Effective Search-space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints

Yunhui Zheng[1], Vijay Ganesh[2],
Sanu Subramanian[2], Omer Tripp[1], Julian Dolby[1], and Xiangyu Zhang[3]

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

Abstract. In recent years, string solvers have become an essential component in many formal-verification, security-analysis and bug-finding tools. Such solvers typically support a theory of string equations, the length function as well as the regular-expression membership predicate. These enable considerable expressive power, which comes at the cost of slow solving time, and in some cases even nontermination. We present two techniques, designed for word-based SMT string solvers, to mitigate these problems: (i) sound and complete detection of overlapping variables, which is essential to avoiding common cases of nontermination; and (ii) pruning of the search space via bi-directional integration between the string and integer theories, enabling new cross-domain heuristics. We have implemented both techniques atop the Z3-str solver, resulting in a significantly more robust and efficient solver, dubbed Z3str2, for the quantifier-free theory of string equations, the regular-expression membership predicate and linear arithmetic over the length function. We report on a series of experiments over four sets of challenging real-world benchmarks, where we compared Z3str2 with five different string solvers: S3, CVC4, Kaluza, PISA and Stranger. Each of these tools utilizes a different solving strategy and/or string representation (based e.g. on words, bit vectors or automata). The results point to the efficacy of our proposed techniques, which yield dramatic performance improvement. We argue that the techniques presented here are of broad applicability, and can be integrated into other SMT-backed string solvers to improve their performance.

1 Introduction

Reasoning over strings is gaining increasing importance due to the security threats imposed by improper handling of untrusted string values [7,19,27]. In response, different powerful string solvers have been developed, including, e.g., HAMPI [19], Kaluza [28], PISA [30], Stranger [32], CVC4 [22], S3 [31], Norn [6] and Z3-str [35]. These tools primarily solve the satisfiability problem over string (aka word) equations, with some of them also providing support for regular-expression (RE) membership predicates and linear arithmetic over the length function. While these tools have improved dramatically in recent years, the demand for even more efficient solvers continues to grow unabated.

Motivated by this need for efficient string solvers, we present two new techniques to solve combined string, regular-expression and integer constraints. These techniques are applicable primarily to SMT solvers that treat strings without abstractions or representation conversions, which we refer to collectively as *word-based string solvers*. Examples of such solvers include the Z3-str, CVC4 and S3 string solvers.

For the sake of completeness, we compare and contrast our techniques against solvers that use automata (e.g., PISA and Stranger) and bit-vector (e.g., Kaluza) string representations. Word-based string solvers have several important advantages: First, unlike bit-vector-based solvers, they can precisely model unbounded strings and string equalities without over-approximation, a feature that is crucial to string analysis of web applications. Second, by modeling strings and length in native domains, word-based string solvers can leverage the state of the art in integer constraint solving, and further enable hybrid techniques via powerful SMT engines. Finally, such solvers can take advantage of well-developed application-specific rewrite rules.

At the same time, a fundamental problem of word-based string solvers (unlike those based on bit vectors, which impose a finite domain) is that it is unclear, at present, whether the satisfiability problem for the quantifier-free theory of word equations, regular-expression membership predicate and length function is decidable [24]. All current practical string solvers suffer from incompleteness and nontermination. Addressing these problems is of primary importance, calling for new techniques to effectively explore the solution space. In light of this motivation, we have developed two techniques that address the respective problems of nontermination and search-space explosion.

First, a well-known reason for nontermination is overlapping variables [11,35,6], as we illustrate with the equation $a \cdot X = X \cdot b$, where a, b are constant strings and X is a string variable. Stated intuitively, the solution for X has to be in the form of $a \cdot X_1 \cdot b$, where X_1 is a string variable. The reduction step results in $a \cdot X_1 = X_1 \cdot b$, which is in the same form as the original equation, and thus leads to nontermination. However, this equation is obviously unsatisfiable. We revisit this example in section 3, which highlights the need for a robust procedure to detect overlapping variables.

The second technique, given the tight interplay between string and integer values (in index-sensitive string operations), is bi-directional solver-level integration between the string and integer theories. This can be leveraged to drastically reduce the search space for typical constraints obtained from practical applications.

We have implemented both of these techniques atop the Z3-str solver as the Z3str2 solver for the satisfiability problem over a quantifier-free theory of word equations, regular-expression membership predicate as well as the length function. We report on a comprehensive set of experiments that validate the efficacy of our proposed techniques by comparing Z3str2 with Kaluza, PISA, Stranger, S3 and CVC4 over four sets of benchmarks derived from the real world.[1] We emphasize that our techniques are applicable also to other word-based string solvers such as S3 and CVC4.

Contributions To summarize, this paper makes the following principal contributions:

1. Guided search: We present two techniques designed for string solvers that treat strings as primitive types. The first is a sound and complete method to detect overlapping variables, which optimizes performance and avoids exploration of certain paths that may lead to nontermination. The second technique is a two-way integration between the string and integer theories, which enables effective pruning based on cross-domain heuristics.

2. Z3-str integration: We have integrated both of the aforementioned techniques into the core solving algorithm of Z3-str. We describe the architecture of the resulting tool, Z3str2, and prove its soundness.

[1] The Z3str2 code, as well as the data pertaining to our experiments, are all available at [1].

3. Experimental study: To validate the efficacy of our techniques, we have conducted a comprehensive set of experiments where we compare Z3str2 against five solvers — namely, S3, CVC4, Kaluza, PISA and Stranger — on four benchmark suites. The results show that Z3str2 is significantly faster than competing solvers (often by orders of magnitude) in all but few cases.

1.1 Related Work

The theory considered in this paper – namely the quantifier-free (QF) theory T_{wlr} over word equations, membership predicate over REs, and length function – is a multi-sorted theory with string (str) and numeric (num) sorts. Makanin was the first to show, in 1977, that the QF theory of word equations is decidable [23]. Since, many have improved upon this seminal result [29,16,25,26,15]. In particular, Plandowski proved that this problem is in PSPACE [26]. Despite decades of effort, the satisfiability problem for T_{wlr} remains open [23,26,11,15]. Still, many practical solvers have been proposed.

