An SMT Solver for Regular Expressions and Linear Arithmetic over String Length

Murphy Berzish[1](✉), Mitja Kulczynski[2], Federico Mora[3], Florin Manea[4], Joel D. Day[5], Dirk Nowotka[2], and Vijay Ganesh[1]

[1] University of Waterloo, Waterloo, Canada
`mtrberzi@uwaterloo.ca`
[2] Kiel University, Kiel, Germany
[3] University of California, Berkeley, USA
[4] University of Göttingen and Campus-Institute Data Science, Göttingen, Germany
[5] Loughborough University, Loughborough, UK

Abstract. We present a novel length-aware solving algorithm for the quantifier-free first-order theory over regex membership predicate and linear arithmetic over string length. We implement and evaluate this algorithm and related heuristics in the Z3 theorem prover. A crucial insight that underpins our algorithm is that real-world regex and string formulas contain a wealth of information about upper and lower bounds on lengths of strings, and such information can be used very effectively to simplify operations on automata representing regular expressions. Additionally, we present a number of novel general heuristics, such as the prefix/suffix method, that can be used to make a variety of regex solving algorithms more efficient in practice. We showcase the power of our algorithm and heuristics via an extensive empirical evaluation over a large and diverse benchmark of 57256 regex-heavy instances, almost 75% of which are derived from industrial applications or contributed by other solver developers. Our solver outperforms five other state-of-the-art string solvers, namely, CVC4, OSTRICH, Z3seq, Z3str3, and Z3-Trau, over this benchmark, in particular achieving a speedup of 2.4× over CVC4, 4.4× over Z3seq, 6.4× over Z3-Trau, 9.1× over Z3str3, and 13× over OSTRICH.

Keywords: String solvers · SMT solvers · Regular expressions

1 Introduction

Satisfiability Modulo Theories (SMT) solvers that support theories over regular expression (regex) membership predicate and linear arithmetic over length of strings, such as CVC4 [25], Z3str3 [8], Norn [3], S3P [39], and HAMPI [22], have enabled many important applications in the context of analysis of string-intensive programs. Examples include symbolic execution and path analysis [11,32], as well as security analyzers that make use of string and regex constraints for input sanitization and validation [5,33,35]. Regular expression libraries in programming

© The Author(s) 2021

languages provide very intuitive and popular ways for developers to express input validation, sanitization, or pattern matching constraints. Common to all these program analysis applications is the requirement for a rich quantifier-free (QF) first-order theory over strings, regexes, and integer arithmetic over string length. Unfortunately, the QF first-order theory of strings containing regex constraints, linear integer arithmetic over string length, string-number conversion, and string concatenation (but no string equations[1]) is undecidable [7,9]. In a previous paper [19] we showed that a related QF first-order theory over word equations, linear integer arithmetic over string length, and string-number conversion predicate, but without regular expressions is also undecidable. It can also be shown that many non-trivial fragments of this theory are hard to decide (e.g., they have exponential-space lower bounds or are PSPACE-complete). Therefore, the task of creating efficient solvers to handle practical string constraints that belong to fragments of this theory remains a very difficult challenge.

Many modern solvers typically handle regex constraints via an automata-based approach [4]. Automata-based methods are powerful and intuitive, but solvers must handle two key practical challenges in this setting. The first challenge is that many automata operations, such as intersection, are computationally expensive, yet handling these operations is required in order to solve constraints that are relevant to real-world applications. The second challenge relates to the integration of length information with regex constraints. Length constraints derived from automata may imply a disjunction of linear constraints, which is often more challenging for solvers to handle than a conjunction.

As we demonstrate in this paper, the challenges of using automata-based methods can be addressed via prudent use of *lazy extraction of implied length constraints* and *lazy regex heuristics* in order to avoid performing expensive automata operations when possible. Inspired by this observation, we introduce a length-aware automata-based algorithm, Z3str3RE (and its implementation as part of the Z3 theorem prover [18]), for solving regex constraints and linear integer arithmetic over length of string terms. Z3str3RE takes advantage of the compactness of automata in representing regular expressions, while at the same time mitigating the effects of expensive automata operations such as intersection by leveraging length information and lazy heuristics.

Contributions: We make the following contributions in this paper.

Z3str3RE: An SMT Solver for Regular Expressions and Linear Integer Arithmetic over String Length. In Sect. 3, we present a novel decision procedure for the QF first-order theory over regex membership predicate and linear integer arithmetic over string length. We also describe its implementation, Z3str3RE, as part of the Z3 theorem prover [8,18]. The basic idea of our algorithm is that formulas obtained from practical applications have many implicit and explicit length constraints that can be used to reason efficiently about automata representing regexes. In Sect. 4 we present four heuristics that aid in solving regular expression constraints and that can be leveraged in general settings. Specifically, we present a heuristic to derive explicit length information directly from

[1] We use the terms "word" and "string" interchangeably in this paper.

regexes, a heuristic to perform expensive automata operations lazily, a heuristic to refine lower and upper bounds on lengths of string terms with respect to regex constraints, and a prefix/suffix over-approximation heuristic to find empty intersections without constructing automata. All heuristics are designed to guide the search and avoid expensive automata operations whenever possible. Our solver, Z3str3RE, handles the above theory as well as extensions (e.g. word equations and substring function) via the existing support in Z3str3. We focus on the core algorithm as it is the centerpiece of our regex solver. We also carefully distinguish the novelty of our method from previous work.

Empirical Evaluation and Comparison of Z3str3RE[2] Against CVC4, OSTRICH, Z3seq, Z3str3, and Z3-Trau: To validate the practical efficacy of our algorithm, we present a thorough and extensive evaluation of Z3str3RE in Sect. 5, where we compare it against CVC4 [24], OSTRICH [15], Z3's sequence solver [18], Z3str3 [42], and Z3-Trau [1] on 57256 instances across four regex-heavy benchmarks with connections to industrial security applications, including instances from Amazon Web Services and AutomatArk [16]. Z3str3RE significantly outperforms other state-of-the-art tools on the benchmarks considered, having more correctly solved instances in total, lower running time, and fewer combined timeouts/unknowns than other tools, and no soundness errors or crashes. We note that almost 75% of the benchmarks were obtained from industrial applications or other solver developers. Over all the benchmarks, we demonstrate a speedup of 2.4× over CVC4, 4.4× over Z3seq, 6.4× over Z3-Trau, 9.1× over Z3str3, and 13× over OSTRICH.

2 Preliminaries

This section contains some basic definitions as well as a brief overview of the theoretical results which shape the landscape in which we state our contribution.

2.1 Basic Definitions

We first describe the syntax and semantics of the input language supported by our solver Z3str3RE (Algorithm 1).

Syntax: The core algorithm we present in Sect. 3 accepts formulas of the quantifier-free many-sorted first-order theory of regex membership predicates over strings and linear integer arithmetic over string length function. The syntax of this theory is shown in Fig. 1.

We denote the set of all string variables and all integer variables as $\mathrm{Var}_{\mathrm{str}}$ and $\mathrm{Var}_{\mathrm{int}}$ respectively, and the set of all string constants and all integer constants as $\mathrm{Con}_{\mathrm{str}}$ and $\mathrm{Con}_{\mathrm{int}}$ respectively. String constants are any sequence of zero or more characters over a finite alphabet (e.g., ASCII).

