inducing test case. Otherwise, the fuzzer continues testing. If no crashing input is found within a given time limit, JFS returns an *unknown* result.

Before compiling the program, JFS applies fuzzing-friendly optimisations to boost effectiveness, such as lifting equalities. Since coverage-guided fuzzing needs an input corpus, JFS provides *smart seeds* featuring special constant values (e.g., infinities and zeros) [16].

This illustrative example is simple; a larger example with more constraints would contain more branching logic, providing the coverage-guided fuzzer with numerous coverage points to emit a feedback signal for fuzzing.

C. SMTSampler

SMTSAMPLER [25] is a scalable *non-uniform* SMT sampler; not all satisfying assignments are equally likely to be generated. SMTSAMPLER has been shown to outperform other samplers [27], [31], [32] by orders of magnitude in throughput. It can handle formulas in the quantifier-free theories of bitvectors, arrays, and uninterpreted functions (QF_AUFBV). The non-uniform nature of SMTSAMPLER also applies to JFSAMPLER.

As the state-of-the-art non-uniform SMT sampler, we chose SMTSAMPLER as a baseline for our evaluation. We straightforwardly adapted its implementation for floating-point constraints without altering its algorithm (see §IV-B1).

In short, SMTSAMPLER uses a MAX-SMT solver [33] to find random satisfying assignments (*samples*) and applies a heuristic to mine additional samples without calling the solver, improving scalability. Each iteration of this process, called an *epoch*, repeats until a time limit or sample count is reached.

We now discuss the technique in more detail, assuming for ease of presentation that all free variables are booleans.

An epoch consists of three sample-generation steps. The first step computes a *base solution*, a randomly generated satisfying assignment, using a MAX-SMT solver. A MAX-SMT solver finds an assignment that satisfies a set of *hard constraint* and maximises the satisfiability of a set of *soft constraints*. SMTSAMPLER creates a MAX-SMT query in which the input formula is a hard constraint, and constraints assigning the free variables of the formula to random values are soft constraints. Soft constraints are important because without

[1] `Float32` is defined by JFS following the SMT-LIB specification.

them the deterministic behaviour of solvers could undermine the diversity of the satisfying assignments.

The second step uses the base solution as a string of bits; all assignments for the input formula have the same length in bits (a property that is guaranteed by the supported theories). Each base solution's bit, corresponding to a variable, is tested to determine whether it can be flipped while minimising the number of extra flips required by additional variables. This is encoded as a MAX-SMT query with the input formula and the bit flip as two hard constraints; the values of the remaining bits are each set as soft constraints. The found solutions are called *atomic mutations* because they are neighbouring solutions of the base solution.

Finally, once the atomic mutations are available, SMTSAMPLER uses a simple but effective heuristic that combines the assignments at the bit level. Two atomic mutations A^i and A^j of the base solution B are combined in the following way to produce a new assignment N. For A^i and A^j, a *diff patch* is computed with respect to B by using the bit-wise XOR operator: $A^{\mathrm{i}}{}_{diff} = A^{\mathrm{i}} \oplus B$. These two patches are combined using the bit-wise OR operator, and applied to B to yield N: $N = B \oplus (A^{\mathrm{i}}{}_{diff} \vee A^{\mathrm{j}}{}_{diff})$. Every assignment generated by the heuristic is tested for satisfiability because it does not come from a solver; satisfiability can be checked by just evaluating the formula with the assignment.

This heuristic is effective because it cheaply generates potential satisfying assignments, compared to off-the-shelf solvers, and has a high success rate in practice. By flipping only differing bits between the two satisfying assignments, it often produces a new solution since those bits can vary without violating the formula.

The heuristic is first applied to all atomic mutation pairs, generating a new set. This process is repeated five times on each subsequent set (operating only on satisfying assignments), yielding an exponential number of solutions based on the atomic mutations. This step is crucial for scalability as it avoids constraint solving. After completion, SMTSAMPLER proceeds to the next epoch.

SMTSAMPLER only outputs unique satisfying assignments. To ensure uniqueness, assignments are represented as strings of bits of equal length. These strings are then hashed into a set to avoid duplicates.

D. SMT Coverage Metric

SMTSAMPLER [25] has shown great scalability at the expense of no uniformity guarantees (solutions do not have the same probability of being generated). Therefore, the authors proposed an SMT coverage metric to assess the samples' diversity with respect to the formula evaluation. The input QF_AUFBV formula can be considered as an abstract syntax tree, where the internal nodes represent boolean, bitvector or floating-point sub-expressions to be evaluated. Each bit of each internal node is tracked when evaluating all samples. The coverage metric is incremented by one for each bit which is assigned its two possible values across all generated samples.

To illustrate, consider the SMT formula of Listing 1 representing the constraint $(x + 0) \geq x$. The SMT coverage metric will consider its unique internal node that represents the addition, typed as a 32-bit floating-point expression. When evaluating the constraints with an assignment for x, the addition sub-expression $(x + 0)$ will evaluate to x.

Let us assume that the following satisfying assignments for x are considered for computing the SMT coverage: *positive zero*, *negative zero* and -2. The binary representation for their first byte is as follows: `00000000`, `10000000` and `11000000` respectively (the remaining bytes are omitted because they are all zeroes).

The metric starts by evaluating the first assignment for x, which is the positive zero. The coverage computation keeps track that all the addition expression's bits were evaluated to zero, and not one (the possible values for a bit). At this point, the SMT coverage metric is zero because the expression's bits were only evaluated with one possible value.

Next, when evaluating the following satisfying assignment, negative zero, all bits in the expression are evaluated as before except the one corresponding to the sign bit. Consequently, the SMT coverage metric is increased by one, because one bit, the sign bit, was evaluated with its two possible values.

Finally, when -2 is evaluated, all bits are evaluated to zero except the sign bit and one bit in the mantissa. The sign bit has saturated in coverage, but the one in the mantissa has not, and as a result the coverage metric is now increased again by one. The total SMT coverage for these assignments is 6.25% ($\frac{\textit{covered bits}=2}{\textit{total bits}=32} * 100$).

