StringFuzz: A Fuzzer for String SMT Solvers

by

Dmitry Blotsky

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2018

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Some tables and text in this thesis are restated from the StringFuzz paper [13].

Some of the source code of StringFuzz was written and edited by Federico Mora.

Abstract

We introduce StringFuzz, a software tool for automatically testing string SMT solvers. String SMT solvers are specialised software tools for solving the Satisfiability Modulo Theories (SMT) problem with string contraints, which is a type of constraint satisfaction problem applicable in industry. Like all tools, string SMT solvers need testing. The developers of solvers commonly test them with published test suites: pre-generated sets of problem instances (i.e. example problems). As new features are added to string SMT solvers, they often are not exercised by existing suites. We introduce StringFuzz, a tool for solver developers to generate SMT instances to exercise and find defects in their solvers. We describe StringFuzz's features for generating and transforming SMT instances with string and regex constraints. We also show StringFuzz's many controls, and show how to use them to generate specially tuned scaling instances. For public use, we present our own suite of StringFuzz-generated SMT instances. We also introduce StringBreak, an automated exploratory tester for string SMT solvers, which uses a genetic algorithm to generate SMT instances that take a long time for solvers to solve. To demonstrate the usefulness of StringFuzz and StringBreak, we show experimental results from testing leading string SMT solvers (Z3str3, CVC4, Z3str2, and Norn) with them. We describe two defects and one potential future enhancement that we discovered in Z3str3 as a result of our experiments.

Acknowledgements

Firstly, I would like to thank my family - my parents, Sergey Blotsky and Ekaterina Blotskaya, and my fiancée, Kristine Prelich - for their unwavering support of my graduate studies and work on this thesis. Through all the setbacks, "no time for anything"s, and evenings of my complaints, they were always there to comfort me, encourage me, and be my backbone when I didn't have one. I would most definitely have abandoned this work if it was not for you. I love you all.

I am also grateful to my dear friend John Shultis, who supported me throughout the four years since the end of undergraduate studies until the completion of this thesis. For all the reasoned discussions, the proofreading, and the unending encouragement when I wanted to just give up, I thank you.

I also thank Federico Mora, who was not only a skilled expert and valuable contributor to the source code of StringFuzz, but also simultaneously a mentee and mentor. Without his work and feedback, StringFuzz would have been much the lesser, and there would still be nothing to write about.

A great thanks as well to the evaluating readers of this thesis: Dr. Lin Tan, Dr. Derek Rayside, and Dr. Vijay Ganesh. I appreciate you putting up with all the short-notice emails, handling the uncertainties, and ultimately reading all the walls of text enclosed herein. In addition, I want to thank all the people who helped proofread this thesis.

I also thank en masse the organisations: the University of Waterloo and the Computer Aided Verification conference program committee; the CAV PC for reviewing and graciously accepting the StringFuzz paper for publication, and the faculty at the University of Waterloo, my alma mater, for enabling and providing the entirety of my undergraduate and graduate education.

Perhaps intangibly so, I would also like to express my thanks for Canadian society as a whole, and my thanks to my parents for bringing me to Canada in my youth. I would not have had nearly the educational opportunities elsewhere that I was afforded here.

Finally, I would like to thank my supervisor, Dr. Vijay Ganesh, for guiding me through the entire process of completing this thesis, from conception to printing. Without your

academic direction, I would still be looking for something about which to write. Thank you for all the advice - be it over desk, dinner, or phone - to steer my flailing efforts into something useful.

Dedication

This is dedicated to my two grandfathers: Виталий (Vitaliy) and Иван (Ivan).

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit 1

CSP Constraint Satisfaction Problem 1

DPLL Davis–Putnam–Logemann–Loveland 6

regex regular expression 4, 11, 30

SMT Satisfiability Modulo Theories 1, 2, 4, 6, 7, 10, 30

List of Symbols

Bool The sort inhabited by the Boolean values $True$ and $False$. 7

Int The sort inhabited by integers. 7

RegExp The sort inhabited by regular expressions. 7

String The sort inhabited by strings. 7

Chapter 1

Introduction

Constraint Satisfaction Problem s (CSPs) are, informally, a type of mathematical problem that involves a set of variables and a set of constraints on those variables. CSPs are common in academia and industry as formalisations of many real-world phenomena. Some examples of things that can be expressed as CSPs are: solving Sudoku puzzles, optimising cargo route plans at a shipping company, and verifying that a Central Processing Unit (CPU) design implements its specifications. Formulating and solving CSPs is therefore an economically valuable activity, and many software tools exist to facilitate it.

