StringFuzz: A Fuzzer for String Solvers

Dmitry Blotsky[1], Federico Mora[2], Murphy Berzish[1], Yunhui Zheng[3], Ifaz Kabir[1], and Vijay Ganesh[1]

[1] University of Waterloo
[2] University of Toronto
[3] IBM T.J. Watson Research Center

Abstract. In this paper, we introduce StringFuzz: a modular SMT-LIB problem instance transformer and generator for string solvers. We supply a repository of instances generated by StringFuzz in SMT-LIB 2.0/2.5 format. We systematically compare Z3str3, CVC4, Z3str2, and Norn on groups of such instances, and identify those that are particularly challenging for some solvers. We briefly explain our observations and show how StringFuzz helped discover causes of performance degradations in Z3str3.

1 Introduction

In recent years, many algorithms for solving string constraints have been developed and implemented in SMT solvers such as Norn [6], CVC4 [12], and Z3 (e.g., Z3str2 [13] and Z3str3 [7]). To validate and benchmark these solvers, their developers have relied on hand-crafted input suites [1, 5, 4] or real-world examples from a limited set of industrial applications [2, 11]. These test suites have helped developers identify implementation defects and develop more sophisticated solving heuristics. Unfortunately, as more features are added to solvers, these benchmarks often remain stagnant, leaving increasing functionality untested. As such, there is an acute need for a more robust, inexpensive, and automatic way of generating benchmarks to test the correctness and performance of SMT solvers.

Fuzzing has been used to test all kinds of software including SAT solvers [10]. Inspired by the utility of fuzzers, we introduce StringFuzz and describe its value as an exploratory testing tool. We demonstrate its efficacy by presenting limitations it helped discover in leading string solvers. To the best of our knowledge, StringFuzz is the only tool aimed at automatic generation of string constraints. StringFuzz can be used to mutate or transform existing benchmarks, as well as randomly generate structured instances. These instances can be scaled with respect to a variety of parameters, e.g., length of string constants, depth of concatenations (concats) and regular expressions (regexes), number of variables, number of length constraints, and many more.

Contributions [4]

1. **The StringFuzz tool**: In Sect. 2, we describe a modular fuzzer that can transform and generate SMT-LIB 2.0/2.5 string and regex instances. Scaling inputs (e.g., long string constants, deep concatenations) are particularly useful in identifying asymptotic behaviors in solvers, and StringFuzz has many options to generate them. We briefly document StringFuzz's components and modular architecture. We provide example use cases to demonstrate its utility as an exploratory solver testing tool.
2. **A repository of SMT-LIB 2.0/2.5 instances**: We present a repository of SMT-LIB 2.0/2.5 string and regex instance suites that we generated using StringFuzz in Sect. 3. This repository consists of two categories: one with new instances generated by StringFuzz (`generated`); and another with transformed instances generated from a small suite of industrial benchmarks (`transformed`).
3. **Experimental Results and Analysis**: We compare the performance of Z3str3, CVC4, Z3str2, and Norn on the StringFuzz suites *Concats-Balanced*, *Concats-Big*, *Concats-Extracts-Small*, and *Different-Prefix* in Sect. 4. We highlight these suites because they make some solvers perform poorly, but not others. We analyze our experimental results, and pinpoint algorithmic limitations in Z3str3 that cause poor performance.

2 StringFuzz

Implementation and Architecture StringFuzz is implemented as a Python package, and comes with several executables to generate, transform, and analyze SMT-LIB 2.0/2.5 string and regex instances. Its components are implemented as UNIX "filters" to enable easy integration with other tools (including themselves). For example, the outputs of generators can be piped into transformers, and transformers can be chained to produce a stream of tuned inputs to a solver. StringFuzz is composed of the following tools:

`stringfuzzg`
: This tool generates SMT-LIB instances. It supports several generators and options that specify its output. Details can be found in Table 1a.

`stringfuzzx`
: This tool transforms SMT-LIB instances. It supports several transformers and options that specify its output and input, which are explained in Table 1b. Note that transformers *Translate* and *Reverse* also preserve satisfiability under certain conditions.