III. Design of JFSampler

We now present the design of JFSAMPLER, an SMT sampler for the QF_FP and QF_BVFP theories. From a baseline where JFS is naively turned into an SMT sampler via repeated fuzzing target runs (§III-A), we explain the two novel innovations that make JFSAMPLER scale well: a diversity encoding based on SMT coverage (§III-B), and a custom mutator based on a heuristic used in SMTSAMPLER for combining existing satisfying assignments (§III-C).

A. Naive Extension of JFS for Sampling

As explained in §II-B, the JFS solver halts fuzzing and reports SAT as soon as LIBFUZZER discovers a crashing input (abort) in the C++ encoding of the input formula. To turn JFS into a *naive* SMT sampler, one can simply change JFS so that instead of halting when a crash is discovered, it saves the crashing input but carries on fuzzing in search of further crashing inputs. We refer to this simple extension of JFS as JFSAMPLER$^{\text{Naive}}$, which is a key baseline for comparing our more sophisticated techniques proposed in §III-B and §III-C.

The JFSAMPLER$^{\text{Naive}}$ approach is likely to have only limited success because the standard JFS encoding of a formula (which was not designed with SMT sampling in mind) does nothing to reward the fuzzer for synthesising *successive* inputs that crash the fuzzing target (and thus satisfy the formula). To illustrate this, consider again the program of Listing 2. Once

an input that reaches the `abort()` and an input that does not reach the `abort()` have been synthesised, the fuzzer will have fully covered the program code.[2] As a result, coverage-guided fuzzing will in fact no longer be guided by coverage, because fuzzing will now be no more effective than un-guided black box fuzzing. Later, we explain how we solve this limitation via a novel encoding and a custom mutator.

Implementation. The function `abort()` in Listing 2 is a standard library function that triggers an abort signal and does not return. We replaced it by `raise(SIGABRT)` to allow it to return, and we modified LIBFUZZER's signal handling to save the crashing input and continue the fuzzing loop. To keep track of the number of generated samples, we hash and count each new input that reaches `raise(SIGABRT)`.

B. Diversity Encoding

To enable sampling with better diversity and throughput compared with JFSAMPLER$^{\text{Naive}}$, we propose a new method for encoding an SMT formula as a program-to-be-fuzzed. This new *diversity encoding* (DE) is based on the SMT coverage metric proposed in prior work on SMTSAMPLER and described in §II-D. In our evaluation, we refer to the extension of JFSAMPLER$^{\text{Naive}}$ with this encoding as JFSAMPLER$^{\text{DE}}$.

Our new encoding inserts additional semantic-preserving program code just before the `abort()` statement, i.e. code that will only be reached when a satisfying assignment has been found. Following the SMT coverage metric, the additional code tests whether each bit of the input formula's sub-expressions is enabled. From the perspective of coverage-guided fuzzing, these tests yield new edges in the program's control flow graph that can be covered only by satisfying assignments. Different satisfying assignments will cover the additional code in different ways, depending on which sub-expression bits are enabled. Since edge coverage is the reward mechanism for the coverage-guided fuzzer, by encoding SMT coverage as program code coverage, the fuzzer will be rewarded for finding diverse satisfying assignments that achieve new SMT coverage.

Listing 3 illustrates the JFSAMPLER$^{\text{DE}}$ encoding. The macro `DIVERSIFY_FLOAT(expr)` inlines the bit-level tests on the input floating-point expression. For reasons of space, the example only shows a formula that requires diversifying a floating-point expression. The new edges introduced by this macro's encapsulated tests give the fuzzer feedback on code coverage, rewarding satisfying assignments that explore different values in `expr`'s bits.

Implementation. The new encoding is built on top of three C/C++ macros: `DIVERSIFY_BOOL(expr)`, `DIVERSIFY_BV(expr)` and `DIVERSIFY_FLOAT(expr)`. These macros incorporate a check for each bit in the input expression, determining whether the target bit is set.

Listing 3 demonstrates the implementation of `DIVERSIFY_FLOAT(expr)` for 32-bit floats. For ease of presentation we have slighly simplified the code; for example line continuation characters for multi-line macros are omitted. The macro uses the bit-level representation of the float value, inspecting each byte bit by bit, which adds two new edges at the inlining point for each bit check. Optimisations are disabled during compilation (the default in JFS), ensuring these checks remain intact. The same approach applies to other data types, as we also operate at their bit-level representation.

```
1  #define TEST_BIT(BYTE, POS) {
2    if ((BYTE & (1 << POS)) != 0) {}
3  }
4
5  #define DIVERSIFY_FLOAT(F) {
6    uint32_t bits = F.getRawBits();
7    uint8_t* buffer = (uint8_t*)&bits;
8    // Test each bit in each byte
9    uint8_t byte = buffer[0];
10   TEST_BIT(byte, 0) TEST_BIT(byte, 1)
11   TEST_BIT(byte, 2) TEST_BIT(byte, 3)
12   TEST_BIT(byte, 4) TEST_BIT(byte, 5)
13   TEST_BIT(byte, 6) TEST_BIT(byte, 7)
14   // Idem for buffer[1 <= i <= 3]
15 }
16
17 int LLVMFuzzerTestOneInput(uint8_t* data)
18 {
19   Float32 x = makeFloatFrom(data, 0, 31);
20   Float32 ssa0 = getPositiveZero();
21   Float32 ssa1 = ssa0.add(RM_RNE, x);
22   bool ssa2 = ssa1.fpgeq(x);
23   if (!ssa2) return 0;
24   DIVERSIFY_FLOAT(ssa1);
25   abort();
26 }
```

Listing 3: New diversity encoding (§III-B) for Listing 1.

In pilot experiments, the overhead while enabling these runtime checks in full led to an unacceptable decrease in fuzzing throughput. To control the overhead, `DIVERSIFY_FLOAT(expr)` and `DIVERSIFY_BV(expr)` are guarded with a runtime-random check that triggers them with 20% probability, which we found to provide a good balance in our pilot experiments;[3] for brevity the guards are omitted in Listing 3. Therefore, JFSAMPLER$^{\text{DE}}$ generates a C++ program diversifying *all* sub-expressions, but at runtime, their execution is randomly triggered each time the fuzzer tests an input corresponding to a satisfying assignment. We found this to be more effective than only selecting a fraction of the sub-expressions at compilation time for diversification.