Automata-based Solvers. Regular languages (or automata), as well as context-free grammars (CFGs), can be used to represent strings and handling regex-related operations. JSA [8] computes CFGs for string variables in Java programs. Hooimeijer et al. [13] suggest an optimization whereby automata are built lazily. A primary challenge faced by automata-based approaches, which we do not suffer from, is to capture the connections between strings and other domains, e.g., integers. To overcome this limitation, refinements have been proposed. JST [12] extends JSA. It asserts length constraints in each automaton, and handles numeric constraints after conversion. PISA [30] encodes Java programs into M2L formulas that it discharges to the MONA solver to obtain path- and index-sensitive string approximations. PASS [21,20] combines automata and parametrized arrays for efficient treatment of unsat cases. Stranger is a powerful extension of string automata with arithmetic automata [32,33].

Bit-vector based solvers. Another group of solvers converts string constraints to constraints into other domains such as integers or bit-vectors. HAMPI [19] is an efficient solver that represents strings as bit-vectors, though it requires the user to provide an upper bound on string lengths. Early versions of Kaluza [28] extended both STP [10] and HAMPI to support mixed string and numeric constraints represented as bit-vector. A similar approach powers Pex [7], though strings are reduced to integer abstractions.

Word-based string solvers. CVC4 [22] handles constraints over the theory of unbounded strings with length and RE membership. Solving is based on multi-theory reasoning backed by the DPLL(T) architecture combined with existing SMT theories. The Kleene star operator in RE formulas is dealt with via unrolling as in Z3str2. S3 [31] is another word-based solver, and it can be viewed as an extension of an early version of Z3-str. Roughly speaking, CVC4, S3 and Z3str2 embody similar approaches, and hence CVC4 and S3 can also benefit from the techniques proposed in this paper.

1.2 Formal Preliminaries

Syntax of Word Equations, RE membership, and Length. We fix a disjoint two-sorted set of variables $var = var_{str} \cup var_{int}$; var_{str} consists of string variables, denoted $X, Y, S, \ldots$ and var_{int} consists of integer variables, denoted $m, n, \ldots$. We also

fix a two-sorted set of constants $Con = Con_{str} \cup Con_{int}$. Moreover, $Con_{str} \subset \Sigma^*$ for some finite alphabet, Σ, whose elements are denoted $f, g, \ldots$. Elements of Con_{str} will be referred to as *string constants* or *strings*. Elements of Con_{int} are nonnegative integers. The empty string is represented by ϵ, and length 0. Terms may be string terms or integer terms. A string term is either an element of var_{str}, an element of Con_{str}, or a concatenation of string terms (denoted by the function $concat$ or interchangeably by $\cdot$). An integer term is an element of var_{int}, an element of Con_{int}, the length function applied to a string term, a constant integer multiple of a integer term, or a sum of integer terms. The theory contains three types of atomic formulas, namely, word equations, length constraints, and RE membership predicates. REs are defined inductively, where constants and the empty string form the base case, and the operations of concatenation, alternation, and Kleene star are used to build up more complicated expressions (see details in [14]). REs may not contain variables. Z3str2 supports a list of common string-related operators such as charAt, containts, startswith, endswith, indexof, lastindexof, substring and etc. They are desugared to word equations with length functions. Formulas are defined inductively over atomic formulas and are quantifier-free.

Semantics of Word Equations, RE membership, and Length. For a word, w, $len(w)$ denotes the length of w. The universe of discourse for the str sort is the set of strings Σ^*, and for the int sort is the set of natural numbers. For a word equation $t_1 = t_2$, we refer to t_1 as the left hand side (LHS), and t_2 as the right hand side (RHS). We fix a string alphabet, Σ. Given a formula θ, an *assignment* for θ (with respect to Σ) is a map from the set of variables in θ to $\Sigma^* \cup \mathbb{N}$ (where string variables are mapped to strings and integer variables are mapped to numbers). Given such an assignment, θ can be interpreted as an assertion about Σ^* and $\mathbb{N}$. If this assertion is true, then θ is *satisfiable* or SAT. A formula with no satisfying assignment is *unsatisfiable* or UNSAT. Two formulas θ, ϕ are *equisatisfiable* if θ is SAT iff ϕ is SAT. The *satisfiability problem* for a set S of formulas is to decide whether any given formula in S is SAT or not. The satisfiability problem for a set of formulas is decidable if there exists an algorithm (or *satisfiability procedure*) that solves its satisfiability problem. Satisfiability procedures must have three properties: soundness, completeness, and termination. Soundness and completeness guarantee that the procedure returns SAT if and only if the input formula is indeed SAT. More precisely, the procedure is sound if the procedure says UNSAT then the input is indeed unsatisfiable. Completeness is the converse of soundness.

2 Overview of the design Z3str2 String Solver

The Z3str2 solver is essentially a string plug-in built into the Z3 SMT Solver [9], with an efficient integration between the string plug-in and Z3's integer solver. As can be seen from the architectural schematic of the Z3str2 string solver given in Figure 1 (and an algorithmic description is given in Algorithm 1), the first step is to purify the input into two, namely, string constraints (word equations and RE membership) on the one hand, and integer linear arithmetic constraints over the length function on the other. Next, the word equations and RE constraints are input to the string plug-in. The plug-in may consult the Z3 core to detect equivalent terms. The word equations are solved using an algorithm described in detail in the section 3 below. The RE constraints are solved

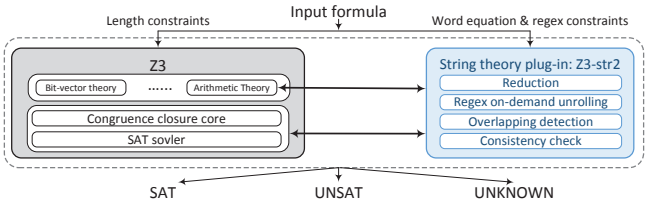


Fig. 1: Architecture of Z3str2.

by unrolling as described also in section 3. The length constraints are converted into a system of pure integer linear arithmetic inequations and solved using Z3's integer solver. During the solving process, the string plug-in may generate length constraints that are incrementally added on demand to Z3's integer solver, that are regularly checked for consistency with both the input length constraints and previously added ones.