1.1 String SMT Solvers

The Satisfiability Modulo Theories (SMT) problem is a decision problem that can be stated as follows: is a given first-order logic formula (with predicates from some set of formal theories) satisfiable? It is a type of CSP. Many real-world scenarios can be formalised as SMT problems; for example: proving mathematical theorems, and finding bugs in software. The string SMT problem is further explained and defined more formally in Sect. 2.1.

Software tools that can solve SMT problem instances are called SMT solvers. SMT solvers that can solve SMT problem instances with string constraints are called string SMT solvers. There are many algorithms for solving the SMT problem, and for solving

specifically the string constraints. They are described in further detail in Sect. 2.3. There are many string SMT solvers currently available, and in this thesis we will focus on four of them: Z3str3, CVC4, Z3str2, and Norn [12, 20, 24, 10]. They are described in further detail in Sect. 2.5.

1.2 Solver Testing

Like all tools, string SMT solvers need to be tested. Solver developers commonly test their solvers with published test suites: sets of SMT problem instances. For example, the authors of each of the solvers CVC4, Z3str2, and Kaluza have published their test suites for use by others [1, 8, 4].

To test a solver for correctness, solver authors need to compare its answer for a problem instance with an oracle: something that gives the correct answer. Usually an oracle is an existing solver that is known to give correct answers, or a human that carried out some decision procedure (like a proof, at instance construction time) to find the correct answer.

1.3 Motivation

Unfortunately, once instance suites are published, their authors rarely update them over time. That means that when testing requirements for solvers change, existing suites may no longer address them. Furthermore, it is not always possible to use another solver as an oracle, so solver authors need to rely on another method of knowing the answers to problem instances. From this, we identified two needs of solver authors: generating new instances to suit their needs, and easily knowing the answers to those instances.

1.4 Contributions

In this thesis we present the following contributions:

1. **The StringFuzz tool**: In Chap. 3 we describe StringFuzz, a modular tool we created for solver developers to test their solvers. StringFuzz can generate and transform SMT instances in SMT-LIB format, and has many modules and controls for doing so. We describe its features and give example use cases, some of which we used ourselves in our experiments.

2. **A suite of SMT-LIB instances**: In Sect. 3.5 we describe a suite of SMT instances in SMT-LIB format that we generated with StringFuzz and published on our website [6]. The suite consists of two sub-suites: one (called *Generated*) with problems generated by StringFuzz from scratch, and another (called *Transformed*) generated by StringFuzz by transforming seed instances from industrial applications.

3. **The StringBreak tool**: In Chap. 4 we describe StringBreak, another tool we created for solver developers to test their solvers. StringBreak uses a genetic algorithm to generate and mutate SMT instances, selecting for instances that take a long time for a given solver to solve. We describe elements of the genetic algorithm (like the fitness function), and present some example instances that StringBreak discovered.

4. **Experimental Results and Analysis**: In Sect. 3.6 we compare the performance of Z3str3, CVC4, Z3str2, and Norn on the StringFuzz sub-suites *Concats-Balanced*, *Concats-Big*, *Concats-Extracts-Small*, and *Different-Prefix*. We highlight these sub-suites because some solvers could solve them quickly and others could not, and vice versa. We analyse the results and identify algorithmic limitations in Z3str3 that cause it to perform poorly on some suites.

   In Sect. 4.2 we exercise Z3str3 and CVC4 with StringBreak,. We present the performance results of these two solvers on the instances StringBreak found.

1.5 Related Work

There are currently several published SMT problem instance suites. Some small suites were created and published by solver developers to test their own respective solvers [1, 9, 8].

There are also the larger Kaluza [4] and Kausler [19] suites, which are commonly used for benchmarking as well as testing.

Aside from published suites, there are already fuzzers and instance generators currently available, but none of them can generate instances with string or regular expression (regex) constraints. For example, the FuzzSMT [15] tool can generate SMT instances with bit-vector constraints and arrays, but not with strings or regexes. The SMTpp [14] tool can pre-process and simplify existing instances, but cannot generate new ones or arbitrarily alter existing ones.

StringFuzz can both generate and transform instances with string and regex constraints, and it has input flags for its user to control its generating and transforming behaviour.

Chapter 2

Background

This chapter provides background information and formalisations of concepts that will be relevant in later chapters. We assume that the reader is familiar with propositional and first-order logic, and with UNIX shell scripting syntax.

2.1 The SMT Problem

Definition 2.1.1. *Decision problem* A problem that can be framed as a yes-or-no question about its input variables.

Definition 2.1.2. *Satisfiability* A formula is satisfiable if there exists some assignment of values to its variables that make the formula evaluate to $True$.

The SMT Problem

Input: A first-order logic formula with the equality relation and predicates from background theories.

Output: $True$ if the formula is satisfiable; $False$ otherwise.