C. A Custom Mutator for SMT Sampling

Recall from §II-C that SMTSAMPLER uses a heuristic for combining existing satisfying assignments that has proven to be highly effective as a means of generating new satisfying

[2]The source code for the types defined by JFS is relatively simple. It offloads the corresponding operations to native types as much as possible, which is not enough to mitigate the *lack of coverage* issue.

[3]In our pilot experiments, the overhead was more noticeable in the QF_FP suite than in QF_BVFP because in practice floating-point sub-expressions are 32 or 64 bits, whereas bitvector ones are often smaller.

```
1  void SamplingMutator(uint8_t* Cur,
2                       uint8_t* Out){
3    // Cur is chosen by the fuzzer based on
         code-coverage feedback.
4    uint32_t Size = TestCasesSize();
5    uint8_t* First = RandPrevSAT();
6    uint8_t* Second = RandPrevSAT();
7
8    for (uint32_t i = 0; i < Size; i++){
9      Out[i] = (Cur[i] ^ First[i]);
10     Out[i] |= (Cur[i] ^ Second[i]);
11     Out[i] ^= Cur[i];
12   }
13 }
```

Listing 4: Implementation of the coverage-feedback driven SMT mutator inspired by SMTSAMPLER (§III-C).

assignments; we refer to this heuristic as SAMPLINGMUTATOR. This is the only way SMTSAMPLER attempts to find satisfying assignments without calling a constraint solver. It is worth noting that SMTSAMPLER applies SAMPLINGMUTATOR blindly and without any feedback on the satisfying assignments.

The fuzzer that underlies JFSAMPLER$^{\text{Naive}}$ is LIBFUZZER, and LIBFUZZER performs byte-level mutations on a corpus of inputs to achieve new code coverage. We equip LIBFUZZER with the SAMPLINGMUTATOR heuristic as a new custom mutator; we refer to this version as JFSAMPLER$^{\text{SM}}$.

We improved and relaxed how the SAMPLINGMUTATOR heuristic is applied in JFSAMPLER$^{\text{SM}}$. Unlike SMTSAMPLER, which only combines satisfying assignments, in JFSAMPLER$^{\text{SM}}$, SAMPLINGMUTATOR is applied to any assignment that achieves new coverage (whether satisfying or not). In this case, the assignment that has achieved new coverage is combined with two randomly-picked satisfying assignments from earlier in the fuzzing run. Our hypothesis is that applying this mutator to all assignments that gain new coverage will improve the diversity of fuzzing. We decided to *randomly* select the two other satisfying assignments to keep our implementation simple and efficient. Any other criteria would have required a more expensive implementation, which would damage the throughput of fuzzing.

In summary, the above design improves JFSAMPLER$^{\text{SM}}$ by generating new inputs via an effective heuristic for SMT constraints, and increases SMT coverage by leveraging code coverage feedback. JFSAMPLER$^{\text{SM}}$, with its custom mutator, is orthogonal to the diversity encoding used by JFSAMPLER$^{\text{DE}}$ (see §III-B)—these features can be used independently or combined.

Implementation. LIBFUZZER has a list of built-in mutators that are applied on the current feedback-driven input. We implemented SAMPLINGMUTATOR as an additional built-in mutator in LIBFUZZER.

Listing 4 shows the implementation of SAMPLINGMUTATOR in JFSAMPLER$^{\text{SM}}$. The mutator receives the `Cur` buffer, which is the currently picked assignment by the fuzzer based on the coverage feedback. Then, it randomly selects two satisfying assignments from an internal list (if there are not enough satisfying assignments, the mutator returns without yielding a new input). Finally, the mutator combines them at the byte level following the heuristic defined in SMTSAMPLER (§II-C); in JFSAMPLER all assignments have the same length in bytes. The fuzzing loop treats the output written in the `Out` buffer like any other mutator's result, and it is added to the corpus if and only if it triggers new code coverage.

IV. EVALUATION

We now compare JFSAMPLER and SMTSAMPLER in terms of throughput (number of generated samples within a given time budget) and diversity (SMT coverage). We aim to answer the following research questions (RQs):

RQ1 Does JFSAMPLER achieve higher throughput and diversity than the state-of-the-art SMTSAMPLER technique?

RQ2 Do the diversity encoding (§III-B) and SMT sampling mutator (§III-C) lead to higher throughput and diversity compared to the naive extension of JFS (§III-A)?

A. Benchmark Selection

JFS's evaluation [16] identified challenging satisfiable formulas from the SMT-LIB benchmark [26], resulting in two suites for the QF_FP and QF_BVFP logics, containing 160 and 702 formulas, respectively. Our experiments are conducted on these two suites.

B. SMTSampler Implementation

We use SMTSAMPLER [25] as a baseline SMT sampling tool that represents the state-of-the-art in SMT sampling. Our evaluation is based on commit `8186483` of the SMTSAMPLER GitHub repository [34]. We explain the minimal extensions we made to enable floating-point support in SMTSAMPLER, and key aspects about its underlying constraint solver:

1) Extension for QF_FP Support: SMTSAMPLER is designed for quantifier-free bitvector theories, specifically supporting the QF_AUFBV logic. IEEE-754 floating-point values are represented as bitvectors, so that in principle the SMTSAMPLER algorithm can process them as bitvector variables. However, in SMT-LIB, direct access to the bitvector representation of a floating-point value is not permitted; conversion must be explicit.

We minimally extended SMTSAMPLER by utilising Z3 API's `mk_to_ieee_bv(FloatExpr)` function, which returns the bitvector representation of a floating-point input expression as an SMT-LIB bitvector.

The input formula remains unchanged and is added to the constraint solver as before. When constructing a base solution, random bitvector constants are assigned to free variables as soft constraint equalities (as explained in §II-C); this process is adjusted for floating-point variables. A floating-point variable expression is wrapped with `mk_to_ieee_bv` for conversion to a correctly typed bitvector. Similarly, during the atomic mutation constraints generation, if a bit from a floating-point free variable is flipped, the variable expression is also wrapped with `mk_to_ieee_bv` to treat it as a bitvector.

