ZaligVinder: A generic test framework for string solvers

Kulczynski, Mitja; Manea, Florin; Nowotka, Dirk; Poulsen, Danny Bøgsted

Published in:
Journal of Software: Evolution and Process

DOI (link to publication from Publisher):
10.1002/smr.2400

Creative Commons License
CC BY 4.0

Publication date:
2021

Document Version
Publisher's PDF, also known as Version of record

Link to publication from Aalborg University

Citation for published version (APA):
Kulczynski, M., Manea, F., Nowotka, D., & Poulsen, D. B. (2021). ZaligVinder: A generic test framework for string solvers. *Journal of Software: Evolution and Process*. https://doi.org/10.1002/smr.2400

Received: 10 January 2021 | Revised: 13 July 2021 | Accepted: 28 September 2021

DOI: 10.1002/smr.2400

RESEARCH ARTICLE - TECHNOLOGY

Journal of Software: Evolution and Process WILEY

ZaligVinder: A generic test framework for string solvers

Mitja Kulczynski[1] | Florin Manea[2] | Dirk Nowotka[1] | Danny Bøgsted Poulsen[3]

[1]Department of Computer Science, Kiel University, Kiel, Germany

[2]Department of Computer Science, University of Göttingen, Göttingen, Germany

[3]Department of Computer Science, Aalborg University, Aalborg, Denmark

Correspondence
Mitja Kulczynski, Department of Computer Science, Kiel University, Kiel, Germany.
Email: mku@informatik.uni-kiel.de

Abstract

The increased interest in string solving in the recent years has made it very hard to identify the right tool to address a particular user's purpose. Firstly, there is a multitude of string solvers, each addressing essentially some subset of the general problem. Generally, the addressed fragments are relevant and well motivated, but the lack of comparisons between the existing tools on an equal set of benchmarks cannot go unnoticed, especially as a common framework to compare solvers seems to be missing. In this paper, we gather a set of relevant benchmarks and introduce our new benchmarking framework to address this purpose.

KEYWORDS
analysis of string solvers, string solving benchmarks, test framework for string solvers

1 | INTRODUCTION

Strings (also known as words or sequences of symbols) are a fundamental data type, occurring in almost all domains of Computer Science. Not only are they implemented in almost every modern programming language, but they are also of the highest importance in areas ranging from compiler construction to bioinformatics, from database theory (i.e., query processing) to algorithmic learning theory (e.g., descriptive patterns computation), as well as software analysis and verification. For instance, if we consider the area of programming and programming languages, strings are the way users input data to a program, and they are the primary means for relaying information back to the user. Usage of strings in programs can however also be dangerous: number one of the Open Web Application Security Project (OWASP) top 10 security flaws is *injection attacks* that are caused by improper usage of strings. In such an attack, the attacker runs an application with a malicious input that, due to improper string handling, ends up being interpreted as which the application executes. Consequently, the attacker may damage or get unauthorized access to the data handled by the application. Similarly, another one of the top 10 security flaws of OWASP is Cross Site Scripting (XSS) attacks. Here, an application inserts an attackers unsanitized input into a HTML page rendered for a target user. The browser of the target displays the rendered HTML page and executes any JavaScript present in the attackers input. A cross-site scripting attack can be used to steal the cookies of a target and ultimately take over their session leading to impersonation attacks.

Given strings are so prevalent in computer science and improper string handling is the culprit of many security issues, it should come as no surprise that researchers incorporated string reasoning into their program analyzing tools: Initially, this resulted in a number of dedicated string constraint solvers (e.g., Hampi,[1] NORN,[2] Stranger,[3] and ABC[4]) and eventually established general-purpose SMT solvers (e.g., CVC4[5] and Z3[6,7]) started integrating strings into their supported logics.

A relatively new addition to the string solving community is our own tool Woorpje,[8,9] which is a result of combining SAT solving, word-theoretic results, and classical SMT-development ideas. During development of Woorpje we realized that the community missed means for comparing solvers. In particular, 1. there was no general collection of standard string solving benchmarks, and it seemed very hard to identify the relevant benchmarks for string solving tasks, without considering a high amount of literature, and collecting from each paper the input data they used to test their tools; 2. benchmarks were mostly uncategorized as to which subset of string constraints they contained; 3. benchmarks from different tools used different input formats or slight variations of the same core format (SMT-LIB); 4. the satisfiability/unsatisfiability verdicts provided by the tools used in the literature were wrong on many benchmarks; 5. benchmarks were huge and really hard to get an overview, in which one solver was better than the other; and, finally, 6. existing comparison frameworks[10] focused on comparing the speed of solvers rather than comparing their verdicts and locating instances where solvers produce different results.

Naturally, this is a result of a field growing organically with lack of standardization. Efforts have gradually converged so that we now have a unified string logic standard as part of SMT-LIB—addressing items (partly) 2 and 3. We however still have a large back-catalog of old nonstandardized benchmarks that should be updated to the new standard. Furthermore, the standard does not solve the overarching problem that many benchmark instances satisfiability verdict in the literature are incorrect.

In this paper, we address items 1, 4, 5, and 6 by 1. collecting several benchmarks from the literature in a single repository, 2. identifying which string operation the different benchmarks use the most thus helping developers identify which set to use when working on different features, and 3. implementing the comparison framework ZaligVinder (Figure 1). ZaligVinder allows running the collected benchmarks on different string solvers and gather their results in a database. It allows running all of the established string solvers on the benchmark sets. ZaligVinder includes a graphical overview of the results to ease the comparison of the results and 4. to overcome the problem that existing benchmarks verdicts are wrong ZaligVinder includes a cross-validation mechanism for asserting what is the correct result for inputs with unknown verdicts.

This work is an extension of Kulczynski et al[11] and is organized as follows: We introduce the input language state-of-the-art string solvers support and give a short introduction into the field of string constraints in Section 2. The next section summarizes the set of collected benchmarks. With respect to Kulczynski et al,[11] we extended this set even further reaching a total of 114.475 instances split over 152 different benchmark sets. Section 4 is devoted to our tool ZaligVinder. We give an introduction on how to perform an analysis using ZaligVinder and how to analyze the resulting data with our tool by explaining the different techniques. In Section 5, we perform two cases studies on how to use ZaligVinder for a specific analysis of a string solver and how to extend the tool to use the resulting data for further postprocessing steps. Before concluding, we analyze our framework and also the collected benchmarks based on several research questions, as well as briefly discussing related work.

2 | STRING CONSTRAINTS

In the following, we give a brief overview of string constraints. For more details, see, for example, Zheng et al[6] and Liang et al.[12] Here, we present the atomic constraints, but note that they are combined using propositional logic operators.

String solvers solve constraints involving constants (and variables) of four different sorts: strings, regular expressions, integers, and Boolean. Strings are sequences of characters from two distinct sets, namely, variables $\mathcal{X}$ and constants Σ. Two strings can be concatenated using a concatenation operation. Regular expressions are defined in the classical sense ranging over the sets of strings.

The complexity of string solving depends on what kind of constraints are included. The most elementary constraint is a pure equality between strings—called a word equation. A word equation has the form $u \doteq v$ where u and v are strings and asks to unify the two sides of the equality by replacing all variables $x \in \mathcal{X}$ by a string $w_x \in \Sigma^*$. The mapping h unifying the two sides is called a solution to $u \doteq v$. A constraint system involving only a conjunction of word equations is known to be decidable.[13]

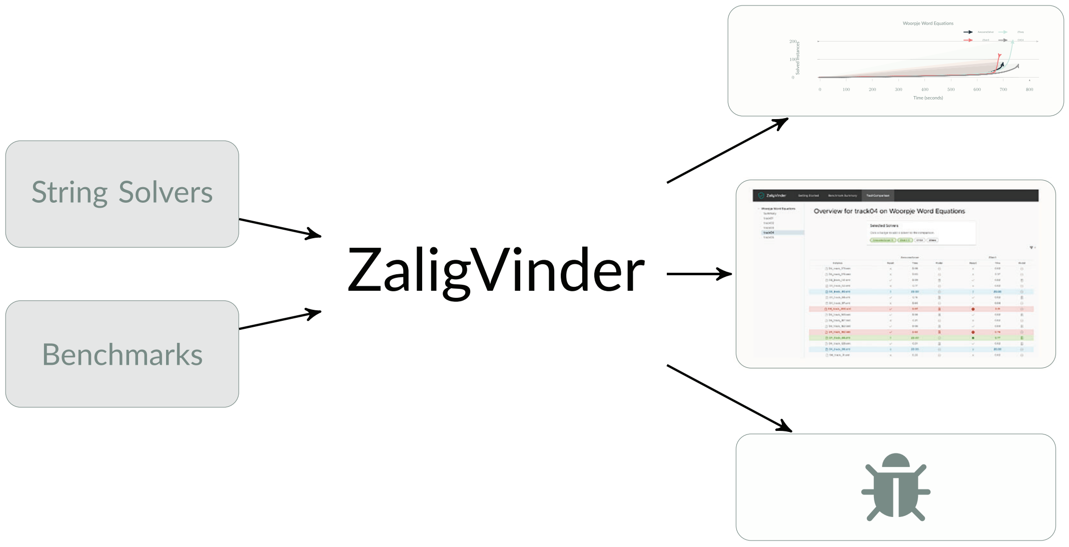


FIGURE 1 ZaligVinder overview

As an example, consider the word equation $\mathtt{a}x\mathtt{ab}y\mathtt{ab}x\mathtt{b} \doteq y\mathtt{ba}x\mathtt{aabb}x$ where $x, y \in \mathcal{X}$ are variables and $\mathtt{a}, \mathtt{b} \in \Sigma$ are constants. Mapping x to $\mathtt{b}$ and y to $\mathtt{aba}$ forms a solution to the equation. We get $\mathtt{abababaabbb} = \mathtt{abababaabbb}$.