Informally, the SMT problem is a decision problem that asks whether a given first-order logic formula with variables from mixed domains is satisfiable. The different domains and the values, functions, and relations that belong to them are called a *background theory*. A background theory in the SMT problem can be any formal theory. Some examples are: the theory of integers (integers, addition, negation, etc.), the theory of sequences (empty sequence, intersection, etc.), and the theory of bit vectors. Each theory in in the SMT problem has a *signature*: a 3-tuple of *sorts*, *values* that inhabit the sorts, and *functions* that operate on the values.

2.2 The Theory of Strings

The specific background theory relevant to StringFuzz and string SMT solvers is the theory of strings. Although they share fundamental parts, the signatures for the theory of strings differ from solver to solver. For example, some solvers only define the theory of strings in terms of regular expressions, and others only support strings and do not support regular expressions at all.

Tables 2.1a and 2.1b describe an example signature for a theory of strings. For completeness, the full theories supported by CVC4 and Z3str3 are available online [7, 3].[1]

2.3 String SMT Solving Algorithms

The Davis–Putnam–Logemann–Loveland (DPLL) [17] algorithm is an algorithm commonly used to solve the satisfiability (SAT) problem for a propositional logic formula. However, propositional logic formulae only have Boolean variables and operators, and SMT problem instances have variables and functions from background theories. To solve the SMT problem there is a similar algorithm: the DPLL(T) algorithm [23] (the **T** is for **T**heories). The formal definitions of either algorithm are not reproduced in this thesis, but informally,

[1] The documents at the provided links use the SMT-LIB format, which is described in Sect. 2.4.

Table 2.1: One possible signature of the theory of strings

(a) String sorts

Sort Symbol	Domain
String	Strings
RegExp	Regular expressions

(b) String functions

Function Symbol	Signature	Value
Length	$(String) \rightarrow Int$	The length of a string
Concat	$(String\ String) \rightarrow String$	The concatenation of two strings
CharAt	$(String\ Int) \rightarrow Bool$	The character at a given position in a string

the DPLL(T) algorithm maps theory-specific predicates to Boolean variables, and invokes theory-specific solvers to determine satisfiability of those predicates.

Most SMT solvers implement the DPLL(T) algorithm, and include a SAT solver and several theory-specific sub-solvers. String SMT solvers include sub-solvers for the theory of strings. Different solvers have different approaches for solving string and regex constraints, and Sect. 2.5 briefly describes the approach of each solver that we tested.

2.4 The SMT-LIB Language

There is a standard language for describing SMT problem instances, called SMT-LIB [11]. Most SMT solvers accept instances in the SMT-LIB language as input, and produce an answer as output. The SMT-LIB language describes both the formula that constitutes a problem instance, and the commands for a solver to process it. For example, below is a simple first-order logic formula with elements from the theory of strings.

$$\texttt{"Hello "} \cdot X = \texttt{"Hello World"}$$

It has one string variable, X, and one string constraint: an equality between a string literal and the concatenation (represented by the $\cdot$ symbol) of X and another string literal. Informally, to solve the SMT problem for this formula is to find a value for X such that the concatenation of `"Hello "` and X is equal to `"Hello World"`. Its corresponding SMT-LIB representation (with a command on the last line for the solver to compute the satisfiability of the formula) is:

```
(set-logic QF_S)
(declare-fun X () String)
(assert (= (str.++ "Hello " X) "Hello World"))
(check-sat)
```

The output of the CVC4 solver for the abover instance is:

```
sat
```

If a command is added to also produce the model (which is `(get-model)` in SMT-LIB), then CVC4 outputs:

```
sat
(model
(define-fun X () String "World")
)
```

The full grammar of SMT-LIB is available online at `http://smt-lib.org`.

2.5 Existing String SMT Solvers

We chose four solvers for our experiments: Z3str3, CVC4, Z3str2, and Norn. Each solver implements different solving techniques, and we briefly describe them in this section. For more formal descriptions, we direct the reader to the solvers' respective published papers.

CVC4 The CVC4 solver is an SMT solver and has a sub-solver for the theory of strings. The authors of CVC4 formalise a calculus with reduction rules, which reduce string constraints to simpler string constraints, and ultimately to SAT or UNSAT [20]. CVC4 repeatedly applies those rules to solve string constraints.

Z3str3 (and Z3str2) The Z3str3 (and Z3str2) solver is a theory-specific extension (for the theory of strings) to the Z3 SMT solver [18, 12]. Z3str3 solves string constraints by interpreting them as word equations, and then searching the space of possible arrangements of variable concatenations for a satisfying assignment.

Norn The Norn solver is an SMT solver and has a sub-solver for the theory of strings [10]. It uses a mixed approach that is similar to both Z3str3 and CVC4: it treats some constraints as word equations and simplifies them (like Z3str3), and it applies reduction rules to other constraints (like CVC4).