2) Underlying Constraint Solver: The SMTSAMPLER implementation uses Z3 for MAX-SMT solving. We upgraded Z3 to commit fa2c0e027 from 2024; by default, it was using a commit from 2018.

In principle the SMTSAMPLER approach could work with any SMT solver that provides MAX-SMT support for relevant theories. However, the SMTSAMPLER implementation is tightly coupled to the Z3 API: this API is used for building expressions; SMTSAMPLER uses a complex binary encoding of satisfying assignments also coupled to Z3's in-memory models; and SMTSAMPLER relies on custom changes to Z3 to support its SMT coverage metric. For these reasons it was not possible for us to compare JFSAMPLER against the SMTSAMPLER's approach based on solvers other than Z3.

We also considered whether SMTSAMPLER could be configured with JFS as its underlying solver. However, in addition to the technical issues outlined above, more fundamentally the SMTSAMPLER algorithm requires a MAX-SMT solver, whereas JFS is only an incomplete SMT solver.

C. Samplers Evaluated

We compare SMTSAMPLER and various variants of JFSAMPLER variants, as follows:

1) SMTSAMPLER is the state-of-the-art baseline technique, extended with floating-point support (§II-C and §IV-B1).
2) JFSAMPLER$^{\text{Naive}}$ is our sampler in its simplest form (§III-A).
3) JFSAMPLER$^{\text{DE}}$ is JFSAMPLER$^{\text{Naive}}$ with our diversity encoding (§III-B).
4) JFSAMPLER$^{\text{SM}}$ is JFSAMPLER$^{\text{Naive}}$ with the SMT heuristic for combining inputs (§III-C).
5) JFSAMPLER$^{\text{SM+DE}}$ combines the features in JFSAMPLER$^{\text{DE}}$ and JFSAMPLER$^{\text{SM}}$.

D. Experimental Setup

Hardware setting: Experiments were conducted on a cluster of AMD EPYC 7742 64-Core machines. Each formula sampling process, whether using JFSAMPLER or SMTSAMPLER, is allocated to a single physical core, with each core assigned 8 GiB of RAM and 1TB of storage.

JFSAMPLER and SMTSAMPLER keep the generated samples in memory during the sampling time budget, and do not write to disk. We observed that the writing of samples to disk caused I/O bottlenecks due to the high level of parallelism achieved in the cluster. Therefore, we modified both tools to write the generated samples to disk only after the sampling time budget is exhausted.

Sampler configurations: The sampling time budget per formula is set to 5 min, and the maximum number of samples is unbounded. Sampling of each formula is repeated 10 times per sampler, using different RNG seeds.

JFSAMPLER (all variants) is configured using the JFS-LF-SS mode of JFS, which was the most successful mode in its evaluation [16]. SMTSAMPLER is configured in the SMTBIT mode, which led to the best coverage results [25].

Both tools are configured with a memory limit of 6 GiB of RAM. For JFSAMPLER this is configured in LIBFUZZER, while in SMTSAMPLER this is configured using the Z3 API. In case the limit is reached, both tools exit gracefully.

In order to calculate the SMT coverage metric (§II-D), we randomly select 10% of the total samples generated by each tool. We only choose 10% of the samples because computing the SMT coverage metric is computationally expensive. The metric requires evaluating the input formula in a Z3 model object (a satisfying assignment). The metric is hooked into the Z3 evaluation method of its models. Alongside the evaluation, which is already expensive [35], computing the coverage metric also requires testing each bit of each of the formula's sub-expressions. Due to the large number of generated models, this overhead can easily exceed the time spent in sampling if all samples are selected.

SMTSampler's solver timeout: SMTSAMPLER uses a MAX-SMT solver and requires a specified timeout value. Using too short or too long of a timeout could unfairly penalise SMTSAMPLER by spending too little or too much time in generating a base solution or an atomic mutation (see II-C).

To ensure a fair comparison, we profiled the SMTSAMPLER with various timeouts to identify the most competitive value. We randomly selected 30 formulas from each logic and executed the SMTSAMPLER following the previously outlined methodology. Timeout values ranged from 5 s to 295 s in 10-second increments, totalling 30 values; this profiling required 62 days of CPU time. The initial timeout of 5 s matches the one used in the SMTSAMPLER's evaluation [25].

We analysed the SMT coverage distribution per timeout using boxplots for both suites, which were all equal in size and height, with variations only in whiskers and outliers. Minor differences in the number of generated samples' distributions were noted in QF_BVFP. Thus, we determined the optimal timeout for SMTSAMPLER based on the generated samples. Boxplots are omitted for brevity and can be found in [36].

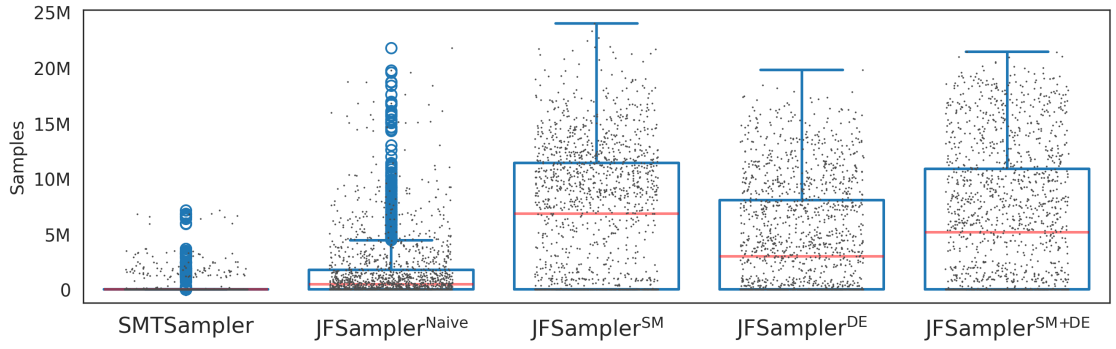
We selected the first timeout value (5 s) as the control experiment and computed the effect size (A^{12}) for each subsequent timeout (29 remaining values) relative to the control. The effect size indicates the probability that a non-control value scores higher than the control on a formula. We selected the timeout with the highest effect size greater than 0.5, ensuring it outperformed the control. For the QF_FP suite, the best timeout was 255 s, while for the QF_BVFP suite, it was 125 s.