A collection of word equations can be enriched by adding constraints over the length of strings. String solvers therefore have to support the length function $\mathtt{len}: (\mathcal{X} \cup \Sigma)^* \to \mathbb{Z}$, which counts the number of characters within a string (keeping in mind that the length of a variable $\mathcal{X}$ is obviously an integer variable itself). A typical length constraint could be that the length of y has to be at most twice the length of x, that is, $\mathrm{len}(x) \cdot 2 \geq \mathrm{len}(y)$. The length constraints can involve arbitrary arithmetic expression over the length of variables, but whether solving word equations with length constraints is decidable or not remains open. However, it is worth mentioning the effort to reach an answer to this question, which is pursued also in, for example, Day et al,[14] where other related fragments of word equations are analyzed.

Adding regular expression constraints amounts to a string solver supporting classical operators on regular languages like union, intersection, and the Kleene star. Furthermore, it must be able to determine if a string belongs to a regular language denoted `in_re`.

Regular expressions including variables and membership constraints are also an active topic on their own rights and still offer fragments with an unknown decidability status (see, e.g., Berzish et al[15]).

A recent trend supplements word equation-, length-, and regular expression constraints with higher order string operations. The `at` operation takes a string w and a positive integer i, and returns the character at position i of w if the length of w at least i and the empty word otherwise. A similar function that asks not only for a specific character but also for a substring is `substr`, where `substr`(ℓ, i, w) returns a substring of length ℓ starting at position i within w (if the length of w is larger than $i + \ell$ and the empty word otherwise). The predicates, `prefixof`, `suffixof`, and `contains`, are also defined for words u, v. If u is contained in (resp. prefix of, or suffix of) v, then `contains`(u, v) (resp., `prefix of`(u, v), or `suffixof`(u, v)) returns true. The function `indexof` for parameters u, v, and i returns the index of the first occurrence of the v within u after position i. If there is no v present, it returns -1. The function `replace`(u, v, r) replaces the first occurrence of v in u by r, or returns u if u does not contain v.

The unary predicate `is_digit` holds all characters (sequences of length 1) with a decimal digit representation. This is a code within the range 0x0030 and 0x0039. The functions to_int and from_int convert strings to integers and vice versa. If the abovementioned function `is_digit` evaluates a string w to a positive integer, to_int returns exactly this representation, and otherwise, -1. from_int returns for a non-negative integer its string representation. For negative integers, the function simply returns ϵ, the empty word. As an example for a larger string constraint, consider $\mathtt{a}x\mathtt{ab}y\mathtt{ab}x\mathtt{b} \doteq y\mathtt{ba}x\mathtt{aabb}x \wedge \mathrm{len}(x) \cdot 2 \geq \mathrm{len}(y) \wedge \mathtt{in}_{\mathtt{r}}\mathtt{e}(x, \{\, a \,\}^*) \wedge \mathtt{contains}(ab, y)$, which again has the solution mapping x to $\mathtt{b}$ and y to $\mathtt{aba}$.

According to a recently published survey on string constraint solving by Amadini,[16] all prominent state-of-the-art string solvers are integrated into SMT solvers, which is why we also focus on SMT style solvers within this work. The key insight of SMT solvers is to incorporate theories (e.g., the theory of string constraints as introduced in this section) into the classical DPLL algorithm[17] by Davis et al to support a more expressive input language than pure propositional logic formulae. To this extent, most modern SMT solvers implement an extension of this well-known algorithm called DPLL(T).[18] The core idea is the interaction of the theory solvers, which decompose their parts of the input formula into a propositional logic satisfiability problem by building an over approximation, asking the core SAT solver about the satisfiability and a model, respectively, of their abstraction, and then checking feasibility of the original formula.[19] Therefore, for example, word equations get their attention within a subsolver handling exactly this equality.

SMT solvers read an input format called SMT-LIB.[20] Recently, the SMT-LIB initiative also introduced the first standard for string constraints.[21] In Figure 2, we briefly review the SMT-LIB syntax for string constraints. The semantics are directly induced by the aforementioned definitions in this section. Note, that string constraints do not contain quantifiers. The SMT-LIB standard introduces two commonly used theories of string constraints. That are `QF_S` called *quantifier-free theory of strings* and `QF_SLIA` called *quantifier-free theory of string and linear arithmetic*.

$$
\begin{array}{lll}
F & ::= & Atom \mid (\mathtt{and}\ F\ F) \mid (\mathtt{or}\ F\ F) \mid (\mathtt{not}\ F) \\
Atom & ::= & (=\ t_{str}\ t_{str}) \mid A_{int} \mid A_{ext} \mid (\mathtt{str.in\_re}\ t_{str}\ RE) \\
A_{int} & ::= & (=\ t_{int}\ t_{int}) \mid (<\ t_{int}\ t_{int}) \\
A_{ext} & ::= & (\mathtt{str.contains}\ t_{str}\ t_{str}) \mid (\mathtt{str.prefixof}\ t_{str}\ t_{str}) \mid (\mathtt{str.suffixof}\ t_{str}\ t_{str}) \\
t_{int} & ::= & m \mid \mathsf{x} \mid (\mathtt{str.len}\ t_{str}) \mid (+\ t_{int}\ t_{int}) \mid (*\ m\ t_{int}) \mid (\mathtt{str.indexof}\ t_{str}\ t_{str}\ t_{int}) \mid (\mathtt{str.to\_int}\ t_{str}), \\
 & & \qquad \text{with } m \in \mathbb{Z} \text{ and } \mathsf{x} \in \mathcal{X} \\
t_{str} & ::= & \text{“}w\text{”} \mid \mathsf{x} \mid (\mathtt{str.{+}{+}}\ t_{str}\ t_{str}) \mid (\mathtt{str.from\_int}\ t_{int}) \mid (\mathtt{str.replace}\ t_{str}\ t_{str}\ t_{str}) \mid (\mathtt{str.at}\ t_{str}\ t_{int}) \mid \\
 & & (\mathtt{str.substr}\ t_{str}\ t_{int}\ t_{int}), \\
 & & \qquad \text{with } w \in A^* \text{ and } \mathsf{x} \in \mathcal{X} \\
RE & ::= & \text{“}w\text{”} \mid (\mathtt{re.none}) \mid (\mathtt{re.{+}{+}}\ RE\ RE) \mid (\mathtt{re.comp}\ RE) \mid (\mathtt{re.union}\ RE\ RE) \mid (\mathtt{re.inter}\ RE\ RE) \mid (\mathtt{re.*}\ RE), \\
 & & \qquad \text{with } w \in A^*
\end{array}
$$

FIGURE 2 Basic SMT-LIB syntax for string constraints

`QF_S` contains all aforementioned atoms and operations but the ones reasoning over integer arithmetic. These atoms and operations are included within the `QF_SLIA` logic. Therefore, `QF_S` is a strict subset of `QF_SLIA`.

3 | BENCHMARK SETS

We searched the literature to identify existing benchmarks, used by the different string solvers. These benchmarks have been incorporated into our tool ZaligVinder. We first overview their origin and briefly discuss which constraints are used within each of them and summarize the details in Table 1.

3.1 | Leetcode

Wei-Cheng Wu used the concolic execution engine Conpy[22] and PyExZ3[23] to generate 43 different sets consisting of a total of 2666 instances. The basis are interview questions from the website https://leetcode.com. The length constraint heavy set consists of challenging, 881 satisfiable and 1785 unsatisfiable instances forms a good regression test to make sure the interleaving theories of an SMT solver behave correctly.

3.2 | PyEx

Reynolds et al[24] used the tool PyEx[25]—a symbolic executor for Python programs—to generate a set of 25,421 benchmarks. They used 19 target functions sampled from four popular Python packages to generate the resulting benchmark set.