On any well-formed input as described in section 1.2, Z3str2 may return SAT, UNSAT or UNKNOWN. Note that while Z3str2 can handle a boolean combination of atomic formulas, we refer only to conjunction of literals in the rest of paper without loss of generality. If either Z3's integer solver or our string plug-in determines that their respective purified inputs are UNSAT, Z3str2 reports UNSAT. If the string plug-in detects that the input equations have complicated overlaps that its heuristics cannot handle, it reports UNKNOWN. This is a source of incompleteness in Z3str2's implementation. Note that Z3str2, like other competing solvers such as CVC4, is sound but not complete.

The third and only remaining possibility is that both Z3's integer solver and the string plug-in determine that their respective purified inputs are SAT. However, this does not necessarily mean that the input is SAT. It could be that the solution produced by the integer solver is inconsistent with the solution produced by the string solver, or vice-versa. For example, the integer solver may say that a particular string variable, say X, has length equal to 1, while the string plug-in might produce a specific assignment for X that is of length equal to 2. One way to overcome this problem is to iterate through all possible solutions until a consistent one is found, assuming one exists. However, given that the domain of strings and natural numbers is infinite, it is possible that such an iterative procedure may loop forever in the event there are no consistent solutions to be found. In other words, if the input is indeed SAT, the procedure discussed here will correctly establish consistency and determine that the input is SAT. Unfortunately, it is possible that, when the input is in fact UNSAT, both the integer solver and string plug-in may determine that their respective purified inputs are SAT and may then loop forever searching for a combined consistent solution. They are obviously not going to find a combined consistent solution in such cases given that the input is in fact UNSAT, and hence the iterative procedure may not terminate.

The above-described problem of non-termination due to the interaction between the integer and string parts of the theory is not specific to Z3str2. In fact, the problem of deciding the satisfiability problem for the quantifier-free theory of word equations and length function remains open after decades of research and is a major open problem in mathematical logic [24]. In conclusion, if Z3str2 reports that the input is UNSAT, then indeed the input is UNSAT (soundness). However, the converse is not necessarily true, i.e., just like all other competing practical solvers, Z3str2 is not complete.

Algorithm 1 A high-level description of Z3str2's Algorithm

Input: Word equations $\mathcal{Q}_w$, and the corresponding integer linear arithmetic constraints $\mathcal{Q}_l$ over the length function
Output: SAT / UNSAT / UNKNOWN

```
1:  procedure solveStringConstraint(Q_w, Q_l)
2:    if equations in Q_w are all in solved form and Q_w is UNSAT or Q_l is UNSAT then
3:      return UNSAT
4:    if equations in Q_w are all in solved form and Q_w and Q_l can be consistently determined as SAT together then
5:      return SAT
6:    Convert Q_w equisatisfiably in disjunctive normal form (DNF) formula Q_a
7:    for each disjunct D in Q_a do
8:      Convert each equation in D to an arrangement consistent with the length constraints from the integer theory
9:      for each string variable x do
10:       Merge per-equation arrangements to a set of possible global arrangements, denoted as G(x)
11:       Detect arrangements with overlaps in G(x)
12:       if there is any overlap then
13:         Prune the global arrangements with the overlap from G(x)
14:     for each global arrangement combination selected from G(x), G(y)..., for all variables x, y,... do
15:       Split each variable to sub-variables according to the selected global arrangement
16:       Convert Q_w equisatisfiably to a system Q'_w of simpler equation based on the new variables
17:       Q'_l is the corresponding new set of length constraints
18:       r = solveStringConstraint(Q'_w, Q'_l)
19:       if r ≡ SAT then
20:         return SAT
21:   if overlapping variables have ever been detected then
22:     return UNKNOWN
23:   else
24:     return UNSAT
```

3 Word Equation Sub-solver in Z3str2

In this subsection, we focus on the word equation solving component of Z3str2. Starting with the work of Makanin [23], many decision procedures [26,29,15] have been proposed. While most procedures are not accompanied by practical implementations, they are a rich source of ideas for all the solvers that have recently been implemented. For example, the Z3str2 solver follows ideas, namely, *boundary labels, generalized word equations and arrangements* (discussed in greater detail in subsection 4.1), that have their roots in the very first decision procedure for word equations by Makanin.

The key technique used by Z3str2 to solve a word equation W is to recursively convert W equisatisfiably into disjunction of conjunctions of simpler equations we call arrangements. These arrangements are computed by aligning the concatenation function on the LHS and RHS of a given equation such that an occurrence of concatenation function in the LHS (resp. RHS) may "split" or "cut" variables on the RHS (resp. LHS).

As an illustration, consider the following formula composed of three equations: $Z = X \cdot Y \ \wedge \ Z = W \cdot c \ \wedge \ c \cdot Y = c \cdot b \cdot c$, where X, Y, Z and W are string variables, and b and c are characters. A simple rewriting is the following:

$$Z = X \cdot Y \Longrightarrow \ Z_1 = X \ \wedge \ Z_2 = Y \ ^{[1.1]}$$
$$Z = W \cdot \ c \Longrightarrow \ Z_1 = W \ \wedge \ Z_2 = c \ ^{[1.2]}$$
$$c \cdot Y = c \cdot b \cdot c \Longrightarrow \ Y = b \cdot c \ ^{[1.3]}$$

Observe that Z is split into Z_1 and Z_2, which are constrained differently. However, this rewriting is not satisfiable because $Y = c$ from equations $[1.1]$ and $[1.2]$, and $Y = b \cdot c$ from equation $[1.3]$. Now observe that the alignment described above is not the only one we can consider. Below is a different alignment that leads to a new splitting and in fact yields a satisfying assignment:

$$Z = W \cdot \ c \Longrightarrow \ Z_1 = W_1 \ ^{[1.4]} \ \wedge \ Z_2 = W_2 \cdot c \ ^{[1.5]}$$

The difference now is that W is also split (into W_1 and W_2), and hence this splitting yields a satisfying assignment. In particular, from $[1.1]$, $[1.3]$ and $[1.5]$, we have $W_2 = b$. Also note that X, Z_1 and W_1 become free variables as they are all equivalent but not constrained by any other variable.