Chapter 3

StringFuzz

StringFuzz is the main contribution of this thesis. It is a software package for string SMT solver developers, and comes with several tools to generate, transform, and measure SMT instances with string constraints, in SMT-LIB format. StringFuzz consists of the following tools:

`stringfuzzg`
: This tool generates SMT-LIB instances with string and regex constraints. Sect. 3.1 describes it in detail.

`stringfuzzx`
: This tool transforms SMT-LIB instances with string and regex constraints. Sect. 3.2 describes it in detail.

`stringbreak`
: This tool generates an SMT-LIB instance that takes a long time for a given solver to solve. Chap. 4 describes it in detail.

`stringmerge`
: This tool takes two or more SMT-LIB instances as input and merges them into one, which it outputs. Sect. 3.3 describes it in detail.

`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.

All of the tools, source code, documentation, and published instances that come with StringFuzz are available on our website [6].

3.1 Generator

`stringfuzzg` is an executable provided by StringFuzz. It can generate SMT-LIB instances with string and regex constraints. It implements several generation strategies (called *generators*), each one with input flags that can be used control its behaviour. Table 3.1 describes the built-in generators and the instances they generate. The command `stringfuzzg --help` documents the meanings of specific flags and generators that `stringfuzzg` supports. We explain some example uses of `stringfuzzg` below.

In Sect. 3.5.1 we present the suite of instances we created using `stringfuzzg`. For example, we generated the last instance of the *Concats-Small* sub-suite with this command:

```
stringfuzzg concats --depth 88 --depth-type semantic --solution solution
```

In addition to generating instances in a batch, StringFuzz can also work as a random fuzzer. For example, the script below repeatedly feeds random (but semantically valid) instances to CVC4 until the solver takes more than 5 seconds to solve one of them:

```
while true; do
    ANSWER=$(stringfuzzg -r random-ast --meaningful --num-vars 3 --depth 3 \
            | tee instance.smt25 \
            | cvc4 --lang smt2.5 --tlimit=5000 --strings-exp)
    echo $ANSWER
```

```
    if [[ "$ANSWER" = "unknown" ]] || [[ "$ANSWER" = "timeout" ]]; then
        cat instance.smt25
        break
    fi
done
```

Table 3.1: `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.

3.2 Transformer

`stringfuzzx` is an executable provided by StringFuzz. It implements several transformation strategies (called *transformers*), each one with input flags that can be used control its behaviour. Table 3.2 describes the supported transformers. In Sect. 3.5.2 we describe the suite of instances we created using `stringfuzzx`.

The transformers *Translate* and *Reverse* preserve satisfiability under certain conditions. The command `stringfuzzx --help` documents the meanings of specific flags and controls that it supports.

Table 3.2: `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	Multiplies integers and repeats strings by N.
Nop	Does nothing (can translate between SMT-LIB).
Reverse	Reverses all string literals and concat arguments.
Rotate	Rotates compatible nodes in syntax tree.
Translate	Permutes the alphabet.
Unprintable	Replaces characters in literals with unprintable ones.

3.3 Merger

`stringmerge` is an executable provided by StringFuzz. It merges two or more SMT-LIB instances into one. `stringmerge` supports three variable merging strategies: Concat, Disjoint, and Intersect.

Concat The output instance is a concatenation of the inputs, with duplicate statements removed. The following command implements this merge:

```
stringmerge a.smt25 b.smt25 simple
```

Disjoint This strategy is the same as Concat, but it renames all the variables in each input instance to fresh names. The output instance describes disjoint formulas, one for each input instance. The following command implements this merge:

```
stringmerge a.smt25 b.smt25 simple --renaming disjoint
```

Intersect This strategy does the opposite of Disjoint. Instead of renaming all variables to fresh names, it creates one name pool and renames all variables to names in the pool. The output instance describes one formula with some variables having constraints from one or more of the input formulas. The following command implements this merge:

```
stringmerge a.smt25 b.smt25 simple --renaming intersecting
```

3.4 Implementation

We implemented StringFuzz in the Python programming language. We made several design decisions to optimise for its interoperability with other tools and for ease of future extension.

To ensure interoperability, we implemented the `stringfuzzg` and `stringfuzzx` tools as UNIX filters. This means that `stringfuzzg` can send its output into `stringfuzzx`, and then in turn that output can be sent to a solver. The following command illustrates this:

```
stringfuzzg concats | stringfuzzx fuzz | z3str3 -in
```

We organized StringFuzz to be easily extended. To show this, we note that while the whole project contains 3,183 lines of code, on average it has 45 lines of code per transformer and 109 lines of code per generator.

3.5 StringFuzz Suite

We generated a suite of SMT-LIB instances with `stringfuzzg` and `stringfuzzx`, and published them online [5]. We conducted the experiments in Sect. 3.6 on these instances.