Atomic formulas are regular expression membership constraints and linear integer (in)equalities. Regex terms are denoted recursively over regex concatenation, union, Kleene star, and complement, and for a string constant w, the

[2] A reproduction package is available at https://figshare.com/s/5ae73a6f3c55f5c5e4c1.

$$
\begin{array}{lll}
F & ::= & Atom \quad | \quad F \land F \quad | \quad F \lor F \quad | \quad \neg F \\
Atom & ::= & t_{str} \in RE \quad | \quad A_{int} \\
A_{int} & ::= & t_{int} = t_{int} \quad | \quad t_{int} < t_{int} \\
RE & ::= & \text{“}w\text{”} \mid RE \cdot RE \mid RE \cup RE \mid RE^{*} \mid \overline{RE}, \text{ with } w \in \mathrm{Con}_{\mathrm{str}} \\
t_{int} & ::= & m \mid v \mid len(t_{str}) \mid t_{int} + t_{int} \mid m \cdot t_{int}, \text{with } m \in \mathrm{Con}_{\mathrm{int}}, v \in Var_{int} \\
t_{str} & ::= & s, \text{ with } s \in \mathrm{Var}_{\mathrm{str}} \cup \mathrm{Con}_{\mathrm{str}}
\end{array}
$$

Fig. 1. Syntax of the input language accepted by Algorithm 1. Z3str3RE accepts an extension of this syntax supporting word equations and other string terms.

regex term “w” represents the regular language containing w only. All regex terms must be grounded (i.e. cannot contain variables). Linear integer arithmetic terms include integer constants and variables, addition, and string length. Multiplication by a constant is expanded to repeated addition. String terms are either string variables or string constants. The length of a string S is denoted by $len(S)$, the number of characters in S. The empty string has length 0.

Our implementation Z3str3RE supports the theory in Fig. 1 extended with more expressive functions and predicates, including word equations (equality between arbitrary string terms) and functions such as `indexof` and `substr` that are needed for program analysis. Z3str3RE handles these terms via existing support in Z3str3. We focus on the above input language in the presentation of our algorithm in this paper and theoretical content.

Semantics: We refer the reader to [42] for a detailed description of the semantics of standard terms in this theory. We focus here on the semantics of terms which are less commonly known. The regex membership predicate $S \in R$, where S is a string term and R is a regex term, is defined by structural recursion as follows:

$$
\begin{array}{ll}
S \in \text{“}w\text{”} & \text{iff } S = w \,(\text{where}\, w \,\text{is a string constant}) \\
S \in R_1 \cdot R_2 & \text{iff there exist strings}\, S_1, S_2 \,\text{with}\, S = S_1 \cdot S_2, S_1 \in R_1, S_2 \in R_2 \\
S \in R_1 \cup R_2 & \text{iff either}\, S \in R_1 \,\text{or}\, S \in R_2 \\
S \in R^{*} & \text{iff either}\, S = \epsilon \,\text{or there exists a positive integer}\, n \,\text{such that} \\
 & \quad S = S_1 \cdot S_2 \cdot \ldots \cdot S_n \,\text{and}\, S_i \in R \,\text{for each}\, i = 1 \ldots n \\
S \in \overline{R} & \text{iff } S \notin R \,(\text{that is}, S \in R \,\text{is false})
\end{array}
$$

2.2 Theoretical Landscape

To put our contributions in context, we briefly discuss a series of (un)decidability and complexity results developed around the fragments and extensions of the theory supported by Z3str3RE.

In particular, we consider extensions which may have a string-number conversion predicate *numstr*[3] and/or string concatenation. Both extensions are

[3] We introduce *numstr*, which is not part of the SMT-LIB standard, in order to simplify presentation of the theoretical results. The predicate is no more expressive than the standard operators `str.to_int`/`str.from_int`, except that those terms handle decimal inputs. The results easily extend to other (finite) alphabets including decimal/hexadecimal digits with appropriate case analysis.

important to real-world program analysis. The predicate $numstr$ has the syntax $numstr(t_{int}, t_{str})$ and the following semantics: $numstr(n, s)$ is true for a given integer n and string s iff s is a valid binary representation of the number n (possibly with leading zeros) and n is a non-negative integer. That is, s only contains the characters 0 and 1, and $\sum_{i=0}^{len(s)-1} s'[i]2^{len(s)-i-1} = n$, where $s'[i]$ is 0 if the ith character in s is '0' and 1 if that character is '1'. String concatenation has the syntax $t_{str} ::= t_{str} \cdot t_{str}$ and the usual semantics defined by SMT-LIB [10].

In the following, $T_{LRE,n,c}$ is the quantifier-free many-sorted first-order theory of linear integer arithmetic over string length function (L), regex (RE) membership predicates, string-number conversion (n), and string concatenation (c) [4]. The following quantifier-free fragments of $T_{LRE,n,c}$ are of interest: $T_{LRE,c}$, T_{LRE}, $T_{RE,n,c}$, $T_{RE,n}$, and T_{RE}. The fragment $T_{LRE,c}$ (respectively, T_{LRE}) has all functions and predicates of $T_{LRE,n,c}$ except the string-number conversion predicate (and, respectively, except the string concatenation function). The theory $T_{RE,n,c}$ (respectively, $T_{RE,n}$ and T_{RE}) has all functions and predicates of $T_{LRE,n,c}$ except the length function (and, respectively, the string concatenation function, and, in the case of T_{RE}, the string-number conversion predicate). Note that while all these theories allow equalities between terms of sort Int, they do not allow equalities between terms of sort Str and cannot express general word equations.

The theoretical landscape is laid out as follows. Firstly, following the results and techniques introduced in [3], we obtain that $T_{LRE,c}$ and, in particular, T_{LRE} is decidable. A procedure deciding a formula from $T_{LRE,c}$ would first construct for each variable (string or integer), based on the regular expression constraints and length constraints which involve it, a finite automaton, then reduce the problem of checking the satisfiability of the formula to checking whether the constructed automata accept at least one string. A similar approach shows that $T_{RE,n}$ is decidable. We observe that the presence of complements in regular expressions is an inherent source of complexity for these procedures. Indeed, we can easily encode the universality problem for regular expressions as a formula in the theory T_{RE}. Moreover, given a regex R of length n over an alphabet Σ, deciding whether $L(R) = \Sigma^*$ is equivalent to deciding the satisfiability of the formula φ of T_{RE} consisting of the atoms $x \in \overline{R}$ and $x \in \Sigma^*$. Accordingly, by the results from [37], if the choice for R is restricted to regular expressions with at least k stacked complements, then there exists a positive rational number c such that the considered problems are not contained in NSPACE$\left(\underbrace{2^{2^{2^{\cdots 2}}}}_{k-1\text{times}}{}^{cn}\right)$.
In other words, the depth of the stack of complements of the formula translates to the height of the tower of exponents in the complexity of deciding that formula φ. On the other hand, if we only consider regular expressions without stacked complements, then the decision problems for the considered theories are PSPACE-complete. Indeed, the automata-based approach described above can be implemented to work in nondeterministic polynomial space; strongly related complexity results are obtained in [26,27].

[4] Note that the fragments considered here do not include word equations.