TABLE 1 Summary of the collected benchmarks including the most used string operations

			Most used string operations (Avarage occurence)				
Benchmark Name	#Sets	#Instances	1st	2nd	3rd	4th	5th
Leetcode	43	2666	`len` (44.31)	`at` (25.3)	`indexof` (10.06)	`substr` (9.2)	++ (2.93)
PyEx	57	25421	`len` (361.29)	`substr` (349.18)	`indexof` (334.44)	`contains` (34.17)	++ (20.46)
AppScan	1	8	`to.re` (9.5)	++ (1.75)	`suffixof` (1.12)	`in.re` (1.0)	`indexof` (1.0)
AutomatArk	2	19979	`to.re` (25.2)	`in.re` (2.25)			
BanditFuzz	1	357	`indexof` (3.31)	`len` (2.86)	`at` (2.36)	`replace` (2.21)	`substr` (2.1)
Cashew	1	394	++ (13.59)	`to.re` (2.81)	`in.re` (2.3)	`len` (1.78)	
JOACO	1	94	`to.re` (26.26)	++ (8.39)	`in.re` (0.81)	`len` (0.13)	`to.int` (0.05)
Kaluza	4	47284	`to.re` (33.53)	++ (12.72)	`len` (6.94)	`in.re` (4.38)	
Kausler	1	120	++ (197.88)	`substr` (10.27)	`len` (1.98)		
Trau Light	1	100	++ (12.0)				
Norn	5	1027	`to.re` (14.88)	`in.re` (4.77)	++ (2.77)	`len` (1.09)	
Pisa	1	12	`contains` (2.25)	++ (1.08)	`substr` (0.83)	`replace` (0.83)	`len` (0.67)
Sloth	1	40	`in.re` (0.97)	++ (0.93)	`to.re` (0.82)	`replace` (0.42)	`replaceall` (0.17)
Stranger	1	4	`to.re` (78.0)	++ (13.0)	`in.re` (1.0)		
StringFuzz	17	1065	++ (66.2)	`to.re` (41.39)	`at` (33.61)	`len` (32.05)	`in.re` (1.38)
StringFuzz Regex Generated	7	4170	`to.re` (160.99)	`in.re` (6.25)	`len` (0.37)		
StringFuzz Regex Transformed	2	10682	`to.re` (1.88)	`in.re` (1.41)	`len` (0.74)	`to.int` (0.2)	++ (0.15)
Woorpje	5	809	++ (43.04)	`len` (0.95)			
Z3str2	1	243	++ (1.19)	`len` (0.6)	`to.re` (0.38)	`contains` (0.29)	`in.re` (0.25)
Total	152	114475	`len` (75.36)	`substr` (68.96)	`indexof` (66.94)	`to.re` (25.46)	++ (11.19)

3.3 | AppScan

Zheng et al[26] generated a second set of benchmarks using the output of security warnings generated by IBM Security AppScan Source Edition.[27] They ran the tool on popular websites to obtain traces of program statements that were translated into SMT formulae. The traces reflect potentially vulnerable information flows and therefore represent common real-world constraints. The set consists of eight instances containing string functions and disequality constraints over strings.

3.4 | AutomatArk

In the need of regular expression heavy benchmarks, Berzish et al[15] generated SMT-LIB benchmarks based on real-world regular expression queries collected by Loris D'Antoni. The set consists of two tracks—a simple and a hard track—having a total of 19,979 instances. The simple track holds instances having a single regular expression membership constraint, and the hard track holds up to five membership constraints over a single variable per instance.

3.5 | BanditFuzz

BanditFuzz[28] is a reinforcement learning driven fuzzing system for SMT solvers. The authors share a set of string benchmarks being generated by using their tool. The resulting 357 instances heavily contain the `indexof` predicate, which makes them unique within our gathered instances.

3.6 | Cashew

Brennan et al[29] used their tool called Cashew to normalize (in terms of their tool) 18,896 by Luu et al[30] extracted benchmarks from the Kaluza benchmark set. By constructing this subset, the authors aimed to eliminate the redundancy in the original set. The set varies between easy and difficult string constraints, with Boolean constraints, without using string operations.

3.7 | JOACO

In order to evaluate their tool JOACO, a tool to detect injection vulnerabilities, Thomé et al[31] created a set of 94 instances based on 11 open-source Java Web applications and security benchmarks used in literature. It displays a variety of instances containing string constraints, regular expressions, and string operations.

3.8 | Kaluza

Saxena et al[32] used their tool Kudzu, a symbolic execution framework for JavaScript, to generate more than 50,000 string solving problems. The instances were obtained by lowering JavaScript operations from real-world AJAX web applications and are available on their website.[33] The instances are build around string constraints, membership in regular languages (given as regular expressions), and inequalities involving length constraints on string variables. Although the size of the formula varies per instance, the variety in the used string operations is rather small. The resulting set of benchmarks was translated by Liang et al[12] into the SMT-Lib format. It uses string, boolean, and linear constraints together with a small amount of string operations.

3.9 | Kausler

Kausler and Sherman[34] generated a set of benchmarks to evaluate string constraint solvers in terms of symbolic execution. The set was brought down to 120 instances by Thomé et al.[31] It contains constraints from eight Java programs via dynamic symbolic execution, aiming for real word application. The set mostly contains Boolean and string constraints without string operations.

3.10 | Light Trau

Within this set of benchmarks generated by Abdulla et al,[35] each instance holds multiple easy, mostly unsatisfiable formulae consisting only of string constraints. The set aims for testing the ability of declaring inputs as unsatisfiable, which is in general harder than finding a solution.

3.11 | Norn

Abdulla et al[2] share a set of five tracks consisting of queries generated during verification of string-processing programs.[36] Each formula is rather small compared with those in other sets of benchmarks but makes heavy use of regular expressions containing Kleene stars. This makes it a challenging one for all solvers.

3.12 | PISA

Zheng et al[26] generated a set of benchmarks using constraints from real-world Java sanitizer methods that were used to evaluate the PISA system.[37] It contains 12 complex instances including multiple different string operations like `indexOf`, `substring` as a result of the Sanitizer structure.

3.13 | Sloth

The string solver Sloth[38] handles the so-called straight line fragment of string constraints, that is, essentially a formula that corresponds to a sequence of program assignments in SSA form including the assertion of regular constraints.[39] Their regression test suite contains 40 instances including the corresponding string operations. However, string solvers like CVC4 and Z3Str3 are not able to handle the present `str.replaceAll` function that is present in seven instances and allows only using 33 out of 40 instance for a comparison between all solvers.

3.14 | Stranger

Yu et al[3] used this set of four real-world PHP web applications to evaluate their tool Stranger—a tool detecting and sanitizing vulnerabilities in PHP applications. Thomé et al[31] manually translated these instances into the SMT-LIB format. The result was a set containing string operations, regular expression membership constraints, and string constraints.

3.15 | StringFuzz

Blotsky et al[40] introduced a tool called StringFuzz to generate and transform SMT-LIB instances of string problems implemented in Python. The authors share a set of benchmarks generated using their tool, which aims to address typical industrial instances, potentially challenging for solvers. The set contains 17 tracks ranging from instances containing pure string constraints to hard to solve regular expression constraints. They aim for generating instances that follow structures hard to handle by some solvers (e.g., tree-like instances).

In 2020, Berzish et al[15] extended the StringFuzz suite by two regex heavy sets having a total of 15,052 instances. The `StringFuzz-regex-generated` set contains randomly fuzzed instances only coping regular expressions and length constraints. The `StringFuzz-regex-transformed` set is the result of a transformation of instances supplied by Amazon Web Services related to security policies, being inspired by real-world input vulnerability violations.

3.16 | Woorpje

Day et al[8] created a set of benchmarks to test the abilities of their tool Woorpje. It contains five tracks with instances containing mostly string constraints but also linear length constraints. Running this set on their competitors revealed its difficulty. The set is generated using several hard involved examples developed in the theoretical study of word equations.

3.17 | Z3str2

This set by the authors of Z3str2[41] was initially generated as a regression test for their tool. Nowadays, it seems to have a broader audience due to the evenly distributed occurrences of multiple string and regular expression predicates. The set consists of a single track holding 243 instances in total.

Overall, this seems to be the largest collection of strings solving benchmarks today. All collected instances are available in our GIT repository at https://github.com/zaligvinder/zaligvinder.

4 | ZALIGVINDER

One of the main focal points is collecting existing benchmarks and setting up a framework for easy comparison of string solvers. In this section, we show how to setup our framework ZaligVinder with the tools (CVC4[5] and Z3Str3[7]) and the collected benchmarks. We also discuss how to extend the number of tools and the benchmark sets. Finally, we show how to start the graphical comparison interface of ZaligVinder and review individual components of our framework.

4.1 | Performing an analysis

After downloading ZaligVinder from https://github.com/zaligvinder/zaligvinder,[1] the first thing one has to do is set up a benchmark-setup file. This is a Python 3 script that sets up which benchmark sets that constitutes the current test setup, selects which solvers to execute on, and a common timeout for each tool.

In Listing 1, we show a basic benchmark-setup file. The file starts with `import`ing helper modules for storage and a voting mechanism for deducing reference results for input models that we do not know the correct categorization of ("satisfied" or "not satisfied") for.

```
import utils
import storage
import voting.majority as voting
from runners.base import TheRunner as testrunner
import summarygenerators
import startwebserver

import tools.z3str3
import tools.cvc4
import tools.z3seq
import models.woorpje

timeout = 30
ploc = utils.JSONProgramConfig ()
voter = voting.MajorityVoter()
verifiers = ["cvc4","z3seq"]

store = storage.SQLiteDB ("Example")

summaries = [ summarygenerators.terminalResult ,
              store.postTrackUpdate]

tracks = models.woorpje.getTrackData ("Woorpje Word Equations")

solvers = {}
for s in [tools.z3str3,tools.cvc4]:
    s.addRunner (solvers)

testrunner().runTestSetup (tracks,solvers,voter,summaries,store,
    ↪ timeout,ploc,verifiers)
startwebserver.Server (store.getDB ()).startServer ()
```

Listing 1: Basic setup script

Listing 1. HTML code

[1]To ease the setup of ZaligVinder, we provide a docker file and instructions within our GIT repository.