What the above example highlights is that there are many different alignments of variable boundaries in the LHS (resp. RHS) that can split variables in the RHS (resp. LHS). We call every such alignment an *arrangement*. Here is the crucial fact about word equations: every equation can be equisatisfiably rewritten into a finite set of arrangements, where each arrangement is a finite set of word equations obtained from the splitting procedure described above. The Z3str2 solver exploits this fact, and solves word equations by converting them into finite sets of arrangements and inspecting each one individually to see if they are satisfiable. The input word equation is SAT if and only if at least one arrangement is SAT. This in a nutshell is how the word equations are solved by the Z3str2 solver, i.e., by recursively converting equations into a disjunction of arrangements (where each arrangement is a simpler set of equations) until a set of arrangements is derived where the satisfiability can determined purely via inspection.

Supporting Regular Expression Membership Predicates: A RE membership predicate $X \in \mathcal{R}$ is reduced to word equations by a transformation function $\rho(X, \mathcal{R})$, where X is a string variable and $\mathcal{R}$ is a regular expression. The function is defined as follows:

$$
\begin{array}{lll}
\rho(X, s) & ::= & X = s, \text{ where } s \text{ is a constant string} \\
\rho(X, \mathcal{R}_1|\mathcal{R}_2) & ::= & \rho(X, \mathcal{R}_1) \vee \rho(X, \mathcal{R}_2) \\
\rho(X, \mathcal{R}_1 \cdot \mathcal{R}_2) & ::= & X = T_1 \cdot T_2 \wedge \rho(T_1, \mathcal{R}_1) \wedge \rho(T_2, \mathcal{R}_2) \\
\rho(X, \mathcal{R}^*) & ::= & X = unroll(\mathcal{R}, n) \wedge n \geq 0
\end{array}
$$

where n is a fresh integer variable for each Kleene star operation; T_1 and T_2 are fresh string variables; $unroll(\mathcal{R}, n)$ represents the expression obtained by unrolling $\mathcal{R}$ n times. After the RE membership predicates are replaced by word equations, the string solver proceeds as usual. When the solver explores various arrangements, the $unroll()$ functions are further simplified by the following rules.

$$
\begin{array}{lll}
X = unroll(\mathcal{R}, n_1) & ::= & \textbf{if } (n_1 = 0) \textbf{ then } \{X = \epsilon\} \textbf{ else } \{X = T_3 \cdot unroll(\mathcal{R}, n_1 - 1) \wedge \rho(T_3, \mathcal{R})\} \\
X \cdot Y = unroll(\mathcal{R}, n_1) & ::= & \textbf{if } (n_1 = 0) \textbf{ then } \{X = \epsilon \wedge Y = \epsilon\} \textbf{ else } \{X = unroll(\mathcal{R}, n_2) \cdot T_3 \\
 & & \quad \wedge Y = T_4 \cdot unroll(\mathcal{R}, n_3) \wedge n_1 = n_2 + n_3 + 1 \wedge \rho(T_3 \cdot T_4, \mathcal{R})\}
\end{array}
$$

Note that $\mathcal{R}$ is essentially unrolled once in the else branch of both rules. Just like in other existing solvers that support RE, the unrolling process may not terminate, especially when there are no length constraints associated with the involved variables. We hence rely on a timeout mechanism.

While simple, elegant and efficient for typical equations obtained from program analysis, the word equation solver described here may fall into infinite loops under certain circumstances. This problem and our approach are described at length below. In fact, the problem of "overlapping" variables described below is recognized by logicians as the crucial source of complexity in solving word equations.

A Word Equation that Highlights the Crucial Overlap Detection Problem: To illustrate the problem, consider the example: $a \cdot X = X \cdot b$, where X is a string variable. X appears both as the LHS suffix and as the RHS prefix. This equation is not satisfiable. However, if we solve it analogously, then the solving procedure will not terminate. In particular, the equation has three arrangements: The first arrangement is where

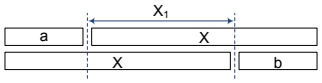


Fig. 2: Graphical representation of an arrangement of $a \cdot X = X \cdot b$, where the two occurrences of X overlap represented by X_1.

$X = \epsilon$, resulting in the equation $a = b$ which is unsatisfiable. The second arrangement is where the concatenation function in the LHS and RHS align exactly such that we get $X = a \wedge X = b$. The third arrangement is where the LHS occurrence of X cuts the RHS occurrence in the RHS illustrated in Figure 2.

Note that the suffix of X in the RHS (bottom part of Figure 2) of the equation overlaps with the prefix of X in the LHS (top part of Figure 2). We represent this overlapping part with a new variable X_1. By applying some simple rewrites we derive the following:

$$a \cdot X = X \cdot b \implies X = a \cdot {X_1}^{[2.1]} \wedge X = X_1 \cdot b^{[2.2]}$$

From $[2.1]$ and $[2.2]$, we can infer $a \cdot X_1 = X_1 \cdot b$. Note this derived equation has the same form as the input formula. As a result, the above-mentioned decision procedure will not terminate, unless some steps are taken to detect such "overlaps" and determine their satisfiability without computing arrangements ad infinitum.

One could imagine heuristics to detect and handle relatively simple overlaps described above. However, in general, when many equations are involved with variables overlapping indirectly, the problem is not easy to detect or decide. In fact, overlapping variables get to the crux of the difficulty of solving word equations, for otherwise simple rewrites can solve such equations. Hence, any solution to detecting overlaps is of universal value, and can be used as subroutine by many different types of string solvers.

4 New Techniques for Improving Efficiency of String Solvers

In this section, we present two search space pruning techniques to improve performance of word-based string solvers.

4.1 Subroutine for Detecting Overlapping Variables in Word Equations

Here we provide details about detecting overlapping variables in word equations.