Algorithm 1: Z3str3RE's length-aware algorithm for the theory T_{LRE} of regex and integer constraints

```
Input   : Conjunction $\phi$ of constraints of the form $S \in RE$, and conjunction $\psi$ of linear
          integer arithmetic constraints over string lengths
Output  : SAT or UNSAT
forall constraints $S \in RE$ in $\phi$ do
    $L_S \leftarrow$ ComputeLengthAbstraction($S$) ;
    $L_{RE} \leftarrow$ ComputeLengthAbstraction($RE$) ;
    if $\psi \cup L_S \cup L_{RE}$ inconsistent then
        return UNSAT
    end
    refine $L_S$ as tightly as possible with respect to $L_{RE}$;
end
forall strings $S_i$ occurring in $\phi$ do
    let $\mathcal{R}$ be the set of all regexes $RE$ in all terms $S_i \in RE$ ;
    Automaton $I \leftarrow$ intersection of all automata corresponding to regexes in $\mathcal{R}$ ;
    if $I$ is empty then
        return UNSAT
    else
        $L_I \leftarrow$ ComputeLengthAbstraction($I$) ;
    end
end
$\mathcal{L}_S \leftarrow$ the union of all length abstractions $L_S$;
$\mathcal{L}_{RE} \leftarrow$ the union of all length abstractions $L_{RE}$;
$\mathcal{L}_I \leftarrow$ the union of all length abstractions $L_I$;
if $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$ has any solution $M$ then
    forall strings $S$ occurring in $\phi$ do
        obtain $len(S)$ from $M$ ;
        let $\mathcal{A}$ be the set of all automata for all regexes $RE$ in all terms $S \in RE$ ;
        Automaton $J \leftarrow$ intersection of all terms in $\mathcal{A}$ ;
        $S \leftarrow$ any string of length $len(S)$ in $J$ ;
    end
    return SAT
else
    return UNSAT
end
```

At the opposite end of the spectrum is the theory $T_{LRE,n,c}$, which is undecidable. Indeed, one can show that the more specific theory $T_{RE,n,c}$ (i.e. disallowing arithmetic over length) has equivalent expressive power to the theory of word equations with regular constraints, a predicate allowing the comparison of the length of string terms, and the *numstr* predicate. Therefore, using the techniques from [17], one can show that the theory $T_{LRE,n,c}$, in which we additionally allow arithmetic over length, is undecidable [7].

3 Length-Aware Regular Expression Algorithm

This section outlines the high-level algorithm used by Z3str3RE to solve the satisfiability problem for T_{LRE}, and its extension based on length-aware heuristics.

3.1 High-Level Algorithm

The pseudocode presented in Algorithm 1 captures the essence of Z3str3RE regex solver. Implementation-specific details are omitted for clarity. Z3str3RE incorporates a version of this algorithm as part of a DPLL(T)-style interaction with a core solver for Boolean combinations of atoms and other theory solvers able to handle arithmetic constraints and other terms. The tool handles string concatenation, string equality, and other string terms and predicates besides regex membership and string length via existing support in Z3str3, and leverages Z3's integer arithmetic solver for arithmetic reasoning and model construction. This high-level presentation is expanded in Sect. 4, where we describe several heuristics used in our implementation as part of the Z3str3RE tool.

The algorithm takes as input a conjunction ϕ of regex membership constraints and a conjunction ψ of linear integer arithmetic constraints over the lengths of string variables appearing in ϕ. Without loss of generality, it is assumed that all constraints in ϕ are positive; negative constraints $S \notin RE$ can be replaced with the positive complement $S \in \overline{RE}$. The algorithm returns SAT iff there is a satisfying assignment to all string variables consistent with the regex constraints ϕ and length constraints ψ. It is assumed that the algorithm has access to a decision procedure for checking the consistency of linear integer arithmetic constraints and for obtaining satisfying assignments to these constraints (in our implementation, this is fulfilled by Z3's arithmetic solver).

Lines 1–8 check whether the length information implied by ϕ is consistent with ψ. The function `ComputeLengthAbstraction` takes as input either a string term S or a regex RE and computes a system of length constraints corresponding to derived length information from string constraints or possible lengths of words accepted by the regex RE. This abstraction is exact, not an over-approximation. For example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. If the length abstractions are inconsistent with the given length constraints, there can be no solution which satisfies both the length and regex constraints, and hence the algorithm returns UNSAT. Otherwise, line 7 refines the length abstraction L_S with respect to the regex RE. This improves the efficiency of finding solutions to the augmented system of length constraints later in the algorithm. In our implementation, the lower and upper bounds of the length of S are checked against the lengths of accepting paths in the automaton for RE. For instance, if L_S implies that $len(S) \geq 5$, but the shortest accepting path in the automaton has length 7, the lower bound is refined to $len(S) \geq 7$.

Lines 9–17 check that the intersection of all automata constraining each string variable is non-empty. Although intersecting automata is relatively expensive (as it runs in quadratic time w.r.t. the size of the intersected automata), it is still more efficient to do this before enumerating length assignments, and taking the intersection here is necessary to maintain soundness. (The heuristics in Sect. 4 illustrate some methods by which this computation can be made more efficient or even avoided.) If the length information is consistent, the algorithm adds a length abstraction constraint L_I encoding the lengths of all possible solutions to the intersection I.

By construction of $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$, the input formula is satisfiable iff this system of integer constraints has a solution. If such a solution M exists, lines 22–28 construct an assignment for each string variable with respect to its length assignment. A solution must exist as the lengths of strings considered are limited to those lengths for which the intersection of the corresponding automata is non-empty; the solution is consistent by construction with both the input length constraints and string constraints. If a solution M does not exist, then the constraints $\phi \wedge \psi$ are not jointly satisfiable, and the algorithm returns UNSAT.

We demonstrate soundness, completeness, and termination of Algorithm 1 as follows. On line 4 we check whether $\psi \cup L_S \cup L_{RE}$ is satisfiable. If not, we return UNSAT on line 5. Lines 9–17 check whether the intersection of regex constraints for each string variable is empty. If so, we return UNSAT; otherwise, we add an additional constraint encoding the lengths of all strings in this intersection. Therefore, $\psi \cup \mathcal{L}_S \cup \mathcal{L}_{RE} \cup \mathcal{L}_I$ has a solution iff there exists an assignment to each string variable that is consistent with the arithmetic constraints ψ and that corresponds to the length of a solution in the intersection of its regex constraints $\mathcal{L}_I$. Lines 22–28 construct this solution if it exists. Therefore, Algorithm 1 is a decision procedure for the QF first-order theory of regex constraints, string length, and linear integer arithmetic.

As previously mentioned, Z3str3RE supports other high-level operations that are not part of this theory via existing support in Z3str3. An extension to this algorithm provides support for including these operations, which may render the theory undecidable. These terms are not in Algorithm 1 because their inclusion would make the algorithm incomplete (see Sect. 2.2). Algorithm 1 describes the part of the implementation which is novel and complete.

4 Length-Aware and Prefix/Suffix Heuristics in Z3str3RE

In this section, we describe the length-aware heuristics that are used in Z3str3RE to improve the efficiency of regular expression reasoning. We present an empirical evaluation of the power of these heuristics in Sect. 5.6.

4.1 Computing Length Information from Regexes

The first length-aware heuristic is used when constructing the length abstraction on line 3. If the regex can be easily converted to a system of equations describing the lengths of all possible solutions (for instance, in the case when it does not contain any complements or intersections), this system can be returned as the abstraction without constructing the automaton for RE yet. As previously illustrated, for example, given the regex $(abc)^*$ as input, `ComputeLengthAbstraction` would construct the length abstraction $S \in (abc)^* \rightarrow len(S) = 3n, n \geq 0$ for a fresh integer variable n. Note that this can be done from the syntax of the regex without converting it to an automaton. Deriving length information from the automaton would be simple by, for example, constructing a corresponding unary automaton and converting to Chrobak normal form. However, performing automata construction lazily means we cannot rely on having an automaton in all cases; this technique also provides length information even when constructing an automaton would be expensive.

In cases where we cannot directly infer the length abstraction, the heuristic will fix a lower bound on the length of words in RE, and possibly an upper bound if it exists. Reasoning about the length abstraction early in the procedure gives our algorithm the opportunity to detect inconsistencies before expensive automaton operations are performed. This gives the arithmetic solver more opportunities to propagate facts discovered by refinement and potentially more chances to find inconsistencies or learn further derived facts.