The Kaluza benchmark set is known to be an example of wrongly provided answers. In Lines 4 and 5, we import the Z3STR3, Z3SEQ, and CVC4 driver module and a collection of models called woorpje. In 10 through 12, we set the timeout, pick a configuration manager, a voting mechanism including the verification procedure, and specify the validating solvers. The configuration manager (utils.JSONProgramConfig) object is used to locate binaries on the host machine. The storage mechanism (an SQLite database) is setup in 18, and in 20, we set up a number of functions that will be run after each track. In particular, we will generate an output to the terminal and perform a postTrackUpdate to the storage. The later one updates the results according to the validation result. In Line 25 through 25, we pick up all the tracks (lists of input files) from the models.woorpje module and collects them under the common name "Woorpje Word Equations" and put the Z3STR3 and CVC4 solvers into the collection of solvers that will be executed. In Line 29, we execute this entire benchmark. Finally, after having executed the benchmarks, a webserver is started where the results can be inspected.

4.1.1 | Adding benchmark input files

In ZALIGVINDER, we represent an input file to the tools as a TrackInstance object that consist of a name, a path to the input file, and an expected result (True for satisfiable, False for not satisfiable and None for unknown). The TrackInstance objects are gathered into Track objects having a name, the TrackInstance objects, and a benchmark name. This benchmark name is useful for grouping several tracks under a common name for presentation purposes.

To make new input files available to ZALIGVINDER, a PYTHON submodule should be added, namely, the MODELS module of ZALIGVINDER. The initialization file (__INIT__.PY) of the submodule should contain a function getTrackData accepting a single parameter being the name we want to group the tracks under. A prototypical implementation of such an __INIT__.PY can be seen in Listing 2. It simply iterates over all files in the directory, and if the filename suggests it being a SMT instance, then a track instance is made. Finally, a list containing only a single Track is returned.

```
import os
import utils
dir_path = os.path.dirname(os.path.realpath(__file__))

def getTrackData (bname = ""):
    filest = []
    for root, dirs, files in os.walk(dir_path, topdown=False):
        for name in files:
            if name.endswith (".smt"):
                filest.append(utils.TrackInstance(name,os.path.join(
                    ↪ root,name)))

    return [utils.Track("Track 1",filest,bname)]
```

Listing 2: Prototypical file to add input files to ZaligVinder

Listing 2. HTML code

4.1.2 | Adding a solver

In ZALIGVINDER, the driver interface is encapsulated into submodules of the TOOLS module. The driver module must expose one function addRunner accepting a dictionary and add a runnable under a solver specific name. The runnable added to this dictionary must accept four parameters eq,timeout,ploc,wd where eq is the path to an input file, timeout is the user specified timeout, ploc is the configuration manager from Listing 1 and wd is a directory the solver can use for temporary storage. Here, we will not go into details about how each individual solver is run but show a standard structure for the solver file in Listing 3. What should be mentioned is that the runnable (run function) returns a utils.Result object that encapsulates the verification result, the time it took, whether the solver timed out, the standard output stream from the tool, and the generated model of the tool.

```
import subprocess
import tempfile
import os
import utils
import sys
import timer

def run (eq,timeout,ploc,wd):
    path = ploc.findProgram ("Solvername")

    time = timer.Timer ()
    try:
        out = subprocess.check_output ([path,eq],timeout=timeout).
           ↪ decode().strip()
    except subprocess.TimeoutExpired:
        return utils.Result(None,timeout,True,1)
    except subprocess.CalledProcessError:
        return utils.Result(None,timeout,False,1)

    time.stop()

    if "unsat" in out:
        return utils.Result(False,time.getTime (),False,1,out)
    elif "sat" in out:
        return utils.Result(True,time.getTime(),False,1,out,"\n"
                                            .join(out.split("\n")[1:]))
    return utils.Result(None,time.getTime  (),False,1,out)

def addRunner (addto):
    addto['Solvername'] = run
```

Listing 3: Prototypical Solver file

Listing 3. HTML code

4.1.3 | Using multiple cores

In Listing 1, line 4, we load the runner for handling the execution of a particular solver. In this example, we load our base runner that holds a function to subsequently execute a solver on all instances of an asked track. To shorten the evaluation time, ZaligVinder offers a multicore setup being based on Pythons `multiprocessing` package. Switching to our multicore setup is made as easy as possible. We simply load the multirunner by replacing line 4 by `from runners.multi import TheRunner as testrunner`. `TheRunner`, which now takes an optional integer that corresponds to the amount of cores being used during your analysis. ZaligVinder has now successfully been configured for a multicore run.

4.1.4 | Verification

Many of our gathered benchmark sets do not contain expected associated results. As written above, some of them even contain incorrect classifications. In order to find misclassified instances by a solver, ZaligVinder offers two different techniques: 1. model validation and 2. result voting. Whenever a solver classifies an instance as satisfiable, the string solving guidelines expect the solver to return a valid model. To check the validity of a model, the corresponding variables are substituted into the input formula. The resulting variable-free formula is then asserted and a set of solvers—seen in Listing 1, line 16—verifies its satisfiability. If all verification solvers treat a model as valid, ZaligVinder marks the instances as correctly solved. Because string solvers are not required to produce a proof for nonsatisfiable cases, we can not use a similar procedure as described above. Therefore, ZaligVinder performs a majority voting considering all results returned for an instance. If at least one solver returned a valid model, being verified by the above procedure, we treat an instance as satisfiable. A majority voting is performed between instances being classified as unsatisfiable or unknow resp. timeout; we treat an instances as unsatisfiable if more solvers were able to classify an instances as unsatisfiable. However, an expected result is only set whenever the above conditions are met.

4.2 | Postprocessing

ZaligVinder stores the benchmarking results into an SQLite database as this enables easy postprocessing of the data. Next to a REST API offering relevant data in JSON format, we have implemented a basic graphical user interface (GUI) using this database. Thirdly, we provide a collection of

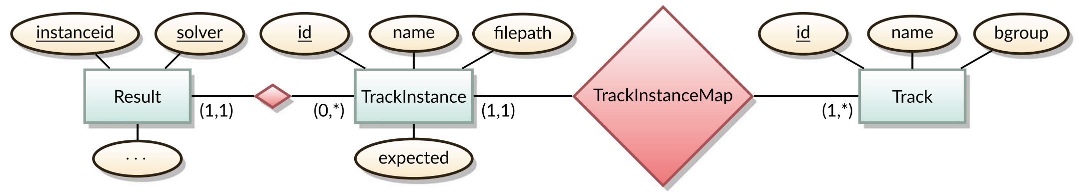


FIGURE 3 Entity relationship diagram of ZaligVinder's database schema. For ease of readability, the `Result` entity omits the attributes `timeouted`, `result`, `time`, `output`, and `model`

`Python3` scripts to generate LATE X plots and tables as well as a Markdown website to review and share the results of a run. The GUI and the API are started by running `python startwebserver.py dbfile`, where `dbfile` is the database file created by ZaligVinder and guide a web browser to https://localhost:8081[2].

4.2.1 | Database

The database model was designed to ease the postprocessing abilities. In Figure 3, we depict our database as an entity relationship diagram.

The database holds four tables, namely, `Result`, `TrackInstance`, `TrackInstanceMap`, and `Track`, whereas `TrackInstanceMap` forms the relationship between `Track` and `Instances`. The `Result`s entity holds a specific result of a solver on a particular instance. We store the running times, the solvers returned result, a flag, whether the solver timeouted, the solvers output on this particular instances, the model—in case of determining the instance as satisfiable, and a verified flag. The result is threefold: It can either be `SAT`, `UNSAT`, or `UNKNOW`. Due to undecidabilty of certain string solving fragments, all string solvers implement incomplete algorithms. Therefore, the result `UNKNOW` requires further investigation: It either indicates an early give up to finding a solution or a timeout. The later fact can be observed by considering the `timeouted` flag that is only true whenever a solver got an external kill signal. The verified flag indicates whether a return model is valid. However, for an instance that is declared unsatisfiable, we fall back to the majority voting as described in Section 4.1.4.

The `TrackInstance` entity holds all registered formulae being identifiable by an `id`. The instance holds a name—usually the file name of the input formula, a location path within the file system, and the expected result of the instance. If this result is not known prior to the run, we used the same procedure as described in Section 4.1.4 to fill the expected result. Thus, it cannot be fully trusted.

The `Track` entity consists of track names and their corresponding benchmark groups. The instances of each track are linked via the relation `TrackInstanceMap` using the corresponding primary keys.

4.2.2 | REST API

The API offers an easy way to processing the data generated by ZaligVinder externally. In Table 2, we review the basic functionalities of the API.

We offer endpoints to the most general information being present within the database such as solver information, track and instance details, as well an refurbished look on the results of the considered run, for example, summary data of a run. Next to these features, we offer an access to various chart representations—namely, cactus, distribution, and scattered plots—of the results. The user can either access the plain data or an image rendered by using `matplotlib`. An unrelated but interesting view in regards to string solving is the ability of accessing a chart visualizing the most used string functions within a run by visiting `chart/keywords`.