`stringstats`
: This tool takes an SMT-LIB instance as input and outputs its properties: the number of variables/literals, the max/median syntactic depth of expressions, the max/median literal length, etc.

[4]All source code, problem suites, and supplementary material referenced in this paper are available at the StringFuzz website [3].

Table 1: StringFuzz built-in (a) generators and (b) transformers.

(a) `stringfuzzg` built-in generators.

Name	**Generates instances that have ...**
Concats	Long concats and optional random extracts.
Lengths	Many variables (and their concats) with length constraints.
Overlaps	An expression of the form A.X = X.B.
Equality	An equality among concats, each with variables or constants.
Regex	Regexes of varying complexity.
Random-Text	Totally random ASCII text.
Random-AST	Random string and regex constraints.

(b) `stringfuzzx` built-in transformers.

Name	**The transformer ...**
Fuzz	Replaces literals and operators with similar ones.
Graft	Randomly swaps non-leaf nodes with leaf nodes.
Multiply[5]	Multiplies integers and repeats strings by N.
Nop	Does nothing (can translate between SMT-LIB 2.0/2.5).
Reverse[6]	Reverses all string literals and concat arguments.
Rotate	Rotates compatible nodes in syntax tree.
Translate[6]	Permutes the alphabet.
Unprintable	Replaces characters in literals with unprintable ones.

We organized StringFuzz to be easily extended. To show this, we note that while the whole project contains 3,183 lines of code, it takes an average of 45 lines of code to create a transformer. StringFuzz can be installed from source, or from the Python PIP package repository.

Regex Generating Capabilities StringFuzz can generate and transform instances with regex constraints. For example, the command "`stringfuzzg regex -r 2 -d 1 -t 1 -M 3 -X 10`" produces this instance:

```
(set-logic QF_S)
(declare-fun var0 () String)
(assert (str.in.re var0 (re.+ (str.to.re "R5"))))
(assert (str.in.re var0 (re.+ (str.to.re "!PC"))))
(assert (<= 3 (str.len var0)))
(assert (<= (str.len var0) 10))
(check-sat)
```

Each instance is a set of one or more regex constraints on a single variable, with optional maximum and minimum length constraints. Each regex constraint

[5]Can guarantee satisfiable output instances from satisfiable input instances [3].
[6]Can guarantee input and output instances will be equisatisfiable [3].

is a concatenation (`re.++` in SMT-LIB string syntax) of regex terms:

```
(re.++ T1 (re.++ T2 ... (re.++ Tn-1 Tn )))
```

and each term `Ti` is recursively defined as any one of: repetition (`re.*`), Kleene star (`re.+`), union (`re.union`), or a character literal. Nested operators are nested up to a specified (using the `--depth` flag) depth of recursion. Terms at depth 0 are regex constants. Below are 3 example regexes (in regex, not SMT-LIB, syntax) of depth 2 that can be produced this way:

$$((\mathtt{a}|\mathtt{b})|(\mathtt{cc})+) \qquad ((\mathtt{ddd})*)+ \qquad ((\mathtt{ee})+|(\mathtt{fff})*)$$

Equisatisfiable String Transformations StringFuzz can also transform problem instances. This is done by manipulating parsed syntax trees. By default most of the built-in transformers only guarantee well-formedness, however, some can even guarantee equisatisfiability. Table 1b lists the built-in transformers and notes these guarantees.

Example Use Case In Sect. 3 we use StringFuzz to generate benchmark suites in a batch mode. We can also use StringFuzz for on-line exploratory debugging. For example, the script below repeatedly feeds random StringFuzz instances to CVC4 until the solver produces an error:

```
while stringfuzzg -r random-ast -m \
    | tee instance.smt25 | cvc4 --lang smt2.5 --tlimit=5000 --strings-exp; do
    sleep 0
done
```

3 Instance Suites

In this section, we describe the benchmark suites we generated with StringFuzz, and on which we conducted our experimental evaluation. Table 2a lists instances that were generated by `stringfuzzg`. Table 2b lists instances derived from existing seed instances by iteratively applying `stringfuzzx`. Every transformed instance is named according to its seed and the transformations it undertook. For example, `z3-regex-1-fuzz-graft.smt2` was transformed by applying *Fuzz* and then *Graft* to `z3-regex-1.smt2`.