4.2 Optimizing Automata Operations via Length Information

Similarly, computing the intersection I in line 11 is done lazily in the implementation of Z3str3RE and over several iterations of the algorithm. The most expensive intersection operations can be performed at the end of the search, after as much other information as possible has been learned. We use the following heuristics recursively to estimate the "cost" of each operation without actually constructing any automata:

- For a string constant, the estimated cost is the length of the string.
- For a concatenation or a union of two regex terms X and Y, the estimated cost is the sum of the estimates for X and Y.
- For a regex term X^*, the estimated cost is twice the estimate for X.
- For a regex term X under complement, the estimated cost is the product of the estimates obtained from subterms of X.

In essence, the constructions which "blow up" the least are expected to be the least expensive and are performed first. In the best-case scenario, this could mean avoiding the most expensive operations completely if an intersection of smaller automata ends up being empty. In the worst case, all intersections are computed eventually, as this is necessary to maintain the soundness of our approach.

4.3 Leveraging Length Information to Optimize Search

Our implementation communicates integer assignments and lower/upper bounds with the external arithmetic solver in order to prune the search space. Checking for length assignments is done in practice as an abstraction-refinement loop involving Z3's arithmetic solver. The arithmetic solver proposes a single candidate model for the system of arithmetic constraints; the regex algorithm checks whether that model has a corresponding solution over the regex constraints. If it does not, it asserts a conflict clause blocking that combination of length assignments and regex constraints from being considered again. This is necessary in a DPLL(T)-style solver such as Z3 in order to handle Boolean structure in the input formula.

4.4 Prefix/Suffix Over-Approximation Heuristic

As previously mentioned, computing automata intersections is expensive, but in many cases it is necessary in order to prove that a set of intersecting regex constraints has no solution. In some cases, this can be done "by inspection" from the syntax of the regex terms without constructing or intersecting any automata. From the structure of a regular expression, it is easy to determine the first letter of all possible accepted strings that it matches. If several regexes would be intersected over the same string term, this is used to check whether these regexes have a prefix of length one in common. If they do not, their intersection cannot contain any strings other than the empty string (and we can also check whether the empty string could be accepted by a similar syntactic approach). A similar construction for suffixes of length 1 is also used. In this way, the heuristic can infer that the intersection of several regex constraints is either empty, resulting in a conflict clause, or can only contain the empty string, resulting in a new fact and a simplification of the formula – without actually constructing the intersection or, in fact, constructing any automata for these regexes.

For example, consider the following regex constraints on a variable X:

$$X \in (abc)^*$$
$$X \in a^+ \mid b^+$$

In the first constraint, the pattern abc is matched zero or more times, and could be empty; therefore, either X is empty or it must start with a and end with c. In the second constraint, each pattern is matched at least once, and cannot be empty; therefore X must start with a or b, end with a or b, and cannot be the empty string. Observe that according to the prefix heuristic, these constraints are consistent, since a is a valid prefix of both regexes; however, according to the suffix heuristic, they are inconsistent, as the possible suffixes a and b of the second regex do not include c, and the empty string is not a solution to both constraints. Hence these constraints are not jointly satisfiable.

As demonstrated, all of these facts are derived from the syntax of the regular expression without constructing any automata. By constructing an over-approximation of the possible solutions of X allowed by regex constraints, the heuristic can determine that their intersection is empty (or can only contain the empty string) without computing it precisely using expensive automata-based reasoning. We limit this heuristic to the first letter as each additional letter requires exponentially more space.

5 Empirical Results

In this section, we describe the empirical evaluation of Z3str3RE, our implementation of the length-aware regular expression algorithm presented in Sect. 3, to validate the effectiveness of the techniques presented. We evaluate the correctness and efficiency of our tool against other solvers, as well as against different configurations of the tool in order to demonstrate the efficacy of our heuristics.

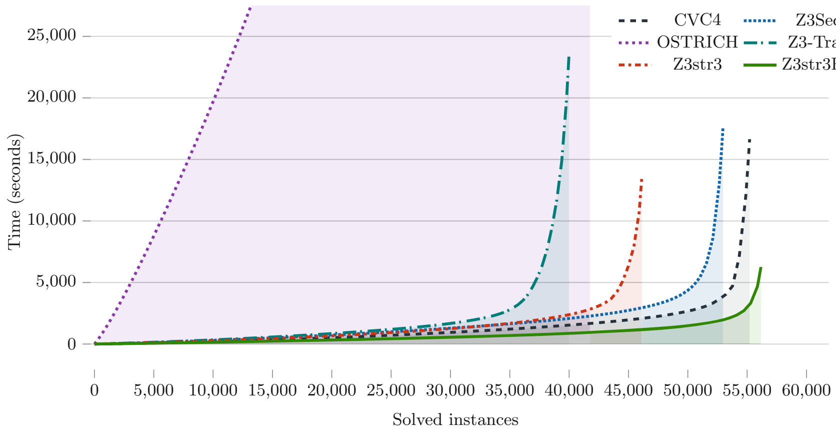

Fig. 2. Cactus plot summarizing performance on all benchmarks. Z3str3RE has the best overall performance.

Table 1. Combined results of string solvers on all benchmarks. **Z3str3RE** has the best overall performance on all benchmarks compared to CVC4, OSTRICH, Z3seq, Z3str3, and Z3-trau and the biggest lead with a score of 1.02.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	33310	31550	22499	24133	27563	**33820**
Unsat	21897	21411	19281	21038	18566	**22339**
Unknown	**0**	**0**	10901	6504	1164	291
Timeout	2049	4295	4575	5581	9963	**806**
Soundness error	**0**	**0**	28	5325	13	**0**
Program crashes	**0**	**0**	**0**	2477	2	**0**
Total correct	55207	52961	41752	39846	46116	**56159**
Contribution score	95.99	19.87	–	–	–	**145.07**
Time (s)	57625.499	103487.844	305243.413	150288.386	213698.954	**23339.266**
Time w/o timeouts (s)	16645.499	17587.844	213743.413	38668.386	14438.954	**7219.266**

5.1 Empirical Setup and Solvers Used

We compare Z3str3RE against five other leading string solvers available today. CVC4 [24] is a general-purpose SMT solver which reasons about strings and regular expressions algebraically. Z3str3 [8] is the latest solver in the Z3-str family, and uses a reduction to word equations to reason about regular expressions. Z3str3RE is based on Z3str3 except for the length-aware algorithm and heuristics described in Sects. 3 and 4. Z3seq [36] is the Z3 sequence solver, implemented by Nikolaj Bjørner and others at Microsoft Research, as part of the Z3 theorem prover. Z3seq uses a new theory of derivatives for solving extended regular expressions. Z3-Trau [1] is also based on Z3 and uses an automata-based approach known as "flat automata" with both under- and over-approximations. OSTRICH [15] uses a reduction from string functions (including word equations) to a model-checking problem that is

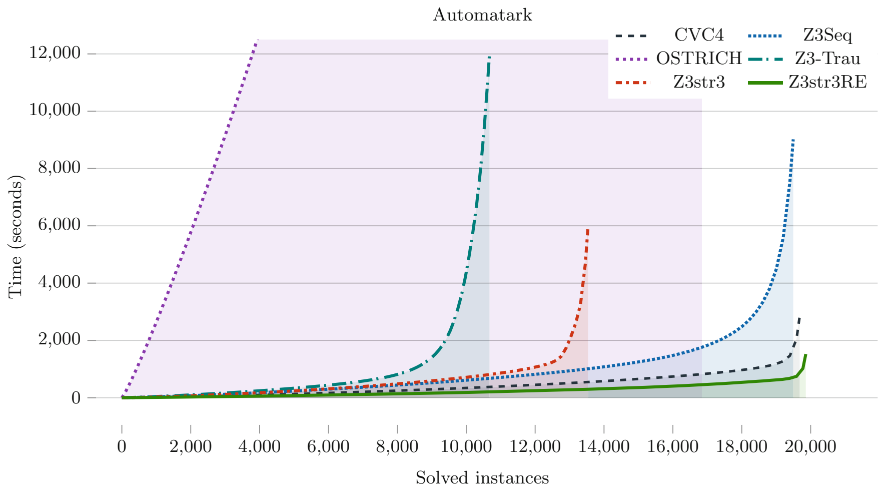

Fig. 3. Cactus plot summarizing detailed performance on Automatark benchmark.