As ZaligVinder was build to be highly extensible the implemented set of API endpoints forms only a start without limits to enhancing it.

4.2.3 | GUI

To ease the comparison of different solvers, we implemented a basic GUI (see Figure 4) using the data of the created SQLite database. The GUI offers two categories to review the results: a benchmark summary view and a tool comparison view. The summary view allows identification of soundness issues, solver crashes, as well as performance comparison of competing solvers. The tool comparison view allows detailed inspection of specific cases for each solver individually.

[2]A demonstration and further explanations are available at https://zaligvinder.github.io

TABLE 2 Rest API entry points and their functionalities

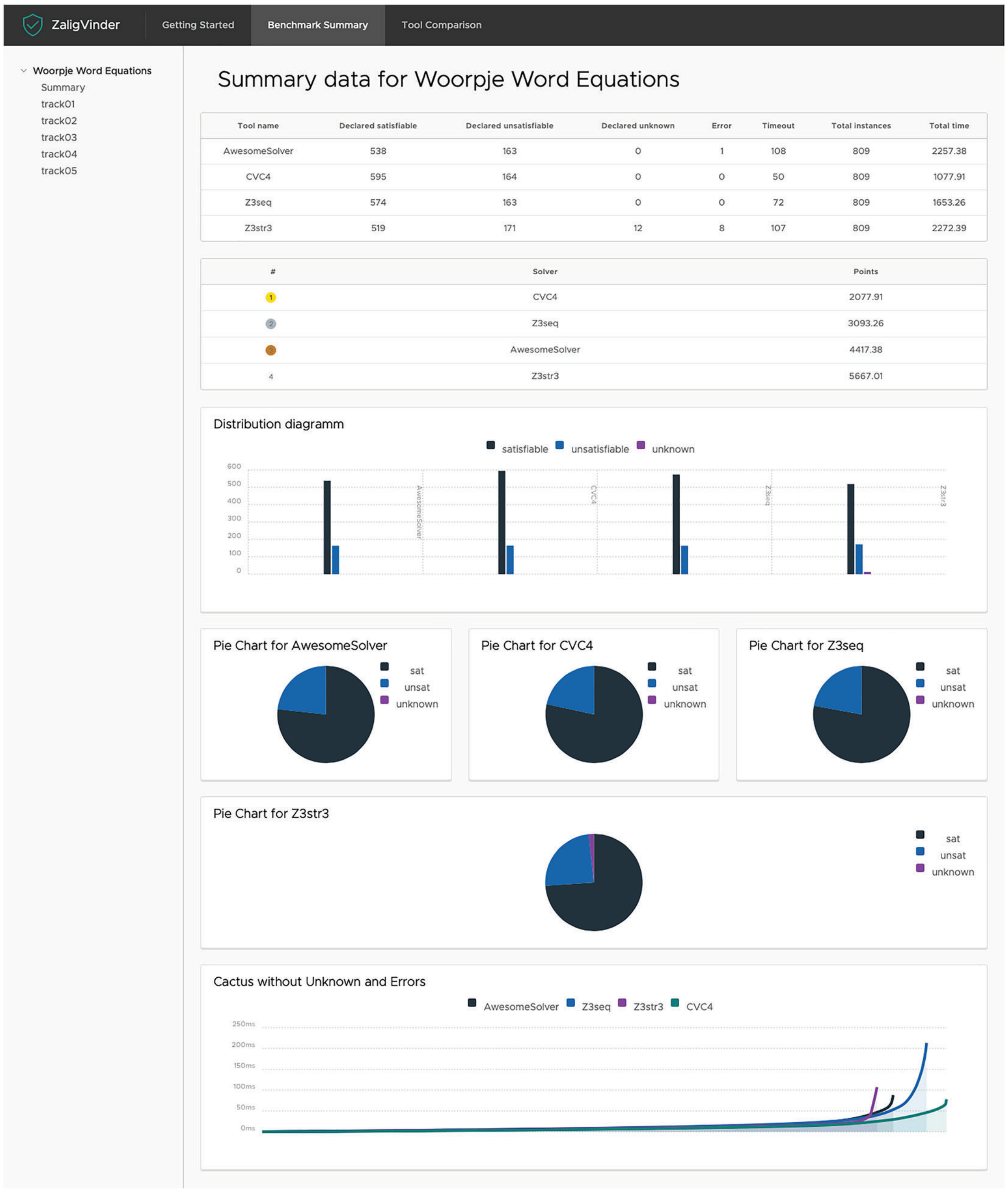
Endpoint	Additional get parameters	Functionalities
`solvers`	—	List of solver names
`tracks`	—	List of tracks including `name`, `id`, corresponding `benchmark` set, and all `instances`
`tracks/ids`	—	List of all tracks `ids`
`tracks/groups`	—	List of all benchmark sets `ids`
`tracks/info`	—	List of all benchmark sets their track `ids` and `names`
`instances`	—	List of all instances including their `name`, id, and `expected` result
`instances/ids/track/<TRACKID>`	—	List of all instances ids of track TRACKID
`instances/ids/group/<BENCHMARKID>`	—	List of all instances ids of benchmark group BENCHMARKID
`instances/solvers/<INSTANCEID>`	`solvers`	Directory mapping the INSTANCEID to a solvers result directory holding `timeouted`, `result` and `time`
`instances/<INSTANCEID>/model.smt`	—	Shows the input formula of instance INSTANCEID
`results`	—	List of all results including `solver` name, `instanceid`, and a `Result` directory holding `timeouted`, `result` and `time`
`results/<TRACKID>`	—	Restricts the results to track TRACKID
`results/reference/<INSTANCEID>`	—	Directory holding expected `result` and `satisfying solvers` and `nsatisfying solvers` for instance INSTANCEID.
`results/<SOLVERNAME>/<INSTANCEID>/output`	—	Shows SOLVERNAMEs output on instance INSTANCEID
`results/<SOLVERNAME>/<INSTANCEID>/model`	—	Shows SOLVERNAMEs model on instance INSTANCEID
`summary/<SOLVERNAME>`	—	Show the accumulated results for SOLVERNAME holding `timeouted`, `satisfied`, `not satisfied`, `error`, `Unknown`, `time`, and total `instances`
`summary/SOLVERNAME/TRACKID`	—	Show the accumulated results for SOLVERNAME on track TRACKID holding the summary data
`chart/cactus`	`format` (e.g png), `solver` names	Shows a cactus plot in the given `format` (default: plain jSON) for `solvers` (default: all solvers)
`chart/distribution/<BENCHMARKID>`	—	Shows a distribution diagram for <BENCHMARKID>
`chart/keywords`	`format` (e.g png)	Shows a barchart with accumulated keywords in the given `format` (default: plain jSON)
`chart/scattered`	`solvers`, `format` (e.g png)	Prints a scattered plot of two solvers
`ranks/TRACKID`	—	List of `solver` name and Par2 score `points` for TRACKID.

4.2.4 | Commandline tools

Another advantage of storing the data in an SQLite database is that it can be used for generating tables and plots for papers. As an example of this, Figure 5 was automatically generated by ZaligVinder. We provide a Python script—called `tablegen.py` —based on the package `npyscreen` allowing the generation of multiple visuals for external usage. After selecting a ZaligVinder database file and an output file name, the user selects benchmark groups and solvers being part of the external visuals. Currently, we offer three selectable techniques: 1. summary tables of the results in LATE X, 2. cactus plots of the results in LATE X using TikZ, and 3. a summary page including cactus plots in Asciidoctor[42] format.

5 | USE CASES

In this section, we present two empirical studies on how to use ZaligVinder to debug a string solver. We highlight the abilities of the GUI to determine performance issues and soundness errors within the string solver of choice. Secondly, we demonstrated the extensibility of ZaligVinder s infrastructure and implemented an analysis technique to discover more insights of our string solver and use this data for further external analysis.

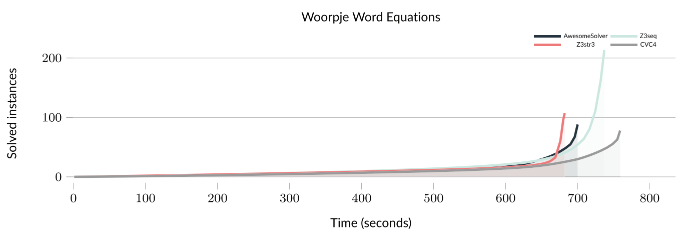


Summary data for Woorpje Word Equations

Tool name	Declared satisfiable	Declared unsatisfiable	Declared unknown	Error	Timeout	Total instances	Total time
AwesomeSolver	538	163	0	1	108	809	2257.38
CVC4	595	164	0	0	50	809	1077.91
Z3seq	574	163	0	0	72	809	1653.26
Z3str3	519	171	12	8	107	809	2272.39

#	Solver	Points
1	CVC4	2077.91
2	Z3seq	3093.26
3	AwesomeSolver	4417.38
4	Z3str3	5667.01

FIGURE 4 Web-GUI: benchmark summary view



	AwesomeSolver	Z3seq	Z3str3	CVC4
sat	538	574	519	595
unsat	163	163	171	164
unknown	0	0	12	0
timeout	108	72	107	50
soundness error	1	0	8	0
program crashes	0	0	0	0
Total correct	700	737	682	759
Time (s)	2.26	1.65	2.27	1.08
Time w/o timeouts (s)	0.1	0.21	0.13	0.08

FIGURE 5 Automatic table and cactus plot generation of ZaligVinder

5.1 | Using the GUI

We are the developers of the fictional solver AwesomeSolver and just finished implementing a new feature. To measure its performance, we set up a run comparing against some of the state-of-the-art string solvers, namely, CVC4, Z3str3, and Z3seq, on the Woorpje benchmark set.