Definition 1 (Boundary Labels, Generalized Word Equations, Label Sets). *We define boundary labels (aka labels) using special symbols $\triangleright_n^\star$ (left) and $\triangleleft_n^\star$ (right), where $\star$ is either a character c or a variable X, and n denotes its n-th occurrence in the equation. A left/right pair of labels on either side of a variable or character uniquely identifies the boundaries of that occurrence of that character or variable. A set of labels is simply called a label set. A word equation E annotated with label sets, where these sets replace every (implicit) occurrence of the concatenation function in the words of E, is called a generalized word equation.*

Below is an example of the word equation $a \cdot X = X \cdot b$ annotated with boundary labels. Note that the right label of the character "a" and the left label of the variable X are grouped into the set $\{\triangleleft_1^a, \triangleright_1^X\}$:

$$\{\triangleright_1^a\} a^{\{\triangleleft_1^a, \triangleright_1^X\}} X^{\{\triangleleft_1^X\}} \;=\; \{\triangleright_2^X\} X^{\{\triangleleft_2^X, \triangleright_1^b\}} b^{\{\triangleleft_1^b\}}$$

Definition 2. *[Label Arrangements and Merging of Label Sets] Given a word equation E, a label arrangement or simply arrangement A of E is an ordered sequence of label sets, where each label set in A is obtained by taking the union (aka "merging") of sequences of label sets from the LHS and RHS of E. We define the merge of two sequences $S_1 \equiv \{l_1, \cdots, l_k\}$ and $S_2 \equiv \{r_1, \cdots, r_k\}$ as some sequence S_3 whose elements are either simply elements of S_1 or S_2 or the union of some elements of S_1 and S_2.*

The intuition behind the construction of an arrangement A of given equation $l = r$ is very straightforward, namely, that we align the natural boundaries of the LHS l and RHS r by appropriately merging the label sets of the LHS and the RHS of E to obtain arrangements. The reason we construct arrangement is that it allows us to recursively derive simpler equations from a given equation E, until they are so simple that their satisfiability can be trivially determined.

By construction, a word equation can be equisatisfiably reduced to a finite disjunction of arrangements (The satisfiability of an arrangement is defined in terms of the word equations it implies). To better understand how arrangements are derived from an equation (or more precisely a generalized word equation) consider the following:

$$^{\{\triangleright_1^a\}}a^{\{\triangleleft_1^a, \triangleright_1^Y\}}Y^{\{\triangleleft_1^Y\}} = {}^{\{\triangleright_1^X\}}X^{\{\triangleleft_1^X, \triangleright_1^b\}}b^{\{\triangleleft_1^b\}}$$

A possible arrangement of its two words is the following:

$$\{\triangleright_1^a, \triangleright_1^X\} \cdot \{\triangleleft_1^a, \triangleleft_1^X, \triangleright_1^Y, \triangleright_1^b\} \cdot \{\triangleleft_1^Y, \triangleleft_1^b\}$$

From this arrangement, we can easily derive two smaller equations, $X = a$ and $Y = b$, which directly yield a solution. In our tech report (TR) [34] we describe, in much greater detail, several operations for label set manipulation and of "merging" label sets to obtain arrangements, and merging arrangements from multiple word equations. These operations are key for detecting complex overlapping variables that occur over multiple equations, and are not immediately obvious as is the case in $a \cdot X = X \cdot b$.

Detecting Overlapping Variables by Merging Arrangements: Just as we can construct arrangements by merging the ordered sequence of labels over words from a single equation, we can merge the arrangements obtained from multiple word equations, when these equations contain occurrences of the same variable. Intuitively, the arrangements from multiple equations may imply a variable being cut/split differently (e.g., X is cut by the boundary of Y in one equation and by the boundary of Z in another). Our algorithm explores all the possible orders of the cuts from different arrangements, denoted as label sets, for the same variable. Each order yields a *global arrangement* for the variable. As such, each variable is divided into a set of sub-variables guided by the global arrangement; the previous system of equations is hence reduced to a new system of equations with shorter and simpler words. More details can be found in our TR.

Detection of overlapping variable can be done by checking the following condition in any global arrangement: in the ordered sequence of label sets of a global arrangement, there exists a left label of an occurrence of a variable X that occurs in a label set in between two label sets where the first contains the left label and the second contains the right label of another occurrence of X. We say that X is an overlapping variable in the given system of word equations. As an example, consider the arrangement in Figure 2 that has overlapping variables. This arrangement written per the formal definition as a sequence of label sets we have:

$$\{\triangleright_1^a, \triangleright_2^X\} \cdot \{\triangleleft_1^a, \triangleright_1^X\} \cdot \{\triangleleft_2^X, \triangleright_1^b\} \cdot \{\triangleleft_1^X, \triangleleft_1^b\}$$

Theorem 1. *The subroutine for detecting overlapping variables is sound, complete and terminating, i.e., it correctly detects all overlapping variables and terminates.*

4.2 String and Integer Theory Integration

Basic Length Rules. For strings X and Y, we assert the following: (1) $|X| \geq 0$ (2) $|X| = 0 \leftrightarrow X = \epsilon$ (3) $X = Y \rightarrow |X| = |Y|$ (4) $|X \cdot ... \cdot Y| = |X| + ... + |Y|$.

String and Integer Theory Integration: As discussed in Section 2, finding a consistent solution for both strings and numbers can be expensive due to the infinite search space. The goal of string and integer theory integration is to achieve synergy from the two such that the procedure can converge faster. In particular, one theory will generate new assertions in the domain of the other theory, and vice versa. Inside the string theory, the set of arrangements that is explored is constrained by the assertions on string lengths, which are provided by the integer theory. On the other hand, the string theory will derive new length assertions when it makes progress in exploring new arrangements. These assertions are provided to the integer theory so that the search space is pruned.