solved using the SLOTH tool and an implementation of IC3. We used CVC4's binary version 1.8, commit `59e9c87` of Z3str3, the sequence solver included in Z3's binary version 4.8.9, Z3-Trau commit `1628747`, and OSTRICH version 1.0.1. All of these tools support the full SMT-LIB standard for strings. We did not compare against the Z3str2 [42] or Norn [3] solvers as neither tool supports the `str.to_int` or `str.from_int` terms which represent string-number conversion, which are used in some sanitizer benchmarks. Additionally, Norn does not support many of the other high-level string terms such as `indexof` or `substr` which are used in the benchmarks. The ABC [4] solver handles string and length constraints by conversion to automata. However, their method over-approximates the solution set of the input formula which may be unsound. Thus, we excluded ABC from our evaluation. We also were unable to evaluate against Trau [2] as the provided source code did not compile. All evaluations were performed on a server running Ubuntu 18.04.4 LTS with two AMD EPYC 7742 processors and 2TB of memory using the ZaligVinder [23] benchmarking framework. A 20 s timeout was used. We cross-verified the models generated by each solver for satisfiable instances against all competing solvers.

5.2 Benchmarks

The comparison was performed on four suites of regex-based benchmarks with a total of 57256 instances. In total, almost 75% of the instances in our evaluation came from previously published industrial benchmarks or other solver developers. Under 10% contain extended regular expressions (having either complement or intersection, or both) and 53% contain only regex predicates. Only 201 instances fall into the undecidable theory $T_{LRE,n,c}$. More details can be found in [7] where we analyse the benchmarks in greater detail. We briefly describe each benchmark's origin and composition.

Table 2. Detailed results for the Automatark benchmark. Z3str3RE has the biggest lead with a score of 1.01.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	14376	14204	11461	8157	9151	**14437**
Unsat	5304	5290	5381	3817	4385	**5422**
Unknown	1	**0**	15	5045	406	**0**
Timeout	298	485	3122	2960	6037	**120**
Soundness error	**0**	**0**	**0**	1300	**0**	**0**
Program crashes	**0**	**0**	**0**	1063	2	**0**
Total correct	19680	19494	16842	10674	13536	**19859**
Contribution score	1.0	1.0	**2.0**	–	0.0	0.5
Time (s)	8789.425	18718.425	158910.126	80021.352	126825.967	**3925.150**
Time w/o timeouts (s)	2829.425	9018.425	96470.126	20821.352	6085.967	**1525.150**

AutomatArk is a set of 19979 benchmarks based on a collection of real-world regex queries collected by Loris D'Antoni from the University of Wisconsin, Madison, USA. We translated the provided regexes [16] into SMT-LIB syntax resulting in two sets of instances: a "simple" set with a single regex membership predicate per instance, and a "complex" set with 2–5 regex membership predicates (possibly negated) over a single variable per instance. The instances in this benchmark are evenly divided between simple and complex problems.

RegEx-Collected is a set of 22425 instances taken from existing benchmarks with the purpose of evaluating the performance of solvers against real-world regex instances. This benchmark includes all instances from the AppScan [41], BanditFuzz,[5] JOACO [38], Kaluza [33], Norn [3], Sloth [21], Stranger [40], and Z3str3-regression [8] benchmarks in which at least one regex membership constraint appears.[6] No additional restrictions are placed on which instances were chosen besides the presence of at least one regex membership predicate. This benchmark tests solvers against challenging instances from widely distributed benchmark suites. Additionally, these instances may contain regex terms in any context and with any other supported string operators. As a result, the benchmark is also exemplary of how string solvers perform in the presence of operations and predicates that are relevant to program analysis.

StringFuzz-regex-generated is a set of 4170 problems generated by the StringFuzz string instance fuzzing tool [12]. These instances only contain regular expression and linear arithmetic constraints. This benchmark isolates the regex performance of a string solver in the context of mixed regex and arithmetic constraints. Tools with better regex and arithmetic solvers should perform better. Fuzz testing, as performed in the **StringFuzz-regex-generated** benchmark, has been shown to be extremely productive in discovering bugs and performance

[5] The BanditFuzz benchmark is an unpublished suite obtained via private communication with the authors.

[6] Other benchmark suites available to us, including the PyEx, PISA, and Kausler benchmarks, did not include any regex membership constraints.

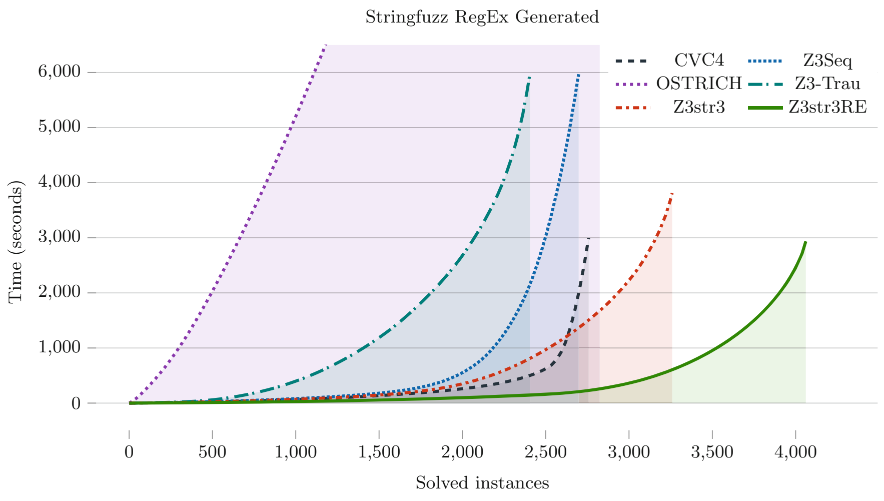

Fig. 4. Cactus plot showing detailed results for the StringFuzz-regex-generated benchmark.

Table 3. Detailed results for the StringFuzz-regex-generated benchmark. Z3str3RE has the biggest lead with a score of 1.25.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	2316	2001	2005	1590	3227	**3231**
Unsat	442	697	819	824	32	**830**
Unknown	**0**	**0**	1	192	**0**	**0**
Timeout	1412	1472	1345	1564	911	**109**
Soundness error	**0**	**0**	**0**	8	**0**	**0**
Program crashes	**0**	**0**	**0**	192	**0**	**0**
Total correct	2758	2698	2824	2406	3259	**4061**
Contribution score	0.0	**3.17**	2.0	–	0.0	0.17
Time (s)	31236.207	35409.000	51571.800	37323.550	22031.636	**5116.456**
Time w/o timeouts (s)	2996.207	5969.000	24671.800	6043.550	3811.636	**2936.456**

issues in SMT solvers. We included these instances because they exercise the performance of the solver on regex-heavy constraints in a way that the industrial benchmarks or instances obtained from other solver developers cannot.