Following Section 4 easily allows us setting up a run because all competing solvers and the benchmark set are already preconfigured. Therefore, we will create a new solver file following the template presented in Listing 3 and add all competing solvers as shown in line 8 to our new runner script. Once the run finishes, ZaligVinder automatically starts a web server showing the website being depicted in Figure 4. The presented diagrams indicate an insufficient performance of our string solver. Moreover, the summary table reveals an error within our solver.

We change our view to the *Tool Comparison* website and select appropriate filters as presented in Section 4.2.3. ZaligVinder presents the summary being depicted in Figure 6: Our solver produces an invalid model on one instance. The produced model is revealed as soon as we click on the brick wall icon. Our solver tries substituting all variables by the empty word. Quickly comparing against the successfully validated model of CVC4 by clicking on the model icon highlights our wrongly presented solution further. The instance indicating the wrong model production is now used for debugging AwesomeSolver.

5.2 | Extending the API for an individual post analysis

Adding a completely new individual analysis to ZaligVinder requires three modifications: 1. setting up appropriate database queries, 2. creating a controller for handling the results being return by the database queries, and 3. adding an endpoint to the webserver.

We are interested in getting all file names where a particular solver produced a wrong model. To add this view to our API, firstly, we come up with the database query to our model. To achieve this, we add the function implemented in Listing 4 to our SQLite interface located in `storage.sqlite`.

```
def getUnverifiedSATInstances(self,solver):
    query = '''SELECT Result.instanceid, TrackInstance.filepath FROM
        ↪ Result, TrackInstance WHERE Result.solver = ? AND Result.
        ↪ result = true AND Result.verified = false AND Result.
        ↪ instanceid = TrackInstance.id'''
    rows = self._db.executeRet (query,(solver,))
    return {t[0] : {"filepath" : t[1]} for t in rows}
```

Listing 4: Adding a new database query to the model `storage.sqlite`

Listing 4. HTML code

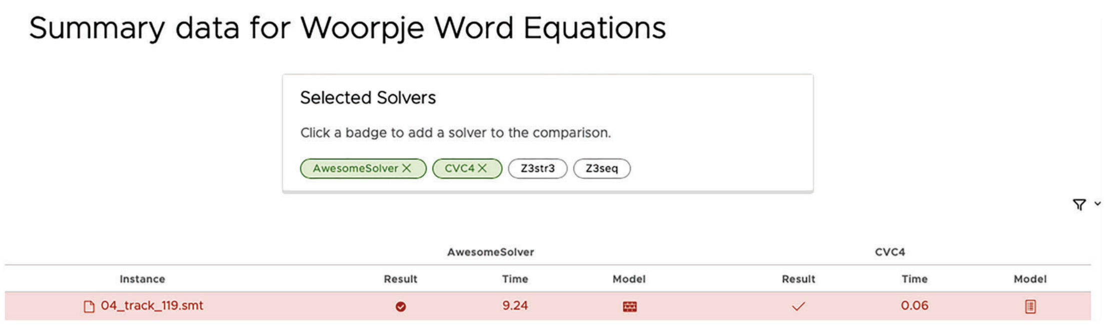
We simply ask for all instances where a solver returned `SAT`, but the verification procedure was not able to validate the model. Secondly, we specify our internal controller logic, which is done by adding the function given in Listing 5 to the results controller located in `webserver.controllers.results`.

```
    def getUnverifiedSATInstancesForSolver (self,params):
        return webserver.views.jsonview.JSONView (self._results.
            ↪ getUnverifiedSATInstances (params["solver"]))
```

Listing 5: Adding a new logic implementation to the results controller `controller.results`

Listing 5. HTML code

to the file `startwebserver.py`. It listens to the URL https://localhost:8081/my_analysis/unverified/SOLVERNAME, where `SOLVERNAME` is an arbitrary string. Whenever this endpoint is called, the previously implemented function of our result controller is called, and the solver name passed on. Guiding your browser to the URL https://localhost:8081/my_analysis/unverified/AwesomeSolver returns in the running example of this section the following string `{“558”: {“filepath”: ~/home/mku/wordy/models/woorpje/track04/04_track_119.smt”}}` —our previously described instance where AwesomeSolver returns a wrong model.


Summary data for Woorpje Word Equations

Selected Solvers

Click a badge to add a solver to the comparison.

AwesomeSolver ✕ CVC4 ✕ Z3str3 Z3seq

	AwesomeSolver			CVC4		
Instance	Result	Time	Model	Result	Time	Model
04_track_119.smt	●	9.24	▦	✓	0.06	▤

FIGURE 6 Detailed analysis of ran instances. The line highlighted in red indicates an error in a solver

```
app.addEndpoint (webserver.routing.RegexMatch("my_analysis/unverified
    ↪ /(?P<solver>[^/]+)"),self._rcontroller.
    ↪ getUnverifiedSATInstancesForSolver)
```

Listing 6: Adding a new endpoint to `startwebserver.py`

Listing 6. HTML code

6 | ANALYSIS OF THE PRESENTED FRAMEWORK AND THE BENCHMARK SETS

Up to this point, we discussed many details on how to apply our framework to string solvers, what a user's output looks like, and how the generated data can be used for a custom post analysis. We also introduced 19 benchmark sets and briefly introduced their origin, as well as the used operations. In this section, we are looking at our framework and also the benchmarks from a different perspective and rate the quality and quantity of both.

6.1 | Our framework

We built the tool ZaligVinder to solve an issue when comparing our own implementations of string solving algorithms with respect to the performance of already published tools. While searching the literature, we were not able to find a tool that is able to reliably compare string solvers with their special needs as for example validating models. To this extent, we designed ZaligVinder tailored towards string solvers with the goal of not only to compare string solvers but also to ease the whole development procedure. As discussed in this paper, our tool features many different techniques to identify performance issues and soundness errors in a user friendly way. We designed our framework not only to ease extensibility but also to easily perform a sufficient post analysis including several mechanism to export the data into a LATE X article.

Given these facts, we evaluate our framework based on the following research questions: 1. Are we able to reliability compare string solvers? 2. Are we able to identify bugs within string solvers? 3. Are we able to gather insights about the string solvers performance? 4. Are the post-processing mechanisms sufficient?

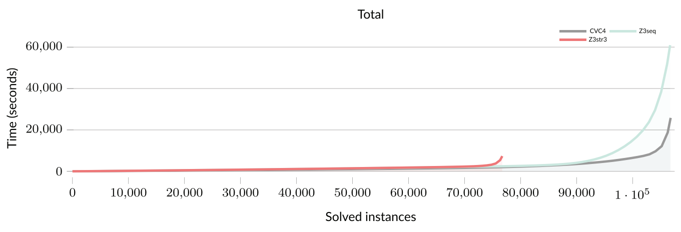
6.1.1 | RQ 1: Are we able to reliability compare string solvers?

With the goal of comparing string solvers in an easy way, we simply follow the steps explained in Section 4.1. Once the run is finished, we are able to review the results within the Web-GUI, simply generate LATE X tables and plots or generate an AsciiDoctor website. Whatever data visualization we choose, we are able to see which solver has the best performance, identify potential performance issues, or review insights on certain misclassified instances.

In Figure 7, we visualize a run of the major SMT-solver binaries CVC4 1.8, Z3Seq 4.8.10, and Z3Str3 4.8.10 on all of our collected benchmarks. We use a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory. For each instance, we use a timeout of 20 s. Clearly, we are able to observe all required details of the run. Overall, we think our goal of building a tool to compare string solvers was achieved in this sense.

6.1.2 | RQ 2: Are we able to identify bugs within string solvers?

One of the key requirements when building a solver is its reliability. To this extent, a solver is supposed to be sound and stable. Because we are facing an undecidable theory, none of the solvers has a warranty for a terminating algorithm, which we dealt with by



	CVC4	Z3seq	Z3str3
sat	76466	76627	52464
unsat	30318	30070	24271
unknown	64	61	33506
timeout	7627	7717	4234
soundness error	0	7	9
program crashes	0	9	9
Total correct	106784	106690	76726
Time (s)	178364.27	215301.37	93145.69
Time w/o timeouts (s)	25824.18	60961.37	8465.69

FIGURE 7 Comparison of CVC4, Z3Seq, and Z3Str3 on all benchmarks

enforcing hardware limits (e.g., a timeout). While developing our tool WOORPJE, our techniques prove to be extremely valuable. As an example, when we extended WOORPJE by a new simplification technique, ZALIGVINDER reported invalid models on a few amount at instances. Looking into the details presented within our Web-GUI revealed the instances where our solver had a bug. Within this, subtle error variables were not fully substituted within the model. Having the abilities to review all affected instance made fixing this bug fairly easy.