3.5.1 *Generated*

Table 3.3 lists instances that we generated with `stringfuzzg`. The instances in each `stringfuzzg`-generated sub-suite are grouped by a common property. For example all instances in the *Concats-Balanced* sub-suite describe formulas with a deep, balanced tree of concatenations. In addition, the instances in each sub-suite are scaling: they range from easier to harder instances. We achieved this using `stringfuzzg`'s controls for instance properties (like depth, or the number of variables).

Table 3.3: The *Generated* suite.

Name	**Each instance is ...**	**Count**
Concats-Small	A right-heavy, deep tree of concats.	60
Concats-Big	Same as above, but more extreme.	60
Concats-Balanced	A balanced, deep tree of concats.	100
Concats-Extracts-Small	A single concat tree with character extracts.	60
Concats-Extracts-Big	Same as above, but more extreme.	60
Lengths-Long	One single, large length constraint.	100
Lengths-Short	Same as above, but more extreme.	100
Lengths-Concats	A tree of fixed-length concats of variables.	100
Overlaps-Small	A formula of the form A.X = X.B.	40
Overlaps-Big	Same as above, but more extreme.	40
Regex-Small	A complex regex membership test.	60
Regex-Big	Same as above, but more extreme.	60
Many-Regexes	Multiple random regex membership tests.	40
Regex-Deep	A membership test in a nested regex.	45
Regex-Pair	A membership test in one out of two regexes.	40
Regex-Lengths	A regex membership test of bounded length.	40
Different-Prefix	An equality of two concats with different prefixes.	60

3.5.2 *Transformed*

Table 3.4 lists instances the we 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 created by applying the *Fuzz* and then *Graft* transformers to `z3-regex-1.smt2`.

The *Amazon* sub-suite contains 472 instances derived from two seeds supplied by our industrial collaborators. The *Regex* sub-suite is seeded by the Z3str2 regex test suite [8], which contains 42 instances. Through cumulative transformations we expanded the 42 seeds to 7,551 unique instances. The *Sanitizer* sub-suite came from five industrial e-mail

address and IPv4 sanitizers.

Table 3.4: The *Transformed* suite.

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

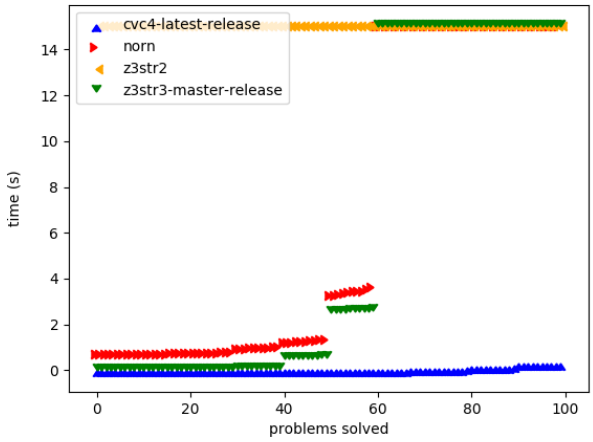
3.6 Results

`stringfuzzg`'s and `stringfuzzx`'s ability to produce scaling instances helped uncover several implementation issues and performance limitations in Z3str3. Varying only one property of an instance (e.g. depth in *Concats-Balanced*) allowed us to see how the solver's performance depends on that property only.

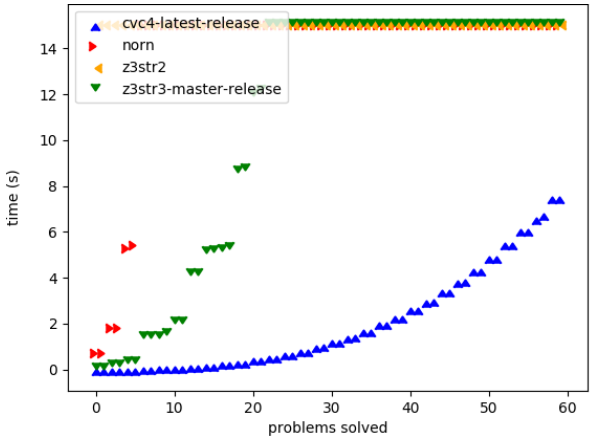
We found several sub-suites on which one solver performed poorly, but not others. They are *Concats-Balanced*, *Concats-Big*, *Concats-Extracts-Small*, and *Different-Prefix*.[1] Fig. 3.2 shows the suites that were uniquely difficult for CVC4. Fig. 3.1 shows the suites that were uniquely difficult for 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™ i7-6700 CPU (3.40GHz).