StringFuzz-regex-transformed is a set of 10682 instances which were produced by transforming existing industrial instances with StringFuzz. We applied StringFuzz's transformers to instances supplied by Amazon Web Services related to security policy validation, handcrafted instances inspired by real-world input validation vulnerabilities, and the regex test cases in Z3str3's regression test suite. The instances contain regex constraints, arithmetic and length constraints, string-number conversion (*numstr*), string concatenation, word equations, and other high-level string operations such as `charAt`, `indexof`, and `substr`. As is

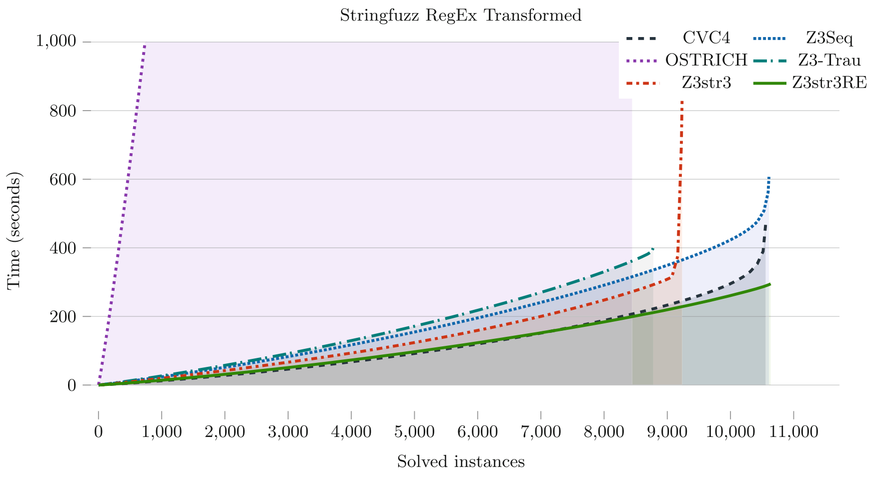

Fig. 5. Cactus plot showing detailed results for the StringFuzz-regex-transformed benchmark.

Table 4. Detailed results for the StringFuzz-regex-transformed benchmark. Z3str3RE has the biggest lead with a score of 1.0.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	4541	**4633**	3899	3672	4417	4599
Unsat	6016	5976	4549	**6282**	4817	6037
Unknown	**0**	**0**	2233	721	**0**	6
Timeout	125	73	**1**	7	1448	40
Soundness error	**0**	**0**	5	1241	**0**	**0**
Program crashes	**0**	**0**	**0**	718	**0**	**0**
Total correct	10557	10609	8443	8713	9234	**10636**
Contribution score	0.5	0.0	–	–	0.0	**4.83**
Time (s)	2969.643	2066.935	23094.737	**722.545**	29788.245	1095.209
Time w/o timeouts (s)	469.643	606.935	23074.737	582.545	828.245	**295.209**

typical for fuzzing in software testing, the goal is to create a suite of tests from a given input that are similar in structure but that explore interesting behaviour not captured by a "typical" industrial instance. These transformed instances are often harder than the original industrial ones.

5.3 Comparison and Scoring Methods

We compare solvers directly against the total number of correctly solved cases, total time with and without timeouts, and total number of soundness errors and program crashes. We also computed the biggest lead winner and largest contribution ranking following the scoring system used by the SMT Competition [6]. Briefly, the biggest lead measures the proportion of correct answers of the leading tool to correct answers of the next ranking tool, and the contribution score measures what proportion of instances were solved the fastest by that solver. In accordance with the SMT Competition guidelines, a solver receives no contribution score (denoted as –) if it produces any incorrect answers on a given benchmark. In both cases, higher scores are better.

5.4 Analysis of Empirical Results

The cactus plot in Fig. 2 shows the cumulative time taken by each solver on all cases in increasing order of runtime. Solvers that are further to the right and closer to the bottom of the plot have better performance.

Overall Z3str3RE solves more instances and performs better than all competing solvers. Across all benchmarks, Z3str3RE is over 2.4× faster than CVC4, 4.4× faster than Z3seq, 6.4× faster than Z3-Trau, 9.1× faster than Z3str3, and 13× faster than OSTRICH (including timeouts). Additionally, Z3str3RE has fewer combined timeouts and unknowns than other tools considered, and no soundness errors or crashes. We summarize these results in Table 1. Notably, both Z3-Trau [1] and OSTRICH [15] had significant runtime issues in our experiments. Z3-Trau produced 5325 soundness errors and 2477 crashes on our benchmarks (13% of all instances), which is significantly higher than other tools used. OSTRICH produced 10901 "unknown" responses on the benchmarks (19% of all instances), due to both unsupported features and crashes, and also produced 28 soundness errors. Over all benchmarks, Z3str3RE produced 291 unknowns. There are several potential reasons for this; the solver may have encountered a resource limit and returned UNKNOWN, or it may have detected non-termination and returned UNKNOWN instead of looping forever. According to SMT Competition scoring, Z3str3RE won the division across all benchmarks with a lead of 1.02, and had the largest contribution to the division with a score of 145.07. CVC4 had a contribution score of 95.99, and Z3seq had a score of 19.87. OSTRICH, Z3-Trau, and Z3str3 received no contribution score as they each returned at least one incorrect answer. The presented results are typical of the performance of the evaluated tools over multiple runs. Results were cross-validated within runs and between multiple runs. For a random single instance, the sample variance in execution time for 100 runs is 0.001 (0.07% of average execution time). Over 57256 instances, this is negligible.

The empirical results make clear the efficacy of length-aware automata-based techniques for regular expression constraints when accompanied with length constraints (which is typical for industrial instances). The effectiveness of our technique is demonstrated particularly by comparing Z3str3RE with Z3str3, as the only differences between these tools are the length-aware regex algorithm and heuristics implemented in Z3str3RE and bug fixes. By improving the regex algorithm and applying our heuristics, we achieved a speedup of over 9x and solved over 10000 more cases than Z3str3.

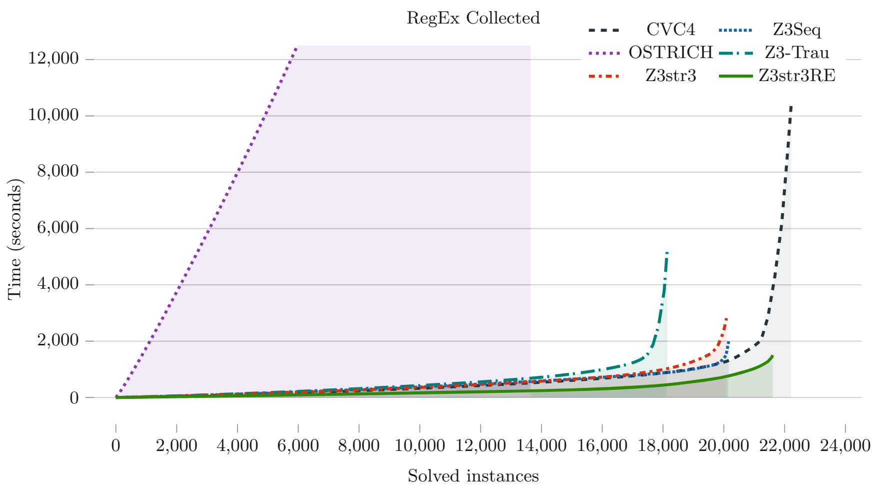

Fig. 6. Cactus plot showing detailed performance for the RegEx-Collected benchmark.

Table 5. Detailed results for the RegEx-Collected benchmark. CVC4 has the biggest lead with a score of 1.03.