Consider $X \cdot Y = M \cdot N$, where X, Y, M and N are nonempty string variables. It has three possible arrangements: $[a1]\ X = M \cdot T_1 \wedge N = T_1 \cdot Y$; $[a2]\ X = M \wedge N = Y$; $[a3]\ M = X \cdot T_2 \wedge Y = T_2 \cdot N$. Assume the integer theory infers that $|X| > |M|$ or $|Y| < |N|$. Thus, only $[a1]$ is consistent with the length conditions. The string solver only needs to explore one arrangement instead of three. On the other hand, assume the string solver is exploring arrangement $[a1]$. It generates a new assertion $[a1] \rightarrow |X| = |M \cdot T_1| \wedge |N| = |T_1 \cdot Y| \wedge |X| > |M| \wedge |N| > |Y|$, which in turn triggers the Z3 core to add an integer assertion.

Note that different string solvers implement string and integer integrations in vastly different ways[7,33,22,31]. [7] focuses on integration in a staged manner. [33] focuses on integration via automata manipulations. [22,31] and Z3str2 are integrations within the DPLL(T) architecture, where the algorithm only solves parts of the formula on demand and learns new constraints as it solves such that these implied constraints often cut the search dramatically. Compared to [22,31], our integration is tighter, powered by the bi-directional heuristics.

5 Soundness of the Z3str2 Algorithm

In this section we sketch the soundness proof of Z3str2's algorithm given in Algorithm 1. For a detailed formal analysis we refer the reader to the associated tech-report. The soundness property of any decision procedure in an SMT solving context can be stated as "If the procedure returns UNSAT, then input formula is indeed UNSAT".

Theorem 2. *Algorithm 1 is sound, i.e., when Algorithm 1 reports UNSAT, the input constraint is indeed UNSAT.*

Proof. To see that Z3str2 is sound, we show that the UNSAT returned at line 3 and line 24 are both sound. First observe that line 3 returns an UNSAT if either string or integer constraints are determined to be UNSAT. For string constraints, we use the algorithms described in [11] to decide the satisfiability of word (dis)equations in the solved form.

The soundness of line 3 relies on the soundness of the procedure [11] and the integer solver (here Z3).

For the UNSAT returned at line 24, we show transformations impacting it are all satisfiability-preserving. If a transformation is satisfiability-preserving, it means its output formula is satisfiable if and only if its input formula is satisfiable. In particular, transformations at (i) line 6 (ii) line 8 (iii) line 10 and (iv) lines 15-16 are satisfiability-preserving: (i) The disjunctive normal form conversion at line 6 is obviously satisfiability-preserving. (ii) The conversion in line 8 is probably the most involved in terms of establishing soundness. This step is a variant of the idea of sound transformation of word equations to arrangements mentioned in Makanin's paper [23]. We can show that arrangement generation is satisfiability-preserving because each arrangement is a finite set of equations implied by the input system of equations. In addition, we extract length constraints from arrangements and they may conflict with the existing integer constraints. If so, we drop inconsistent arrangements based on the UNSAT results determined by the integer theory. Similarly, since we assume the integer theory is sound, this step is also satisfiability-preserving. (iii) At line 10, we systematically enumerate all feasible orders among boundary labels according to the Definition 2. This step is satisfiability-preserving. (iv) In lines 15 and 16, this step derives simpler equations by a satisfiability-preserving rewriting. Please see the technical report for proof details. Note the REs are reduced to word equations so that they can be handled by this same procedure. In addition, although we prune arrangements at line 13, the answer can only be SAT or UNKNOWN once this happens. The algorithm is still sound. Therefore, we return UNSAT exactly when we can prove this to be the case.

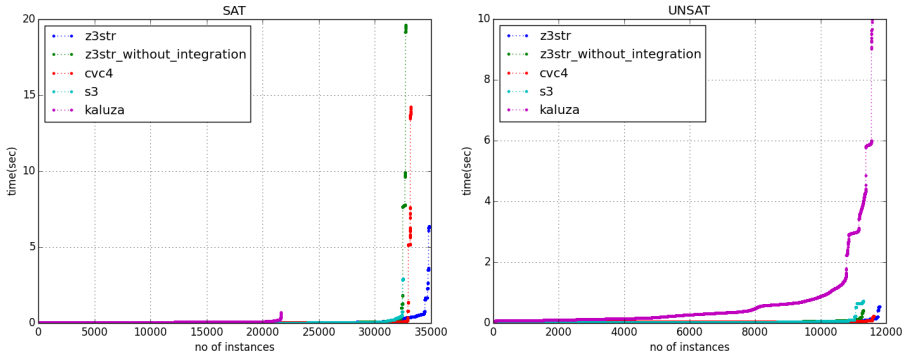
6 Experimental Results

In this section, we describe the implementation of Z3str2, as well as experiments to validate the efficacy of the new techniques proposed in this paper, namely, overlapping-variable detection and deeper string/integer theory integration. Both techniques improve solver efficiency in isolation, as well as when switched on simultaneously. However, in the interest of space we only report their combined contributions.

1. Detection of Overlapping Variables. During solving, Z3str2 prunes away arrangements with overlapping variables, leading to a smaller search space. Thus, if the technique is effective, we would be able to observe that other solvers time out on the cases reported as UNKNOWN by Z3str2. In Z3str2, an UNKNOWN result is returned when no SAT can be established in all arrangements with non-overlapping variables.

2. Evaluating String and Integer Theory Integration. The contribution of the string and integer integration will be illustrated by the improvement on the performance in resolving both the SAT and UNSAT cases, in comparison with other solvers.

We compare Z3str2 against five state-of-the-art string solvers, namely, CVC4[22], S3[31], Kaluza [28], PISA [30], and Stranger [32] across four different suites of benchmarks obtained from Kudzu/Kaluza[28], PISA[30], AppScan Source[2] and Kausler's [18] projects. Given the rich and diversified landscape of string problems, we chose to validate our approach using benchmarks from real-world applications with different characteristics. Additionally, the total number of tests on which we compared Z3str2 with other solvers is approximately 69,000.



(a) SAT cases (b) UNSAT cases

Fig. 3: Cactus plots for the Kaluza benchmark suite (incorrect results excluded)

Kaluza Benchmark suite. The Kaluza constraints were generated by a JavaScript symbolic execution engine [28], where length, concatenation and (finite) RE membership queries occur frequently. Both CVC4 and S3 were originally evaluated only on this suite, which consists of approximately 50K problems in the Kaluza format. The CVC4 team selected $47,284$ of them, and translated them into the CVC4 format. The S3 team did the translation to S3 format. We wrote translators from CVC4 to Z3str2, and from Z3str2 to CVC4. The timeout threshold for comparison over this suite was set at 20 seconds per problem, which was the threshold used in CVC4 [22].