The Z3str3 team identified performance issues and opportunities for a new heuristic by examining Z3str3's execution traces on the instances in the *Concats-Big* suite. In particular, Z3str3 does not make full use of its solving context (e.g. when 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.

[1]We present only the results that made one solver perform poorly and not others, but results for the full suite are available on our website[6].

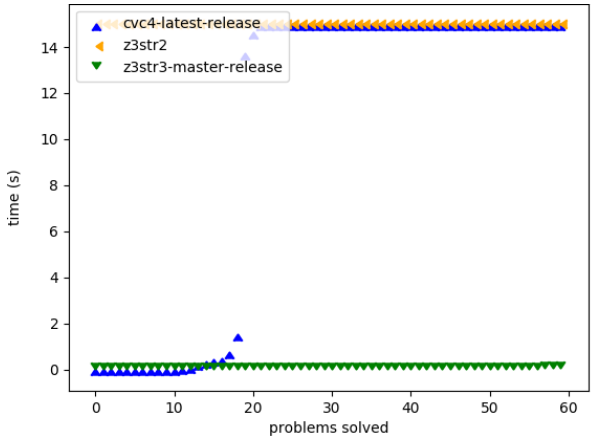


(a) Performance on *Concats-Balanced*

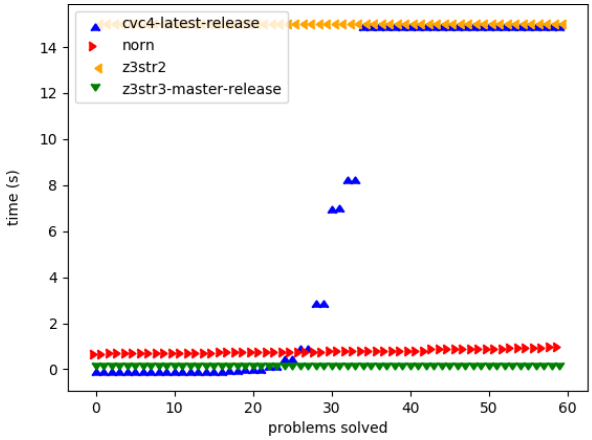


(b) Performance on *Concats-Big*

Figure 3.1: Instances hard for Z3str3



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



(b) Performance on *Different-Prefix*

Figure 3.2: Instances hard for CVC4

3.7 Bugs Fixed in Z3str3

We also found and reported two different implementation bugs in Z3str3 as a result of testing with the *Generated* scaling suites *Lengths-Long* and *Concats-Big*.

The first bug we reported involved interoperability between the theory of integers and the theory of strings. When Z3str3 searched for integer length values to check a model for consistency, it didn't take advantage of an available shortcut to check for constants first. We found this bug by testing Z3str3 with *Concats-Big*. Because the number of variables grows in each instance in the suite, so does the model size, and so does the number of checks that Z3str3 was making to determine consistency. This bug was addressed in pull request #1147 (`https://github.com/Z3Prover/z3/pull/1147`), and was fixed between the commits `6308636` and `3865c45`.

The second bug we reported involved length-testing during model construction. When Z3str3 was trying to compute a length for a variable, it tested possible lengths using linear search, instead of binary search. We found this bug by testing Z3str3 with *Lengths-Long*. Because the length constraints in each instance grew very large, the solving time also grew with it. The Z3str3 team fixed this bug between the commits `66bc68f` and `7b536e9`.

Chapter 4

StringBreak

`stringbreak` is an executable provided by StringFuzz. It generates an SMT-LIB instance that takes a long time for a given solver to solve. It takes a solver command as input[1], and uses a genetic algorithm to mutate a seed instance into another that takes a longer time for the solver to solve. The genetic algorithm and its components are described further in Sect. 4.1.

To run `stringbreak` (assuming CVC4 is installed), execute this command:

```
stringbreak "cvc4 --lang smt2.5 --strings-exp" --trace
```

`stringbreak` can also select for specific answers, and can accept a seed problem. For example, the following script generates a random instance with `stringfuzzg`, and then tries to mutate it to a SAT instance that takes a longer time to solve:

```
stringfuzzg random-ast --meaningful > random.smt
stringbreak "cvc4 --lang smt2.5 --strings-exp" -a SAT -s random.smt
```

Currently `stringbreak` is random in its mutations, but a future enhancement could allow its user to control the allowed mutations.

[1]The command must be such that the solver expects its input on the standard input channel. Some solvers cannot do this.