	CVC4	Z3Seq	OSTRICH	Z3-Trau	Z3str3	Z3str3RE
Sat	**12077**	10712	5134	10714	10768	11553
Unsat	**10135**	9448	8532	10115	9332	10050
Unknown	**0**	**0**	8652	546	758	285
Timeout	213	2265	**107**	1050	1567	537
Soundness error	**0**	**0**	23	2776	13	**0**
Program crashes	**0**	**0**	**0**	504	**0**	**0**
Total correct	**22212**	20160	13643	18053	20087	21603
Contribution score	**91.06**	3.51	–	–	–	14.54
Time (s)	14610.224	47293.484	71666.750	32220.939	35053.106	**13202.451**
Time w/o timeouts (s)	10350.224	**1993.484**	69526.750	11220.939	3713.106	2462.451

5.5 Detailed Experimental Results

Figure 3 and Table 2 show the detailed results for the **AutomatArk** benchmark. In this benchmark, Z3str3RE solves more instances than all other solvers, has the fewest timeouts/unknowns, and has the fastest overall running time. Including timeouts, Z3str3RE is 2.2× faster than CVC4, 4.7× faster than Z3seq, 40.4× faster than OSTRICH, 20.4× faster than Z3-Trau, and 32.3× faster than Z3str3.

Figure 4 and Table 3 show the detailed results for the **StringFuzz-regex-generated** benchmark. Z3str3RE solves more instances than all other solvers, has over 90% fewer timeouts than other solvers, no unknowns, and has the fastest overall running time. Including timeouts, Z3str3RE is 6.1× faster than CVC4, 6.9× faster than Z3seq, 10× faster than OSTRICH, 7.3× faster than Z3-Trau, and 4.3× faster than Z3str3.

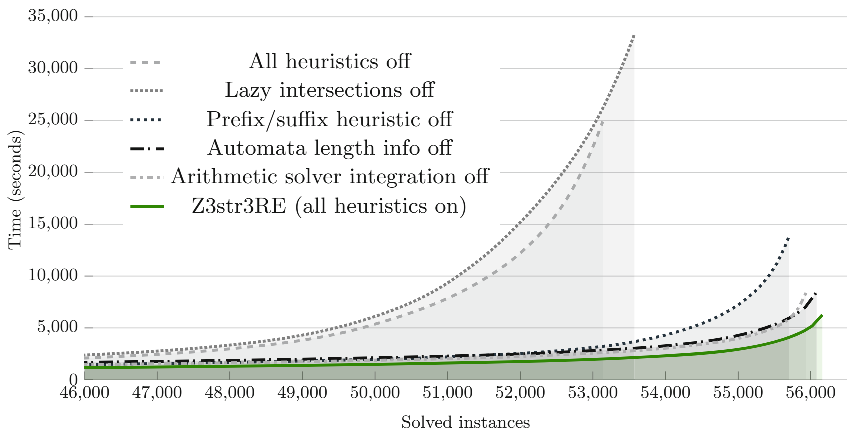

Fig. 7. Cactus plot comparing performance by disabling individual heuristics on all benchmarks.

Figure 5 and Table 4 show the detailed results for the **StringFuzz-regex-transformed** benchmark. Z3str3RE solves more instances in total than all other solvers and has the lowest total running time without timeouts. Including timeouts, Z3str3RE is 2.7× faster than CVC4, 1.9× faster than Z3seq, 21× faster than OSTRICH, and 27× faster than Z3str3. Although Z3-Trau is 1.5× faster than Z3str3RE on this benchmark, including timeouts, Z3-Trau also produces 1241 answers with soundness errors and crashes on 718 other cases. Z3str3RE produces no wrong answers or soundness errors on the benchmark. Z3-Trau also solves 1923 fewer cases correctly in total than Z3str3RE.

Figure 6 and Table 5 show the detailed results for the **RegEx-Collected** benchmark. Z3str3RE outperforms Z3seq, Z3str3, OSTRICH, and Z3-Trau on this benchmark and is competitive with CVC4 both in terms of total number of instances correctly solved and total running time. CVC4 solves 609 more instances than Z3str3RE on this benchmark, but Z3str3RE is 1.1× faster overall (including timeouts). Z3str3RE is 3.6× faster than Z3seq, 5.4× faster than OSTRICH, 2.4× faster than Z3-Trau, and 2.6× faster than Z3str3.

5.6 Analysis of Individual Heuristics and Results

To demonstrate the effectiveness of individual heuristics described in Sect. 4 and implemented in Z3str3RE, we evaluated different configurations of the tool in which one or more heuristics were disabled. Figure 7 and Table 6 show the results. The plot line "Z3str3RE" shows the performance of the tool with all heuristics enabled. The plot line "All heuristics off" shows the performance with all heuristics disabled. Each of the other plot lines shows the performance with the named heuristic disabled and all others kept enabled. From the plots and table, it is clear that Z3str3RE performs best with all heuristics enabled. Z3str3RE is 4.4× faster using all our heuristics than using none. Every other configuration of the

Table 6. Comparison of different heuristics in Z3str3RE on all benchmarks.

	All off	Lazy intersection off	Prefix/suffix off	Automata length info off	Arith. solver integ. off	Z3str3RE
Sat	31046	31486	33817	33816	33804	**33820**
Unsat	22090	22085	21880	22264	22131	**22339**
Unknown	313	323	287	285	**283**	291
Timeout	3807	3362	1272	891	1038	**806**
Soundness error	**0**	**0**	**0**	**0**	**0**	**0**
Program crashes	42	39	**0**	1	**0**	**0**
Total correct	53136	53571	55697	56080	55935	**56159**
Time (s)	102102.388	101799.263	40068.501	27178.746	30006.857	**23339.266**
Time w/o timeouts (s)	25962.388	34559.263	1462.8501	9358.746	9246.857	**7219.266**

tool performs significantly worse relative to the one with all heuristics enabled. Also, the length-aware and prefix/suffix heuristics provide significant boost over lazy intersections and the baseline. These results demonstrate empirically that each heuristic we introduce provides significant benefit in both total number of solved instances and total solver runtime, and that all of the heuristics can be used simultaneously for maximum efficacy.

6 Related Work

Comparison with Z3str3: Z3str3 [8] supports regex constraints via (incomplete) reduction to word equations. We have replaced this word-based technique with our automata-based approach introduced in this paper. As demonstrated by our evaluation, the length-aware automata-based approach used in Z3str3RE is more efficient at solving these constraints, and is sound and complete for the QF theory T_{LRE}.

Comparison with Z3's Sequence Solver: Z3's sequence solver [18] supports a more general theory of "sequences" over arbitrary datatypes, which allows it to be used as a string solver. Z3seq uses regular expression derivatives to reduce regex constraints without constructing automata. The experiments show Z3str3RE performs better than Z3seq overall.

Comparison with CVC4: The CVC4 solver [24] uses an algebraic approach to solving regex constraints. As shown in the experiments, Z3str3RE performs better than CVC4, widely considered as one of the best SMT solvers for strings as well as many other theories.

Comparison with Z3-Trau: The Z3-Trau [1] solver builds on Trau [2], re-implemented in Z3, and enriched with new ideas e.g. a more efficient handling of string-number conversion. The evaluation of Z3-Trau exposed 5325 soundness errors and 2477 crashes on our benchmarks.

Comparison with OSTRICH: The OSTRICH solver [15] implements a reduction from straight-line and acyclic fragments of an input formula to the emptiness problem of alternating finite automata. OSTRICH produced 10901 "unknown" responses and 4575 timeouts on our benchmarks, as well as 28 soundness errors.