PISA Benchmark suite. While the Kaluza suite is large and diverse, and includes string problems of varying sizes, it only contains a small subset of string operations. To make the comparison more comprehensive, we included constraints from real-world Java sanitizer methods that were used in the evaluation of the PISA system [30]. Sanitizers cleanse user input to remove the threat of an injection attack. They are usually complex and utilize various primitive string operations. We generated two groups of constraints: First, as in the PISA paper, we encode the semantics of the sanitizers and check the return value(s) against predefined attack patterns (such as cross-site scripting (XSS)). In the second group, we also encode input constraints per the application defining the sanitizer. For the PISA suite, we set a timeout value of 200 seconds due to its higher complexity.

AppScan Benchmark suite. The third suite of benchmarks is derived from security warnings output by IBM Security AppScan Source Edition [2], an application sold commercially by IBM. These reflect potentially vulnerable information flows, represented as traces of program statements, which yield more representative real-world constraints than focusing on sanitizers only. We ran AppScan on popular websites to obtain traces. Similar to the PISA benchmarks, the AppScan constraints also utilize a rich set of string operators. As with PISA, timeout here too was set at 200 seconds per benchmark.

Kausler Benchmark suite. The final suite is extracted from 8 Java programs by Scott Kausler [18]. They represent path conditions obtained from dynamic symbolic execution, and are pure string constraints [17]. Unlike other benchmarks, Kausler's suite does not dump string constraints to file but instead calls the solvers via an API. The suite contains 174 path condition encoding files, and the resulting constraints are input to

Table 1: Results on Kaluza suite[28].

	Z3str2		Z3str2 w/o integration		CVC4		S3		Kaluza	
	√	×	√	×	√	×	√	×	√	×
sat	34859	0	32752	0	33190	0	32503	488	21651	n/a
unsat	11799	0	11313	0	11625	0	11351	412	12099	10909
unknown	626†		395†		0		0		0	
timeout	0		2824		2469		989		340	
tool reports error	0		0		0		2		2285	
crash	0		0		0		1539		0	
Total time (sec)	4288.8(**1x**)		61232.8(**14.3x**)		52478.8 (**12.2x**)		22543.4 (**5.3x**)		46753.9 (**10.9x**)	

† 'unknown' indicates Z3str2 detected and avoided overlapping arrangements.

the solvers in-memory via their APIs. The comparison [18] was originally done using a driver interface [3]. However, we observed bugs ranging from JNI issues for Stranger to generating invalid constraints for Z3str2. We made our best attempt to compare Z3str2 with Stranger using modified interfaces [1] patched by both the Stranger team and us.

6.1 Performance results

The results we obtained are summarized in Tables 1 - 3 and Figure 3 with appropriate references to the various benchmark suites[2].

Kaluza suite. In Table 1, "tool reports error" counts the number of inputs on which the solver reports an error. "crash", instead, refers to runtime errors such as segfaults. For "sat" and "unsat", × denotes the number of provably incorrect results (either an "unsat" response where the problem has a verified solution or a "sat" response with an infeasible solution, as defined in [22]), and √ the rest. The comparison involves Z3str2 without bi-directional integration, CVC4 and S3, but not PISA, as PISA cannot model string lengths or symbolic arithmetic operations that are intensive in the suite.

According to Table 1, neither Z3str2 nor CVC4 report any provably incorrect result, though Z3str2 is more effective and can solve more cases (46658 compared to 44815) without timeouts. Though Z3str2 additionally has 626 unknown cases, CVC4 times out on all these cases. Z3str2 without bi-directional integration solves 2593 fewer cases and timeouts more often. S3 has errors in both directions, as well as an overall of 989 timeouts, while Kaluza suffers less from timeouts (340) but has many sat-as-unsat errors (10909). Kaluza therefore is unsound. Since Kaluza only provides assignments for variables matching the query, sat answers are not verifiable. Both S3 and Kaluza also have tool errors (2 and 2285, respectively). In addition, S3 crashes on 1539 cases. To compare performance on the `sat` and `unsat` Kaluza cases across the different solvers, we created the cactus plots in Figures 3a (`sat`) and 3b (`unsat`). Incorrect results are excluded. In both categories, Z3str2 and CVC4 have comparable performance, while Z3str2 solves more cases and is faster on complex cases. S3 and Z3str2 without string-integer integration are slower. Kaluza has the worst performance.

PISA suite. Table 2 presents the results on the PISA benchmarks. The "string operators stats" column lists the involved operations and their number of occurrences. In addition,

[2] All experiments were performed on a workstation running Ubuntu 12.04 with an i7-3770 CPU and 8GB of RAM memory. For reproducibility, we have made the Z3str2 source code publicly available [1]. We used V1.5-prerelease of CVC4; the version of S3 from the original paper [31]; the Kaluza version from the CVC4 paper with "`var`" as the query string; and Stranger from [4].

Table 2: Results on constraints generated from sanitizers detected by PISA [30].