E. Results

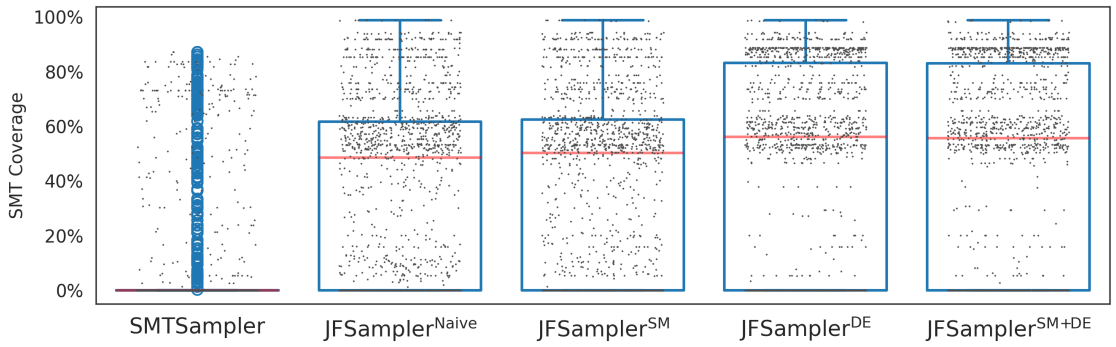
Figures 1 and 2 show the performance of different samplers on the QF_FP and QF_BVFP suites respectively, in terms of total samples generated and SMT coverage achieved in the sampling budget of 5 min.

The figures display boxplots together with a *strip plot* overlay that shows the distribution of all observations; black dots at the same height correspond to observations of the same magnitude.

Tables I and II present a statistical analysis comparing the techniques in terms of generated samples and SMT coverage, respectively. Following Arcuri and Briand's guidelines [37],



(a) Samples per formula in QF_FP.



(b) SMT coverage per formula in QF_FP.

Fig. 1: Total samples and SMT coverage distributions for the QF_FP suite.

TABLE I: Statistical analysis of the difference in the number of generated samples per formula: A^{12} effect size of technique T_A relative to technique T_B (i.e., probability that T_A generated more samples than T_B), and the corresponding *Winner* for statistically significant cases (Wilcoxon signed-rank test, $p < 0.001$).

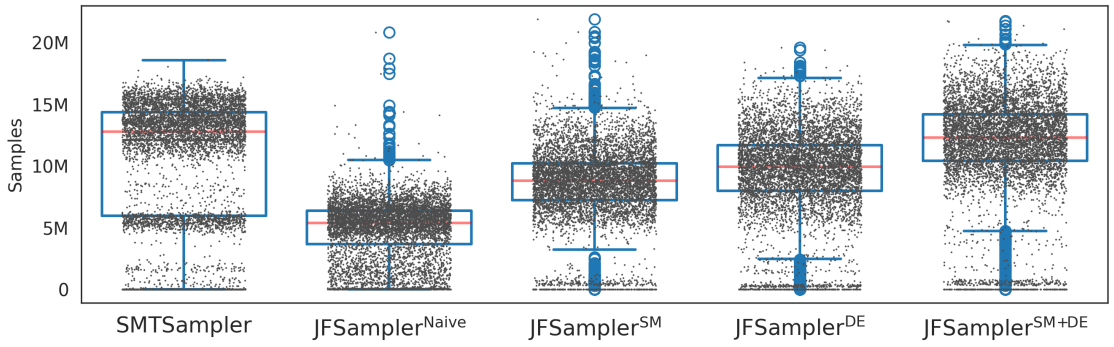
T_A vs T_B		QF_FP			QF_BVFP		
		A^{12}	*Winner*	*p*-value	A^{12}	*Winner*	*p*-value
JFSAMPLER^{Naive}	SMTSAMPLER	0.77	T_A	0.000	0.19	T_B	0.000
JFSAMPLER^{SM}	SMTSAMPLER	0.79	T_A	0.000	0.30	T_B	0.000
JFSAMPLER^{DE}	SMTSAMPLER	0.79	T_A	0.000	0.35	T_B	0.000
JFSAMPLER^{SM+DE}	SMTSAMPLER	0.79	T_A	0.000	0.51	T_A	0.000
JFSAMPLER^{SM}	JFSAMPLER^{Naive}	0.67	T_A	0.000	0.87	T_A	0.000
JFSAMPLER^{DE}	JFSAMPLER^{Naive}	0.63	T_A	0.000	0.88	T_A	0.000
JFSAMPLER^{SM+DE}	JFSAMPLER^{Naive}	0.66	T_A	0.000	0.92	T_A	0.000

the tables report the Vargha-Delaney A^{12} effect size. The A^{12} effect size is the probability that the technique T_A achieves a higher number of samples or SMT coverage than T_B on the same formula. Using the Wilcoxon signed rank test, we report the winning technique when this probability (when it is not 0.5) is statistically significant ($p < 0.001$).

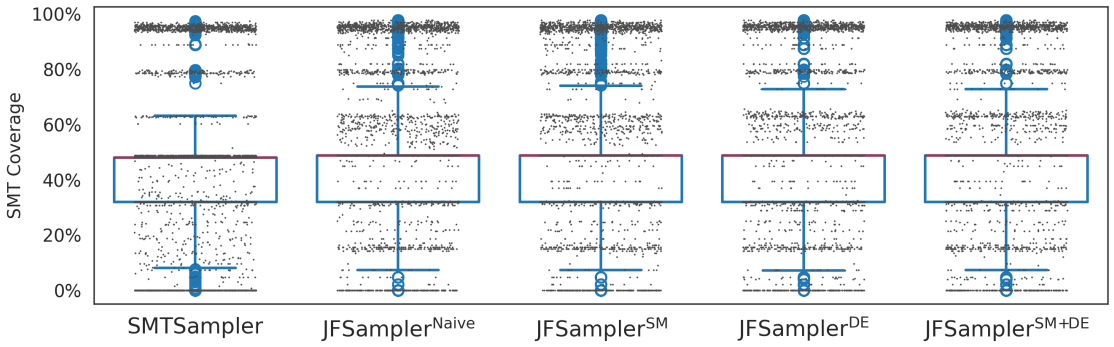
As we explain next, our results demonstrate that JFSAMPLER performs significantly better than SMTSAMPLER for the QF_FP suite, with the diversity encoding and sampling mutator making substantial contributions. In QF_BVFP, JFSAMPLER^{SM+DE} still emerges as the winner, but only by a small margin.