Another interesting bug was revealed by looking at our comping solvers. Even though the solver developers agreed on the SMT-LIB 2.6 standard, many issues especially related to Unicode support are not sorted out. As an example, the solver CVC4 is using the escape sequence `\u{5c}` within the found model instead of using a backslash (\). The string theory in Z3 uses the actual backslash symbol. Having said that, the solvers were not able to reason about the counterpart and therefore, reported an invalid model whenever the above situation occurs. To us, this is an unintended behavior and needs to be resolved in future versions.

We not only discovered soundness issues but also many solver crashes during our tests. Looking again at Z3STR3 within our tests, we detected several segmentation faults on the considered benchmarks.

The run visualized in Figure 7 also reveals some solver errors. Z3SEQ classifies seven instances of the WOORPJE benchmark set as unsatisfiable even though they are satisfiable. Z3STR3 classifies eight instances also on the WOORPJE benchmark set wrongly as unsatisfiable. Furthermore, it produces an invalid model on the KAUSLER benchmark suite.

6.1.3 | RQ 3: Are we able to gather insights about the string solvers performance?

The design of the arm selection method implemented within Z3STR4[43] heavily relays on empirical observations made by using ZALIGVINDER. As a short recap, Z3STR4 probes analyze syntactical structures of the input instances and upon that select a sequence of different solvers. To build the probes and sequences of solvers, the developers used the insights gathered about the benchmarks in Section 3 and analyzed the performance of each individual solver with respect to the higher order functions and relations, as well as occurrences of word equations, regular membership, and length constraints. Without having a framework like ZALIGVINDER, this work would have been very tedious.

6.1.4 | RQ 4: Are the postprocessing mechanisms sufficient?

Our postprocessing techniques introduced in Section 4.2 were developed incrementally while analyzing string solvers. We started only having a rudimentary comparison view displaying the basic information of a run and extended it to visualizing models, soundness issues, as well as the score of a solver according to the *par2* measurement. One of the latest addition is an extra table listing only the details of solver crashes and soundness issues to ease debugging specific cases. The output of results was also refined over time. Within a cactus plot, the user is able to choose between multiple different display modes. The LATE X summary tables themselves were extended to contain many required inputs to ease comparing string solvers also purely based on the article. In total, we are certain that the current setup features all required techniques to ease evaluation. Secondly, the incrementally process of achieving the perfect analysis shows the extensibility of our framework.

Looking at our framework from the aforementioned perspectives highlights the achievement of our requirements. Nevertheless, we believe that there are still many features we can add to make the usability even better.

6.2 | Benchmark sets

The SMT-LIB standard in version 2.6 was released in early 2020,[21] and next to introducing new high-level string operations such as `str.to_code` and `str.from_code` and renaming of operations, it introduces Unicode character support. However, many state-of-the-art string solvers still only support the former version 2.5. Adding axioms for new functions or support for a different naming schema is not the biggest issue the solvers have to this end, but moving to Unicode support involves deep modifications of the SMT-solver core. Making steps into this direction requires manpower which, after optimizing, debugging, and analyzing, many different solvers do not seem to be of highest priority: Nearly all state-of-the-art string solvers contain a multitude of soundness, parser, and other sorts of errors. Analyzing the benchmark sets introduced in Section 3 immediately let to the following research questions: 1. Are the gathered instances usable for all solvers supporting SMT-LIB? 2. What can we say about the general quality of the benchmark sets with respect to uniqueness, covered cases, and real-life applications? 3. Can we reliably use these instances for testing string solvers?

Within this section, we will describe the analysis of our benchmarks.

6.2.1 | RQ1: Are the gathered instances usable for all solvers supporting SMT-LIB?

Even though all collected instances are supposed to follow at least the unofficial SMT-LIB 2.0 standard, many instances are not parsable by all solvers, nor follow the conventions. Since the theory of strings of SMT-LIB is introduced, we are facing the quantifier-free fragment of string constraints, meaning there should not be a single quantifier. The instances reveal a different view: Some of them contain quantifiers. An exceptional example are the benchmarks of the NORN benchmark suite. Many of them quantify integer variables. Therefore, parsers as the one being implemented by the developers of CVC4 simply reject such an instance. Another example of insufficient designed instances are the Z3STR2 regression tests. Within these instances, we observe several occurrences of wrongly parametrized `indexof` functions (e.g., `(str.indexof X "ab")`). As we discussed in Section 2, the function `indexof` takes three parameters, two string terms and an integer. Within these instances, the developers missed the integer parameter, such that most solvers cannot handle the query. There are other examples of not well defined higher level functions within the set of benchmarks such that a proper usage requires an identification of insufficient definitions.

6.2.2 | RQ2: What can we say about the general quality of the benchmark sets with respect to uniqueness, covered cases, and real-life applications?

Most of the gathered sets stem from solver developers trying to showcase the performance of their solver in a particular setting. There are only a few exceptions as we also pointed out in Section 3. In Table 1, also in Section 3, we review the most used string operations within these benchmarks. Notably, an operation directly enforcing the correct handling of word equations, namely, SUBSTR, is not the most prominent operator in these string benchmarks. In general, if a solver developer considers all sets of benchmarks for testing the abilities of a string solver, we think that our set forms a sophisticated basis. All commonly used operators are present, and the instances are diverse in their own rights, again, when considering all sets. Nevertheless, when analyzing all sets with respect to redundancy, we cannot count on these instances. We performed an obvious comparison over all sets by simply computing the MD5 hash value of an instance. These results are visualized in Table 3. This simple method shows that 11 out of 19 sets contain duplicates. Moreover, out of 11,4475 instances, more than 40% are duplicates. Overall, this is a huge drawback on the gathered instances and also questions the empirical evaluations reported within the publication of a certain benchmark set.

TABLE 3 Hash duplicates within the collected benchmarks

Benchmark name	#Instances	#Duplicates
Leetcode	2666	14 (0.53%)
PyEx	25,421	2864 (11.27%)
AppScan	8	0 (0.0%)
AutomatArk	19,979	6626 (33.16%)
BanditFuzz	357	0 (0.0%)
Cashew	394	2 (0.51%)
JOACO	94	26 (27.66%)
Kaluza	47,284	32712 (69.18%)
Kausler	120	7 (5.83%)
Trau light	100	0 (0.0%)
Norn	1027	0 (0.0%)
Pisa	12	0 (0.0%)
Sloth	40	0 (0.0%)
Stranger	4	0 (0.0%)
StringFuzz	1065	220 (20.66%)
StringFuzz Regex Generated	4170	59 (1.41%)
StringFuzz Regex Transformed	10,682	4379 (40.99%)
Woorpje	809	0 (0.0%)
Z3str2	243	2 (0.82%)
Total	114,475	46,911 (40.98%)

6.2.3 | RQ3: Can we reliably use these instances for testing string solvers?

We analyzed the set of gathered benchmarks with respect to the standardized notion `(set-info:status x)` where *x* is either `sat` or `unsat`. We discovered that 47,534 instances were annotated with a *possible* result. A deeper look revealed that 13,725 instances were, in fact, wrongly tagged—meaning again only 33,809 instances were correctly annotated. To identify a wrongly classified instance, we used the three leading SMT solvers, namely, CVC4, Z3STR3, and Z3SEQ, and ran them on these instances. If one of the string solvers was able to produce a model, we validated it by using the other competitors. Afterwards, we compared the verified results with the ones being present within the instance. In total, this high count of misclassified instances made the prior knowledge unusable and proved the benchmarks as not being a reliable source for testing string solvers.

In the future, we plan to develop solutions to the detected issues such that solver developers can safely use our modified set of the present benchmarks for solvers supporting SMT-LIB 2.5.

7 | RELATED WORK

Due to the amount of different string solvers, it is surprising that we have not found a single tool incorporating the needs of comparing, debugging, and analyzing string solvers. The literature around string solvers mostly reports empirical data based on custom scripts that are not publicly available. A single tool that is also used for evaluating solvers at SMT-COMP is BENCHEXEC.[10] A drawback of this tool is that it only features head-to-head comparisons of different solvers without being tailored towards string solving. As for SMT-COMP, there is for example no validation of models. Secondly, because it is a pure benchmarking framework, it does not offer the postprocessing abilities our tool has to offer. Another closely related tool is called BenchKit[44] that is used for the Model Checking Contest.[45] The main drawback of this tool is again that it is primarily build to compare running times of solvers without allowing any further analysis. Secondly, it does not seem to be made to cope with SMT solvers.

With respect to the benchmark collection, the SMT-LIB initiative shares some sets of string solving benchmarks featured of the logics `QF_S` and `QF_SLIA` within their GIT repositories.[46] Meanwhile, some of the explained benchmarks in this paper made it to their repository but not all of them. We plan to submit all collected sets after getting the permission of the respected authors to their repositories, too.

8 | CONCLUSION

In this paper, we have gathered a large collection of string solving benchmarks from the literature. These have been incorporated into our own, novel, benchmarking framework ZALIGVINDER. Adding new benchmarks and tools to the framework is made easy by using PYTHON.