4.1 Genetic Algorithm

StringBreak implements a genetic algorithm to mutate a seed string SMT instance into one that takes longer to solve. Fig. 4.1 shows the pseudocode of the algorithm.

```
1: function SIMULATE(seed, command, numGenerations, worldSize)
2:     P ← {seed} // Initially, the population is a list of one instance: the seed
3:     i ← 0
4:     while i < numGenerations do:
5:         P ← REPOPULATE(P, worldSize)
6:         F ← JUDGE(P, command)
7:         P ← KILL(P, F)
8:         i ← i + 1
9:     return P
```

Figure 4.1: Pseudocode of the StringBreak genetic algorithm

This algorithm implements evolution by simulating "natural" selection over a population of SMT-LIB instances. At each iteration (i.e. a generation) of the algorithm the population goes through two phases described below: repopulation, and culling.

4.1.1 Repopulation Phase

The REPOPULATE function adds new instances to the population using the reproduction function until the population limit (specified by an input flag to `stringbreak`) is reached. Fig. 4.2 shows the repopulation and reproduction functions. The SPAWN function simulates vegetative reproduction: every child instance is a random mutation of one parent instance. The random mutations are picked from among 4 possible ones: the `stringfuzzg` transformers *Reverse* and *Translate*, and two simple mutators that add or remove a random constraint.

```
1: function REPOPULATE(P, worldSize)
2:     space ← worldSize − P.size
3:     children ← {}
4:     while space > 0 do
5:         children.append(SPAWN(P))
6:         space ← space − 1
7:     return P + children
```

```
1: function SPAWN(P)
2:     parent ← random instance from P
3:     return RANDOMLYMUTATE(parent)
```

Figure 4.2: Pseudocode of the repopulation phase

4.1.2 Culling Phase

In the culling phase of the genetic algorithm, instances are ordered according to a fitness function, and all except the first 4 in the order are removed from the population. Fig. 4.3 shows the pseudocode for the culling phase.

```
1: function JUDGE(P, command)
2:     F ← {}
3:     for all I ∈ P do
4:         F.append(GETFITNESS(I, command))
5:     return F
```

```
1: function KILL(P, F)
2:     sorted ← ORDERBYFITNESS(P, F)
3:     survivors = sorted[0 : 3] // Keep the best 4 instances
4:     return survivors
```

Figure 4.3: Pseudocode of the culling phase

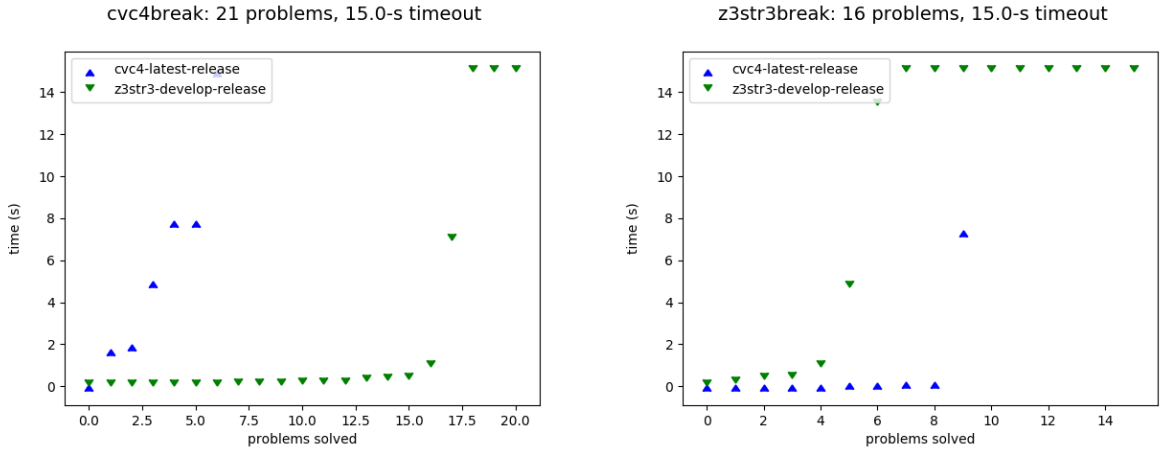
The GETFITNESS function runs the solver command on the given instance and computes the instance's fitness. The fitness is a tuple of the following parameters: solve time, the solver's answer (i.e. SAT or UNSAT), and the instance's size. The solve time is a median solve time from among several invocations of the solver on the instance. We did this to smooth out any irregularities in solver run times.

The ORDERBYFITNESS function sorts the population of instances by their fitness. Because fitness is a multivalued property, this function sorts the population on several values in the following order: by size (ascending), and then by solve time (descending). In the resulting order, instances with the longest run time and shortest size are first. This function favours instances that take the longest time to solve first. Once run times reach the timeout limit and are equal, the function favours smaller instances.

4.2 Results

We ran `stringbreak` on CVC4 and Z3str3. All simulations were conducted in series 15 times, each with a maximum timeout of 3 seconds per instance, for 25 generations. The maximum world size was 8 instances, and the upper limit of assertions per instance was 6.