1) JFSampler^{Naive} vs. JFSampler^{SM}: JFSAMPLER^{SM} achieves higher throughput with statistical significance for both suites (Table I). This is because, as discussed in §III-C, JFSAMPLER^{SM} incorporates a new input mutator tailored to SMT sampling, that combines satisfying assignments (crashing inputs), whereas the default LIBFUZZER mutators are entirely domain-agnostic.

JFSAMPLER^{SM} obtains higher SMT coverage than JFSAMPLER^{Naive} with statistical significance for QF_FP (Table II). The increase in QF_FP is explained by the considerable increase in throughput. In QF_BVFP, the achieved coverage is similar to JFSAMPLER^{Naive}, possibly indicating that the achievable coverage in this suite is low and easily achieved by all techniques, including the SMTSAMPLER. It is worth noticing that the sub-expressions in the QF_FP have greater chances of having more achievable SMT coverage because expressions are either 32 or 64 bits. In contrast, in QF_BVFP, the bitvector expressions are smaller.



(a) Samples per formula in QF_BVFP.



(b) SMT coverage per formula in QF_BVFP.

Fig. 2: Total samples and SMT coverage for the QF_BVFP suite.

TABLE II: Statistical analysis of the difference in the SMT coverage per formula: A^{12} effect size of technique T_A relative to technique T_B (i.e., probability that T_A achieved higher SMT coverage than T_B), and the corresponding *Winner* for statistically significant cases (Wilcoxon signed-rank test, $p < 0.001$).

T_A vs T_B		QF_FP			QF_BVFP		
		A^{12}	*Winner*	*p*-value	A^{12}	*Winner*	*p*-value
JFSAMPLER^{Naive}	SMTSAMPLER	0.77	T_A	0.000	0.63	T_A	0.000
JFSAMPLER^{SM}	SMTSAMPLER	0.77	T_A	0.000	0.63	T_A	0.000
JFSAMPLER^{DE}	SMTSAMPLER	0.78	T_A	0.000	0.63	T_A	0.000
JFSAMPLER^{SM+DE}	SMTSAMPLER	0.78	T_A	0.000	0.63	T_A	0.000
JFSAMPLER^{SM}	JFSAMPLER^{Naive}	0.51	T_A	0.000	0.50	-	0.001
JFSAMPLER^{DE}	JFSAMPLER^{Naive}	0.57	T_A	0.000	0.50	-	0.000
JFSAMPLER^{SM+DE}	JFSAMPLER^{Naive}	0.57	T_A	0.000	0.50	-	0.000

2) JFSampler^{Naive} vs. JFSampler^{DE}: JFSAMPLER^{DE} achieves higher throughput than JFSAMPLER^{Naive} in both suites with statistical significance (Table I), explained as follows. LIBFUZZER tracks a corpus of interesting inputs based on code coverage as discussed in §III-B. In JFSAMPLER^{Naive}, inputs that satisfy the input formula will quickly saturate in coverage, so LIBFUZZER retains fewer in the in-memory corpus for subsequent mutations. However, JFSAMPLER^{DE} allows LIBFUZZER to retain more formula-satisfying inputs with different coverage thanks to the new program encoding. Mutating satisfying inputs has greater chances of generating a new satisfying input, increasing throughput.

JFSAMPLER^{DE} achieves higher SMT coverage compared to JFSAMPLER^{Naive} for the QF_FP suite in a statistically significant manner, but for QF_BVFP, it achieves comparable SMT coverage as JFSAMPLER (Table II).

The new inlined edges introduced by JFSAMPLER^{DE} may introduce certain overhead. This overhead is noticeable in the throughput difference between JFSAMPLER^{DE} and JFSAMPLER^{Naive}, higher in QF_BVFP but smaller in QF_FP. The sub-expressions in QF_FP have a higher number of bits than those in QF_BVFP, which impacts throughput. Nevertheless, the benefits are positive in both suites and JFSAMPLER^{DE} can effectively increase throughput and achieve equal or higher SMT coverage than JFSAMPLER^{Naive}.

3) JFSampler^{Naive} vs. JFSampler^{SM+DE} (all new features): In QF_BVFP, JFSAMPLER^{SM+DE} outperforms JFSAMPLER^{Naive}, JFSAMPLER^{SM} and JFSAMPLER^{DE} in throughput (Table I and Figure 2a). Similarly, in QF_FP, JFSAMPLER^{SM+DE} performs comparable to JFSAMPLER^{SM}, which performs the best (Figure

2a). However, JFSAMPLER^{SM+DE} performs at the top in terms of SMT coverage, whereas JFSAMPLER^{SM} does not (Table II and Figure 1b).

The slight difference in throughput between JFSAMPLER^{SM+DE} and JFSAMPLER^{SM} in QF_FP (Figure 1a) is explained by the overhead that the diversity encoding may introduce, although this does not outweigh the benefits. Based on these results, we consider JFSAMPLER^{SM+DE} to be the strategy that performs the best across the two suites.

4) JFSampler vs. SMTSampler: SMTSAMPLER is outperformed by JFSAMPLER (all variants) in the QF_FP suite in terms of throughput and SMT coverage (Figure 1). The difference between JFSAMPLER and SMTSAMPLER replicates the results in JFS. The low throughput is explained by the large amount of timeouts that SMTSAMPLER suffers during constraint solving, despite our timeout calibration. In addition, our new contributions improve JFSAMPLER^{Naive}, and make this difference more significant.