The *Amazon* category contains 472 instances derived from two seeds supplied by our industrial collaborators. The *Regex* category is seeded by the Z3str2 regex test suite [4], which contains 42 instances. Through cumulative transformations we expanded the 42 seeds to 7,551 unique instances. Finally, the *Sanitizer* category is obtained from five industrial e-mail address and IPv4 sanitizers.

4 Experimental Results and Analysis

We generated several problem instance suites with StringFuzz that made one solver perform poorly, but not others.[7] They are *Concats-Balanced*, *Concats-Big*,

[7]Only the results that made one solver perform poorly and not others are presented, but results for all StringFuzz suites are available on the StringFuzz website[3].

Table 2: Repository of 10,258 SMT-LIB 2.0/2.5 instances.

(a) `stringfuzzg`-generated instances.

Name	**Instances have a ...**	**Quantity**
Concats-{*Small,Big*}	Right-heavy, deep tree of concats.	120
Concats-Balanced	Balanced, deep tree of concats.	100
Concats-Extracts-{*Small,Big*}	Single concat tree, with character extractions.	120
Lengths-{*Long,Short*}	Single, large length constraint on a variable.	200
Lengths-Concats	Tree of fixed-length concats of variables.	100
Overlaps-{*Small,Big*}	Formula of the form A.X = X.B.	80
Regex-{*Small,Big*}	Complex regex membership test.	120
Many-Regexes	Multiple random regex membership tests.	40
Regex-Deep	Regex membership test with many nested operators.	45
Regex-Pair	Test for membership in one regex, but not another.	40
Regex-Lengths	Regex membership test, and a length constraint.	40
Different-Prefix	Equality of two deep concats with different prefixes.	60

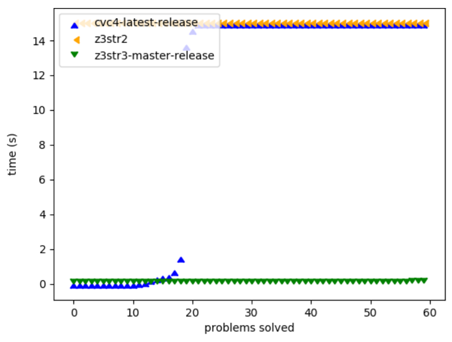
(b) `stringfuzzx`-generated instances.

Name	**Seed**	**Quantity**
Amazon	Two industrial regex membership instances.	472
Regex	Z3str2 regular expression test suite.	7,551
Sanitizer	Five e-mail and IPv4 sanitiser examples.	1,170

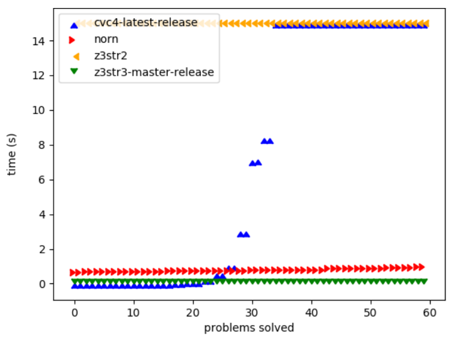
Concats-Extracts-Small, and *Different-Prefix*. Fig. 1 shows the suites that were uniquely difficult for CVC4. Fig. 2 shows the suites that were uniquely difficult for Z3str3. All experiments were conducted in series, each with a timeout of 15 seconds, on an Ubuntu Linux 16.04 computer with 32GB of RAM and an Intel® Core$^{\text{TM}}$ i7-6700 CPU (3.40GHz).

Usefulness to Z3str3: A Case Study StringFuzz's ability to produce scaling instances helped uncover several implementation issues and performance limitations in Z3str3. Scaling inputs can reveal issues that would normally be out of scope for unit tests or industrial benchmarks. Three different performance and implementation bugs were identified and fixed in Z3str3 as a result of testing with the StringFuzz scaling suites *Lengths-Long* and *Concats-Big*.