Related Algorithms and Theoretical Results: The theory of word equations and various extensions have been studied extensively for many decades. In 1977, Makanin proved that satisfiability for the QF theory of word equations is decidable [28]; in 1999, Plandowski showed that this is in PSPACE [30,31]. Schulz [34] extended Makanin's algorithm to word equations with regex constraints. The satisfiability problem for the theory of word equations with length constraints still remains open [20,28,29,31], although the status of many other extensions of this theory was clarified [17]. Automata-based approaches were used to reason about string constraints enhanced with a ReplaceAll function [14] or transducers [21].

Liang et al. [25] present a formal calculus for a theory that extends T_{LRE} with string concatenation (but not word equations). However, in that paper the authors do not present experimental results regarding implementation of the string calculus proposed. We have implemented an algorithm based on fundamentals of the theory and standard automata-based constructions, and presented a thorough experimental evaluation of our implementation.

Abdulla et al. [3] present an automata-based solver called Norn built upon results involving construction of length constraints from regex constraints. This approach differs significantly from our method. In particular, Norn only uses automata in inferring length constraints implied by regular expressions, then uses an algebraic approach to solve the remainder of the formula. By contrast, our tool uses a hybrid approach that includes both algebraic solving and automata-based reasoning in a symbiotic loop. In addition, we present several novel heuristics using length information to guide the search and, in some cases, avoid constructing automata or computing intersections.

The prefix/suffix over-approximation heuristic is inspired partly by the work of Brzozowski on regex derivatives [13]. The heuristic we introduce is conceptually different as we examine possible prefixes (and suffixes) of strings that could be accepted by a regex in order to demonstrate unsatisfiability, rather than examining the set of all possible suffixes given a fixed prefix in order to demonstrate satisfiability. Our heuristic computes suffixes as well, whereas Brzozowski derivatives are traditionally computed with respect to prefixes of a string. Newer versions of Z3seq, including the one we evaluated, use a regex algorithm based on symbolic derivatives [36].

7 Conclusions and Future Work

In this paper, we empirically showcase the power of length-aware and prefix/suffix reasoning for regex constraints with our algorithm and its implementation in Z3str3RE via an extensive empirical comparison against five other state-of-the-art solvers (namely, CVC4, Z3seq, Z3str3, Z3-Trau, and OSTRICH) over a large and diverse benchmark of 57256 instances. Over this entire benchmark suite, we show that Z3str3RE has a speedup of 2.4× over CVC4, 4.4× over Z3seq, 6.4× over Z3-Trau, 9.1× over Z3str3, and 13× over OSTRICH. Our length-aware method is very general and has wide applicability in the broad context of string solving. In the future, we plan to explore further length-aware heuristics which include more expressive functions and predicates, including `indexof`, `substr`, and string-number conversion.

Acknowledgments. The work of Federico Mora is supported by NSF grants CNS-1739816 and CCF-1837132, by the DARPA LOGiCS project under contract FA8750-20-C-0156, by the iCyPhycenter, and by gifts from Intel, Amazon, and Microsoft. The work of Florin Manea is supported by the DFG grant 389613931.

References

1. Abdulla, P.A., et al.: Efficient handling of string-number conversion. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 943–957 (2020)
2. Abdulla, P.A., et al.: Flatten and conquer: a framework for efficient analysis of string constraints. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pp. 602–617 (2017)
3. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 150–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_10
4. Aydin, A., Bang, L., Bultan, T.: Automata-based model counting for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 255–272. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_15
5. Backes, J., et al.: One-click formal methods. IEEE Softw. **36**(6), 61–65 (2019)
6. Barbosa, H., Hoenicke, J., Hyvarinen, A.: 15th international satisfiability modulo theories competition (SMT-COMP 2020): rules and procedures (2020). https://smt-comp.github.io/2020/rules20.pdf
7. Berzish, M., et al.: String theories involving regular membership predicates: from practice to theory and back (2021)
8. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 55–59 (2017)
9. Berzish, M., et al.: A length-aware regular expression SMT solver (2020). https://arxiv.org/abs/2010.07253
10. Bjørner, N., Ganesh, V., Michel, R., Veanes, M.: An SMT-LIB format for sequences and regular expressions. In: SMT workshop 2012 (2012)
11. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2009, pp. 307–321 (2009). https://doi.org/10.1007/978-3-642-00768-2_27
12. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: StringFuzz: a fuzzer for string solvers. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10982, pp. 45–51. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_6

13. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
14. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replace all function. Proc. ACM Program. Lang. **2**(POPL), 3:1–3:29 (2018)
15. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. In: Proceedings of the ACM on Programming Languages **3**(POPL), 1–30 (2019)
16. D'Antoni, L.: Automatark automata benchmark (2018). https://github.com/lorisdanto/automatark
17. Day, J.D., Ganesh, V., He, P., Manea, F., Nowotka, D.: The satisfiability of word equations: decidable and undecidable theories. In: Potapov, I., Reynier, P.-A. (eds.) RP 2018. LNCS, vol. 11123, pp. 15–29. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00250-3_2
18. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR abs/1605.09442 (2016). http://arxiv.org/abs/1605.09442
20. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: what's decidable? In: Biere, A., Nahir, A., Vos, T. (eds.) HVC 2012. LNCS, vol. 7857, pp. 209–226. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39611-3_21
21. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL), 4:1–4:32 (2018)
22. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116 (2009)
23. Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: The power of string solving: simplicity of comparison. In: 2020 IEEE/ACM 1st International Conference on Automation of Software Test (AST), pp. 85–88. IEEE/ACM (2020)
24. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 646–662. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_43
25. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: Lutz, C., Ranise, S. (eds.) FroCoS 2015. LNCS (LNAI), vol. 9322, pp. 135–150. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24246-0_9
26. Lin, A.W., Majumdar, R.: Quadratic word equations with length constraints, counter systems, and Presburger arithmetic with divisibility. In: Lahiri, S.K., Wang, C. (eds.) ATVA 2018. LNCS, vol. 11138, pp. 352–369. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01090-4_21
27. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, 20–22 January 2016, pp. 123–136. ACM (2016)
28. Makanin, G.: The problem of solvability of equations in a free semigroup. Math. Sbornik **103**, 147–236 (1977). English transl. in Math USSR Sbornik 32 (1977)

29. Matiyasevich, Y.: Word equations, fibonacci numbers, and Hilbert's tenth problem. In: Workshop on Fibonacci Words (2007)
30. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3), 483–496 (2004)
31. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the 38th Annual ACM Symposium on Theory of Computing, STOC 2006, pp. 467–476 (2006)
32. Redelinghuys, G., Visser, W., Geldenhuys, J.: Symbolic execution of programs with strings. In: Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference, SAICSIT 2012, pp. 139–148 (2012)
33. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: Proceedings of the 2010 IEEE Symposium on Security and Privacy, SP 2010, pp. 513–528 (2010)
34. Schulz, K.U.: Makanin's algorithm for word equations-two improvements and a generalization. In: Schulz, K.U. (ed.) IWWERT 1990. LNCS, vol. 572, pp. 85–150. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55124-7_4
35. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 488–498. ACM, New York (2013)
36. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. Technical report. MSR-TR-2020-25, Microsoft, August 2020. https://www.microsoft.com/en-us/research/publication/symbolic-boolean-derivatives-for-efficiently-solving-extended-regular-expression-constraints/
37. Stockmeyer, L.J.: The Complexity of Decision Problems in Automata Theory and Logic. Ph.D. thesis, MIT (1974)
38. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.: An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving. IEEE TSE (2018)
39. Trinh, M.-T., Chu, D.-H., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 218–240. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_12
40. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: an automata-based string analysis tool for PHP. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 154–157. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12002-2_13
41. Zheng, Y., et al.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods Syst. Des., 1–40 (2016)
42. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Dolby, J., Zhang, X.: Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 235–254. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_14

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.