On the other hand, in QF_BVFP, SMTSAMPLER outperforms JFSAMPLER^{Naive}, JFSAMPLER^{SM} and JFSAMPLER^{DE} in the number of samples generated (Figure 2a). Notably, JFSAMPLER^{SM+DE}, the combination of our new features, allow us to reverse this difference and win by a slight difference with statistical significance (Table I). In other words, JFSAMPLER^{SM+DE} performs better than the baseline technique SMTSAMPLER and our naive sampler extension of JFS, JFSAMPLER^{Naive}, in both suites.

V. Related Work

A. SAT/SMT Samplers

The total number of models for a satisfiable formula is denoted as N, and a uniform sampler is designed to generate any given model with a probability of $1/N$. Examples of SAT samplers with uniformity guarantees include UNIGEN [38] and UNIGEN2 [31], as well as SEARCHTREESAMPLER [32], which provides approximately uniform guarantees.

In contrast, QUICKSAMPLER [27] is a SAT sampler that lacks uniformity guarantees and focuses on generating a diverse set of solutions. Compared to UNIGEN2 and SEARCHTREESAMPLER, it achieves significantly higher throughput and a reasonable degree of uniformity. For SMT formulas in the QF_AUFBV logic, SMTSAMPLER generalises and outperforms QUICKSAMPLER.

GUIDEDSAMPLER [39] builds on SMTSAMPLER by adding user-defined predicates to bias sampling, but retains the core logic of SMTSAMPLER. Extending JFSAMPLER with such predicates is future work. ESAMPLER [40] speeds up and complements SAT sampling by flipping bits in existing solutions that maintain satisfiability, avoiding solver calls. Integrating ESAMPLER with SMTSAMPLER or JFSAMPLER is non-trivial due to its focus on SAT, not SMT.

CMSGEN [41] is a SAT sampler that shows advantages over QUICKSAMPLER, although its effectiveness in the SMT domain remains unexplored.

B. Non-traditional Solvers

FUZZY-SAT [42], a constraint solver for concolic execution, uses fuzzing heuristics as mutators but does not encode formulas as programs or use coverage-guided fuzzing; it does not support floating-point constraints. JFS allowed us to leverage code coverage feedback to improve the SMT coverage metric.

C. Coverage-guided Fuzzing

ENTROPIC [43] is a seed power scheduling for LIBFUZZER, based on Shannon's information theory, and is now the default in LIBFUZZER. JFSAMPLER uses an old LIBFUZZER version, LLVM 6, containing custom modifications, thus we have not tested ENTROPIC. However, we expect ENTROPIC to benefit from our diversity encoding, improving the differentiation of satisfying assignments via code coverage feedback, because ENTROPIC and our encoding are two orthogonal features.

HASHFUZZ [44] is a semantic-preserving C program transformation for fuzzing that promotes uniform test case generation using *hash* functions to partition input space, rewarding fuzzers for covering these partitions. Similarly, JFSAMPLER^{DE} rewards satisfying assignments that increase SMT coverage. Unlike JFSAMPLER, HASHFUZZ enhances coverage-guided fuzzers rather than serving as an SMT sampler. Investigating HASHFUZZ 's impact on JFSAMPLER 's uniformity is future work; in this work, we compare JFSAMPLER to SMTSAMPLER, which lacks uniformity guarantees.

Recent work on the simulation of concurrency memory models investigated the use of coverage-guided fuzzing to detect violations of memory model properties in the absence of a bespoke simulator using a JFS-like encoding [45]. Adapting this method to search for multiple interesting inputs in the style of JFSAMPLER would allow a memory model analyst to obtain a diverse range of inputs that violate a particular memory model property of interest.

VI. Conclusion

In this work, we propose JFSAMPLER, a novel scalable SMT sampling method for floating-point constraints using coverage-guided fuzzing. JFSAMPLER leverages the efficiency of fuzzing, enhanced by a tailored input mutator and program encoding for sampling.

We benchmarked our contributions, comparing them against baselines in an extensive evaluation. This included the state-of-the-art SMTSAMPLER [25] technique, which matches JFSAMPLER in terms of uniformity. Results show that JFSAMPLER outperforms the baselines in both suites.

Future work will focus on improving the initial *smart seeds* and developing new mutators to combine satisfying assignments for more efficient sampling.

A replication package for our project is is available at [46].

Acknowledgements. This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).

References

[1] C. Cadar, D. Dunbar, and D. Engler, "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)*, Dec. 2008.

[2] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *Proc. of the 29th USENIX Security Symposium (USENIX Security'20)*, Aug. 2020.

[3] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[4] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, Jun. 2005.

[5] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang, "Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction," in *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'20)*, May 2020.

[6] D. Liu, G. Ernst, T. Murray, and B. I. P. Rubinstein, "Legion: Best-first concolic testing," in *Proc. of the 35th IEEE International Conference on Automated Software Engineering (ASE'20)*, Sep. 2020.

[7] C. Robert, J. Guiochet, H. Waeselynck, and L. V. Sartori, "TAF: a tool for diverse and constrained test case generation," in *Proc. of the 21th IEEE International Conference on Software Quality, Reliability and Security (QRS'21)*, Dec. 2021.

[8] R. Heradio, D. Fernandez-Amoros, J. A. Galindo, D. Benavides, and D. Batory, "Uniform and scalable sampling of highly configurable systems," *Empirical Software Engineering (EMSE)*, vol. 27, no. 2, 2022.

[9] KLEE, "Getting Involved," https://klee-se.org/getting-involved, 2024, last accessed 2025-01-21.

[10] angr, "FAQ," https://docs.angr.io/en/latest/faq.html#what-does-unsupportediroperror-floating-point-support-disabled-mean, 2025, last accessed 2025-01-21.

[11] Triton, "Floating point - ISA and the IEEE standard (issue #326)," https://github.com/JonathanSalwan/Triton/issues/326, 2016, last accessed 2025-01-21.

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

[13] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Proc. of the 23rd International Conference on Computer-Aided Verification (CAV'11)*, Jul. 2011.

[14] L. de Moura and N. Bjørner, "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)*, Mar. 2008.