StringFuzz also helped identify a number of performance-related issues and opportunities for new heuristics in Z3str3. For example, by examining Z3str3's execution traces on the instances in the *Concats-Big* suite we discovered a potential new heuristic. In particular, Z3str3 does not make full use of the solving context (e.g. some terms are empty strings) to simplify the concatenations of a long list of string terms before trying to reason about the equivalences among sub-terms. Z3str3 therefore introduces a large number of unnecessary intermediate variables and propagations.

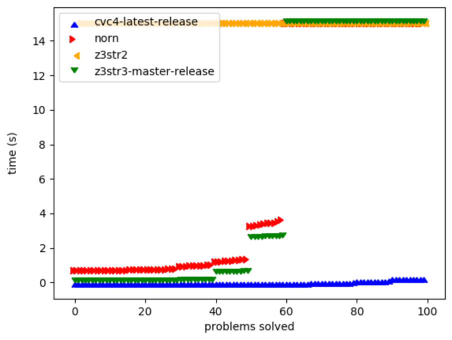


(a) Performance on *Concats-Extracts-Small*

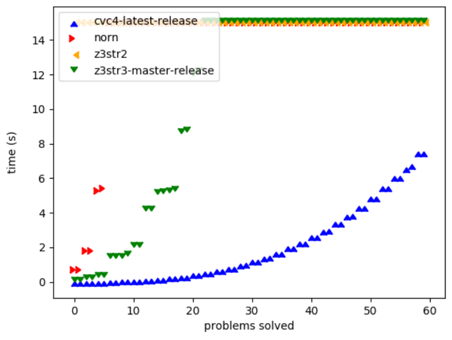


(b) Performance on *Different-Prefix*

Fig. 1: Instances hard for CVC4



(a) Performance on *Concats-Balanced*



(b) Performance on *Concats-Big*

Fig. 2: Instances hard for Z3str3

5 Related Work

Many solver developers create their own test suites to validate their solvers [1, 5, 4]. Several popular instance suites are also publicly available for solver testing and benchmarking, such as the Kaluza [2] and Kausler [11] suites. There are likewise several fuzzers and instance generators currently available, but none of them can generate or transform string and regex instances. For example, the FuzzSMT [9] tool generates SMT-LIB instances with bit-vectors and arrays, but does not support strings or regexes. The SMTpp [8] tool pre-processes and simplifies instances, but does not generate new ones or fuzz existing ones.

References

1. CVC4 regression test suite. `https://github.com/CVC4/CVC4/tree/master/test/regress`.
2. Kaluza benchmark suite. `http://webblaze.cs.berkeley.edu/2010/kaluza/`.
3. Stringfuzz source code, benchmark suites, and suplemental material. `http://stringfuzz.dmitryblotsky.com`.
4. Z3str2 test suite. `https://github.com/z3str/Z3-str/tree/master/tests`.
5. Z3str3 test scripts. `https://github.com/Z3Prover/z3/tree/master/src/test`.
6. P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. Norn: An SMT solver for string constraints. In D. Kroening and C. S. Păsăreanu, editors, *Computer Aided Verification*, pages 462–469, Cham, 2015. Springer International Publishing.
7. M. Berzish, V. Ganesh, and Y. Zheng. Z3str3: A string solver with theory-aware heuristics. In D. Stewart and G. Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.
8. R. Bonichon, D. Déharbe, P. Dobal, and C. Tavares. SMTpp: preprocessors and analyzers for SMT-LIB. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories*, SMT 2015, 2015.
9. R. Brummayer and A. Biere. Fuzzing and delta-debugging SMT solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, SMT '09, pages 1–5, New York, NY, USA, 2009. ACM.
10. R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of sat and qbf solvers. In O. Strichman and S. Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
11. S. Kausler and E. Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 259–270, New York, NY, USA, 2014. ACM.
12. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. *A DPLL(T) theory solver for a theory of strings and regular expressions*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer Verlag, 2014.
13. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a Z3-based string solver for web application analysis. In B. Meyer, L. Baresi, and M. Mezini, editors, *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 114–124. ACM, 2013.