Since 2020 when the first version of ZALIGVINDER was published in Kulczynski et al,[11] our tool was used for the analysis of different string solvers with the goals of finding bugs, benchmarking, and also generating valuable tables and figures for papers (cf. Day et al[9] and Berzish et al[15]). It has proven to be reliable and easily extensible—not only due to being completely build in PYTHON but also the easy ways to adapt the code structure we had chosen. Producing the output data of a virtual best solver, extracting all file paths of timed out instances, or printing the error messages a solver produced within a run had been some of the third-party extensions to ZALIGVINDER. Overall, we still believe that ZALIGVINDER provides the fundamentals for the development of an extended platform, to be used in comparing string solvers, in a more objective and complete manner.

In the future, we plan to change the GUI infrastructure to provide a publicly available service. This gives the abilities to not only share the produced ASCII doctor pages but also the complete visualized data with others. Secondly, an addition to perform a distributed run with ZALIGVINDER is a valuable extension to its core. This allows using not just one server but many to gather insights quicker than ever before.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in our GIT repository at https://github.com/zaligvinder/zaligvinder.

ORCID

Mitja Kulczynski https://orcid.org/0000-0003-4650-1110

REFERENCES

1. Kiezun A, Ganesh V, Guo PJ, Hooimeijer P, Ernst MD. Hampi: a solver for string constraints. In: Proceedings of the eighteenth international symposium on software testing and analysis ACM; 2009:105-116.
2. Abdulla PA, Atig MF, Chen Y, Holík L, Rezine A, Rümmer P, Stenman J. Norn: an SMT solver for string constraints. In: Proc. cav, part 1, LNCS, vol. 9206 Springer; 2015:462-469.
3. Yu F, Alkhalaf M, Bultan T. Stranger: an automata-based string analysis tool for php. In: Proc. tacas, TACAS'10. Springer Springer; 2010:154-157.
4. Aydin A, Bang L, Bultan T. Automata-based model counting for string constraints. In: Proc. cav Kroening D, Păsăreanu CS, eds. Springer International Publishing Springer; 2015:255-272.
5. Barrett CW, Conway CL, Deters M, et al. CVC4. In: Proc. cav, LNCS, vol. 6806 Springer; 2011:171-177.

6. Zheng Y, Ganesh V, Subramanian S, Tripp O, Berzish M, Dolby J, Zhang X. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods Syst Design*. 2017;50(2-3):249-288.
7. Berzish M, Ganesh V, Zheng Y. Z3str3: A string solver with theory-aware heuristics. In: Proc. fmcad; 2017:55-59.
8. Day JD, Ehlers T, Kulczynski M, Manea F, Nowotka D, Poulsen DB. On solving word equations using SAT. In: Proc. rp, LNCS, vol. 11674 Springer; 2019:93-106.
9. Day JD, Kulczynski M, Manea F, Nowotka D, Poulsen DB. Rule-based word equation solving. In: Proc. formalise IEEE; 2020.
10. Wendler P, Beyer D. *sosy-lab/benchexec: Release 3.8*: Zenodo; 2021. online, https://doi.org/10.5281/zenodo.4773326
11. Kulczynski M, Manea F, Nowotka D, Poulsen DB. The power of string solving: simplicity of comparison. In: Proceedings of the ieee/acm 1st international conference on automation of software test IEEE; 2020:85-88.
12. Liang T, Reynolds A, Tsiskaridze N, Tinelli C, Barrett C, Deters M. An efficient SMT solver for string constraints. *Formal Methods Syst Design*. 2016; 48(3):206-234.
13. Makanin GS. The problem of solvability of equations in a free semigroup. *Sbornik: Math*. 1977;32(2):129-198.
14. Day JD, V.Ganesh, He P, Manea F, Nowotka D. The satisfiability of word equations: decidable and undecidable theories. In: Proc. rp, LNCS, vol. 11123 Springer; 2018:15-29.
15. Berzish M, Kulczynski M, Mora F, Manea F, Day J, Nowotka D, Ganesh V. An SMT solver for regular expressions and linear arithmetic over string length. Proc CAV Springer; 2021.
16. Amadini R. A survey on string constraint solving; 2020.
17. Davis M, Logemann G, Loveland D. A machine program for theorem-proving. *Commun ACM*. 1962;5(7):394-397.
18. Nieuwenhuis R, Oliveras A, Tinelli C. Solving sat and sat modulo theories: from an abstract Davis–Putnam–Logemann–Loveland procedure to dpll (t). *J ACM (JACM)*. 2006;53(6):937-977.
19. De Moura L, Bjørner N. Satisfiability modulo theories: introduction and applications. *Commun ACM*. 2011;54(9):69-77.
20. Barrett C, Stump A, Tinelli C, et al. The smt-lib standard: version 2.0. In: Proceedings of the 8th international workshop on satisfiability modulo theories (edinburgh, england), Vol. 13; 2010:14.
21. Tinelli C, Barrett C, Fontaine P. Satisfiability modulo theories—theory of strings. http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml, Accessed: 2021-01-06.
22. Chen T, Zhang X, Chen R, Yang B, Bai Y. Conpy: concolic execution engine for python applications. In: International conference on algorithms and architectures for parallel processing Springer; 2014:150-163.
23. Irlbeck M. Deconstructing dynamic symbolic execution. *Depend Softw Syst Eng*. 2015;40:26.
24. Reynolds A, Woo M, Barrett C, Brumley D, Liang T, Tinelli C. Scaling up dpll (t) string solvers using context-dependent simplification. In: Proc. cav Springer; 2017:453-474.
25. Ball T, Daniel J. Deconstructing dynamic symbolic execution. In: Proc. Marktoberdorf Summer School on Dependable Software Systems Engineering IOS Press; 2014.
26. Zheng Y, Zhang X, Ganesh V. Z3-str: a z3-based string solver for web application analysis. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering ACM; 2013:114-124.
27. Ibm appscan source web site. https://www.ibm.com/dk-da/security/application-security/appscan
28. Scott J, Mora F, Ganesh V. Banditfuzz: a reinforcement-learning based performance fuzzer for smt solvers. 2020.
29. Brennan T, Tsiskaridze N, Rosner N, Aydin A, Bultan T. Constraint normalization and parameterized caching for quantitative program analysis. In: Proc. 11th joint meeting on foundations of software engineering ACM; 2017:535-546.
30. Luu L, Shinde S, Saxena P, Demsky B. A model counter for constraints over unbounded strings. In: Acm sigplan notices, Vol. 49 ACM; 2014:565-576.
31. Thomé J, Shar LK, Bianculli D, Briand L. An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE. IEEE TSE; 2018.
32. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D. A symbolic execution framework for JavaScript. In: Proc. 31st s&p; 2010:513-528.
33. Saxena P, Akhawe D, Hanna S, Mao F, McCamant S, Song D. Kaluza web site. webblaze.cs.berkeley.edu/2010/kaluza/, Accessed: 2020-01-01.
34. Kausler S, Sherman E. Evaluation of string constraint solvers in the context of symbolic execution. In: Proc. ase ieee/acm ACM; 2014:259-270.
35. Abdulla PA, Atig MF, Chen Y, Diep BP, Holík L, Rezine A, Rümmer P. Trau: Smt solver for string constraints. In: Proc. fmcad IEEE; 2018:1-5.
36. Abdulla PA, Atig MF, Chen Y, Holík L, Rezine A, Rümmer P, Stenman J. String constraints for verification. In: Proc. cav Springer; 2014:150-166.
37. Tateishi T, Pistoia M, Tripp O. Path-and index-sensitive string analysis based on monadic second-order logic. *TOSEM*. 2013;22(4):33.
38. Holík L, Janku P, Lin AW, Rümmer P, Vojnar T. String constraints with concatenation and transducers solved efficiently. *PACMPL*. 2018;2(POPL):4:1-4:32.
39. Lin AW, Barceló P. String solving with word equations and transducers: towards a logic for analysing mutation xss. In: Acm sigplan notices, Vol. 51 ACM; 2016:123-136.
40. Blotsky D, Mora F, Berzish M, Zheng Y, Kabir I, Ganesh V. Stringfuzz: a fuzzer for string solvers. In: Proc. cav Springer; 2018:45-51.
41. Zheng Y, Ganesh V, Subramanian S, Tripp O, Berzish M, Dolby J, Zhang X. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods Syst Design*. 2017;50(2-3):249-288.
42. Allen D, White S. Asciidoctor. https://asciidoctor.org, Accessed: 2021-01-06.
43. Berzish M. Z3str4: A solver for theories over strings; 2021.
44. Kordon F, Hulin-Hubard F. Benchkit, a tool for massive concurrent benchmarking. In: 2014 14th international conference on application of concurrency to system design; 2014:159-165.
45. Toullalan A, Kordon F. Mcc 2021. Communications of the ACM; 2021.
46. SMT-LIB benchmarks repository. https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks, Accessed: 2021-06-13.

How to cite this article: Kulczynski M, Manea F, Nowotka D, Poulsen DB. ZaligVinder: A generic test framework for string solvers. *J Softw Evol Proc*. 2021;e2400. doi:10.1002/smr.2400