[15] A. Cimatti, A. Griggio, B. J. Schaafsma, R. Sebastiani, and S. A. Smolka, "The MathSAT5 SMT solver," in *Proc. of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*, Mar. 2013.

[16] D. Liew, C. Cadar, A. Donaldson, and J. R. Stinnett, "Just fuzz it: Solving floating-point constraints using coverage-guided fuzzing," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, Aug. 2019.

[17] M. Souza, M. Borges, M. d'Amorim, and C. S. Păsăreanu, "CORAL: Solving complex constraints for symbolic pathfinder," in *Proc. of the 3rd International Conference on NASA Formal Methods (NFM'11)*, Apr. 2011.

[18] M. A. Ben Khadra, D. Stoffel, and W. Kunz, "goSAT: Floating-point satisfiability as global optimization," in *Proc. of the 17th Formal Methods in Computer-Aided Design (FMCAD'17)*, Oct. 2017.

[19] Z. Fu and Z. Su, "XSat: A fast floating-point satisfiability solver," in *Proc. of the 26th International Conference on Computer-Aided Verification (CAV'16)*, Jul. 2016.

[20] N. Bjørner and et al., "Programming Z3," https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-solving-bit-vectors, last accessed 2025-01-21.

[21] ——, "Programming Z3," https://theory.stanford.edu/~nikolaj/programmingz3.html#sec-floating-point-arithmetic, last accessed 2025-01-21.

[22] "Z3 very slow VC (issue #823)," https://github.com/Z3Prover/z3/issues/823#issuecomment-265463695, 2016, last accessed 2025-01-21.

[23] A. Brillout, D. Kroening, and T. Wahl, "Mixed abstractions for floating-point arithmetic," in *Proc. of the 9th Formal Methods in Computer-Aided Design (FMCAD'09)*, Nov. 2009.

[24] "LibFuzzer website," http://llvm.org/docs/LibFuzzer.html, 2022, last accessed 2025-01-21.

[25] R. Dutra, J. Bachrach, and K. Sen, "SMTSampler: Efficient Stimulus Generation from Complex SMT Constraints," in *Proc. of the 37th International Conference on Computer-Aided Design (ICCAD'19)*, Nov. 2018.

[26] S. Ranise and C. Tinelli, "The SMT-LIB format: An initial proposal," in *Proc. of the Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR'03)*, Jul. 2003.

[27] R. Dutra, K. Laeufer, J. Bachrach, and K. Sen, "Efficient sampling of SAT solutions for testing," in *Proc. of the 40th International Conference on Software Engineering (ICSE'18)*, May 2018.

[28] M. Zalewski, "Technical "whitepaper" for afl-fuzz," http://lcamtuf.coredump.cx/afl/technical_details.txt, last accessed 2025-01-21.

[29] D. Liew, D. Schemmel, C. Cadar, A. Donaldson, R. Zähl, and K. Wehrle, "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)*, Oct. 2017.

[30] "IEEE Standard for Floating-Point Arithmetic," Institute of Electrical and Electronics Engineers, Standard, 2008.

[31] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, "On parallel scalable uniform SAT witness generation," in *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, Apr. 2015.

[32] S. Ermon, C. Gomes, and B. Selman, "Uniform solution sampling using a constraint solver as an oracle," in *Proc. of the 28th Conference on Uncertainty in Artificial Intelligence (UAI'07)*, Aug. 2012.

[33] N. Bjørner, A.-D. Phan, and L. Fleckenstein, "νz - an optimizing SMT solver," in *Proc. of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15)*, Apr. 2015.

[34] R. Dutra, "SMTSampler: Efficient stimulus generation from complex SMT constraints," https://github.com/RafaelTupynamba/SMTSampler, 2018, last accessed 2025-01-21.

[35] D. Bueno, "Model evaluation performance (issue #2341)," https://github.com/Z3Prover/z3/issues/2341, 2019, last accessed 2025-01-21.

[36] M. Carrasco, C. Cadar, and A. Donaldson, "[artifact] SMTSampler's timeout calibration plots," https://zenodo.org/records/14651572/files/smtsampler-calibration-plots.zip, 2025, last accessed 2025-01-21.

[37] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[38] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Balancing scalability and uniformity in SAT witness generator," in *Proc. of the 51th Design Automation Conference (DAC'14)*, Jun. 2014.

[39] R. Dutra, J. Bachrach, and K. Sen, "GuidedSampler: Coverage-guided Sampling of SMT Solutions," in *Proc. of the 19th Formal Methods in Computer-Aided Design (FMCAD'19)*, Oct. 2019.

[40] Y. Xu, F. Song, and T. Chen, "ESampler: Efficient sampling of satisfying assignments for boolean formulas," in *Proc. of the 7th International Symposium on Software Engineering: Theories, Tools, and Applications (SETTA'21)*, Nov. 2021.

[41] P. Golia, M. Soos, S. Chakraborty, and K. S. Meel, "Designing samplers is easy: The boon of testers," in *Proc. of the 21th Formal Methods in Computer-Aided Design (FMCAD'21)*, Oct. 2021.

[42] L. Borzacchiello, E. Coppa, and C. Demetrescu, "Fuzzing symbolic expressions," in *Proc. of the 43rd International Conference on Software Engineering (ICSE'21)*, May 2021.

[43] M. Böhme, V. J. Manès, and S. K. Cha, "Boosting fuzzer efficiency: An information theoretic perspective," in *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'20)*, Nov. 2020.

[44] H. D. Menendez and D. Clark, "Hashing fuzzing: Introducing input diversity to improve crash detection," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 9, pp. 3540–3553, 2022.

[45] D. Iorga, J. Wickerson, and A. F. Donaldson, "Simulating operational memory models using off-the-shelf program analysis tools," *IEEE Trans. Software Eng.*, vol. 49, no. 12, pp. 5084–5102, 2023.

[46] M. Carrasco, C. Cadar, and A. Donaldson, "[artifact] Scalable SMT sampling for floating-point formulas via coverage-guided fuzzing," https://doi.org/10.5281/zenodo.14651572, 2025, last accessed 2025-01-21.