We then ran CVC4 and Z3str3 on the generated instances. Fig. 4.4 shows the performance of both solvers on the instances that the simulations discovered. All the experiments were run on a macOS High Sierra™ laptop computer with 16GB of RAM and an Intel® Core™ i7 CPU (2.30GHz).



(a) Results on instances evolved for CVC4 (b) Results on instances evolved for Z3str3

Figure 4.4: Performance of CVC4 and Z3str3 on instances evolved by `stringbreak`

Chapter 5

Conclusions

We have shown StringFuzz: a tool for string SMT solver developers to test their solvers, find defects in them, and discover performance issues in them. Our motivation at the outset of this thesis was to give solver developers a means to generate their own instances, and to have control over the properties (like satisfiability, structure, etc.) of those instances.

We demonstrated the utility of StringFuzz by presenting the bugs we discovered with its help in Z3str3. We also showed a suite of crafted scaling instances that we published, and showed how the suite helped reveal performance issues unique to each of Z3str3 and CVC4.

Automating and structuring the process of instance generation has helped us systematically test solvers and analyse their performance. We are confident that solver developers will likewise use our tools to analyse and improve their solvers. We hope that the filter design of StringFuzz will enable others to easily integrate it into their own analysis processes, and we also hope that its modular architecture will enable others to extend it in the future.

References

[1] CVC4 regression test suite. https://github.com/CVC4/CVC4/tree/master/test/regress.

[2] Decision procedures – an algorithmic point of view. http://www.decision-procedures.org.

[3] Input language - Z3Str3 string solver 1.0.0 documentation. https://zyh1121.github.io/z3str3Docs/inputLanguage.html.

[4] Kaluza benchmark suite. http://webblaze.cs.berkeley.edu/2010/kaluza/.

[5] Stringfuzz instance suites. http://stringfuzz.dmitryblotsky.com/problems/.

[6] Stringfuzz source code, benchmark suites, and suplemental material. http://stringfuzz.dmitryblotsky.com.

[7] Strings - CVC4. http://cvc4.cs.stanford.edu/wiki/Strings.

[8] Z3str2 test suite. https://github.com/z3str/Z3-str/tree/master/tests.

[9] Z3str3 test scripts. https://github.com/Z3Prover/z3/tree/master/src/test.

[10] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukáš Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. Norn: An SMT solver for string constraints. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 462–469, Cham, 2015. Springer International Publishing.

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

[12] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In Stewart and Weissenbacher [22], pages 55–59.

[13] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. Stringfuzz: A fuzzer for string solvers. In *Computer Aided Verification*, page **TBD**, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg. *(Paper accepted for publication: http://cavconference.org/2018/accepted-papers/).*

[14] Richard Bonichon, David Déharbe, Pablo Dobal, and Cláudia Tavares. SMTpp: preprocessors and analyzers for SMT-LIB. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories*, SMT 2015, 2015.

[15] Robert Brummayer and Armin 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.

[16] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of sat and qbf solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[17] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[18] Leonardo de Moura and Nikolaj Björner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[19] Scott Kausler and Elena 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.

[20] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan 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.

[21] Bertrand Meyer, Luciano Baresi, and Mira 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*. ACM, 2013.

[22] Daryl Stewart and Georg Weissenbacher, editors. *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. IEEE, 2017.

[23] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Sergio Flesca, Sergio Greco, Giovambattista Ianni, and Nicola Leone, editors, *Logics in Artificial Intelligence*, pages 308–319, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[24] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: a Z3-based string solver for web application analysis. In Meyer et al. [21], pages 114–124.

APPENDICES

Appendix A

StringFuzz and StringBreak

This appendix is an archive of source code for StringFuzz and StringBreak. The file name of this archive is "stringfuzz.zip".

If you accessed this thesis from a source other than the University of Waterloo, you may not have access to this file. You may access it by searching for this thesis at http://uwspace.uwaterloo.ca.

Glossary

SMT solver A software tool that can solve instances of the SMT problem. 1, 7, 30

decision problem A problem that can be framed as a yes-or-no question about its input variables. 1

filter A program that accepts a stream of input and produces a stream of output. 14

predicate A function mapping its inputs to a Boolean value. 7

problem instance A specific example of a given problem. 1, 2

regular expression A string that describes a regular language. xii, xiii, 4, 6

regular language A language recognized by a finite automaton. 30

stream A sequence of data elements (e.g. bytes) that are made available over time. 30

string A sequence of (e.g. alphanumeric) characters. xiii, 1, 4, 6, 11, 30

string SMT solver An SMT solver that can solve SMT problem instances that have constraints on string (and sometimes regex) variables. 1, 2, 6, 7, 10

word equation An equality among string variables and concatenations thereof. 9