input	string operators stats (omitting eq and dis-eq)	Z3str2				CVC4				PISA-MONA			
		var	pred	result	time (s)	var	pred	result	time (s)	var	pred	result	time (s)
pisa-000.smt2	contains (3), indexof (1), substring (1)	4	12	sat (√)	0.164	4	12	sat (√)	0.264	9	301	sat ($?^{+}$)	0.029
pisa-001.smt2	contains (1), indexof (1), substring (1)	4	9	sat (√)	0.114	4	9	sat (√)	0.032	—$^{+}$	—$^{+}$	—$^{+}$	—$^{+}$
pisa-002.smt2	contains (4)	2	10	sat (√)	0.114	2	10	sat (√)	50.871	—	—	—	—
pisa-003.smt2	contains (3), concat (1)	3	11	unsat (√)	0.064	3	11	timeout	200.00	—	—	—	—
pisa-004.smt2	contains (2), indexof (1), length (1), lastIndexof† (1), substring (2)	7	22	unsat (√)	0.038	10	32	timeout	200.00	9	331	unsat (√)	0.041
pisa-005.smt2	indexof (1), lastIndexof† (1), length (1), substring (2),	7	23	sat (√)	0.115	10	33	sat (√)	0.165	—	—	—	—
pisa-006.smt2	indexof (1), lastIndexof† (1), length (1), substring (2), contains (1)	7	24	unsat (√)	0.039	11	36	timeout	200.00	9	331	unsat (√)	0.038
pisa-007.smt2	indexof (2), lastIndexof† (1), length (1), substring (2), contains (1)	8	26	unsat (√)	0.042	11	36	timeout	200.00	9	324	unsat (√)	0.039
pisa-008.smt2	replace* (5), contains (2)	6	13	sat (√)	0.214	6	13	timeout	200.00	9	283	sat ($?^{+}$)	0.031
pisa-009.smt2	replace (2), concat (1), contains (2)	3	8	sat (√)	0.447	3	8	sat (√)	0.046	9	292	sat ($?^{+}$)	0.054
pisa-010.smt2	replace (2), concat (1)	3	6	sat (√)	0.165	3	6	timeout	200.00	—	—	—	—
pisa-011.smt2	replace (1), concat (2)	3	6	sat (√)	0.115	3	6	sat (√)	0.016	—	—	—	—

+ We could not generate constraints without changing PISA. No string solutions are generated so it's not verifiable.
† CVC4 doesn't provide operator 'lastIndexof'. We encode it with operators "concat", "length" and "contains".
* replace applies to the first occurrence of the argument string for both Z3str2 and CVC4.

Table 3: Results on constraints derived from AppScan traces [2].

input	string operators stats (omitting eq and dis-eq)	Z3str2				CVC4			
		var	pred	result	time (s)	var	pred	result	time (s)
t01.smt2	indexof (4), substring (3)	7	37	sat (√)	0.265	7	37	timeout	200.00
t02.smt2	concat (3), membership (1), regexConcat (2), regexUnion (14), str2Regex (17), length (1)	5	47	sat (√)	0.215	5	33	sat (√)	0.026
t03.smt2	concat (3), membership (1), regexConcat (2), regexUnion(14), str2Regex (17), length(1)	5	46	sat (√)	2.519	5	32	timeout	200.00
t04.smt2	concat (5), membership (1), regexConcat (2), regexUnion (14), str2Regex (17), length(1)	6	50	sat (√)	4.574	6	35	timeout	200.00
t05.smt2	concat (3), membership (1), regexConcat (2), indexof (1) regexUnion (14), str2Regex (17), length (2), substring (1)	8	56	sat (√)	2.770	8	42	timeout	200.00
t06.smt2	concat (1), indexof (3), endsWith (5)	5	33	sat (√)	0.214	5	33	sat (√)	3.021
t07.smt2	concat (6), regexStar (2), str2Regex (4), endsWith (2) regexUnion (2), membership (2), startsWith (2)	8	32	sat (√)	0.114	8	29	sat (√)	0.115
t08.smt2	concat (2), regexStar (2), str2Regex (4), endsWith (2) regexUnion (2), membership (2), startsWith (2)	5	23	sat (√)	0.164	5	22	sat (√)	151.663

we also count the number of variables and predicates for each format. In this comparison, we included CVC4 and PISA, but not S3 and Kaluza, as we were not able to model popular string operations such as indexof using their language. Besides, for PISA, while one group of constraints is equivalent to the MONA program generated by PISA, enabling proper comparison, the other group requires changes to the PISA translation algorithm (to fix the input constraints as well as the negative output constraints), and thus the respective comparisons were not possible. From Table 2, we have the following observations. First, Z3str2 reports 8 sat cases compared to 6 by CVC4 and 2 timeouts. For the 6 sats in common, Z3str2 solves them in $1.069s$ while CVC4 requires $51.394s$. Second, MONA and Z3str2 are in agreement. MONA runs faster on the sat cases, though it cannot generate satisfying string assignments, and has comparable performance to Z3str2 on the unsat cases.

AppScan suite. The results of the third comparison over the AppScan suite appear in Table 3. Z3str2 reports sat on 8 cases while CVC4 agrees on 4 and times out on the rest. The performance gap between the solvers on sat cases in agreement is significant: Z3str2 completes in $0.707s$, whereas the CVC4 solving time is $154.852s$.

Kausler suite. We were able to run the Stranger tool on 5 sets in this suite — namely, *beasties*, *jerichoHTMLParser*, *mathParser*, *mathQuizGame* and *naturalCLI* — without

crashing or hanging. Across these 5 sets, we found that the average solving time per constraint instance for Z3str2 are $6.4ms$, $10.7ms$, $39.9ms$, $7.1ms$ and $23.4ms$ respectively, and for Stranger are $51.8ms$, $5.9ms$, $1.4ms$, $9.4ms$ and $3.0ms$. Z3str2 is faster than Stranger on two of these sets, Stranger is faster than Z3str2 on the remaining three.

However, these findings should be qualified. First, Stranger crashes or hangs on 98 files. Z3str2 neither crashes nor hangs nor times out on any of the generated instances. We have omitted these 98 files from our comparison. Additionally, Stranger over-approximates disequalities ($\neq$ operator) among variables that can represent multiple strings [5]. We observe that such cases commonly exist in all sets (the percentages of instances with $\neq$ operators in each set are 83.4%, 61.7%, 79.0%, 96.0% and 95.0% respectively, and many fall into this category of disequalities among variables that represent multiple strings). This implies that Stranger produces unsound results. We believe that some of these constraints are easy for Stranger thanks to this over-approximation. By contrast Z3str2 correctly implements all operators and predicates in its input language. Finally, Stranger requires that integers occurring as indices and length bounds be constant, whereas Z3str2 and most other competing solvers support integers symbolically thus providing expressive power that is essential in practice.

6.2 Interpretation of Results

The general trend, across all benchmark suites, is that CVC4 has comparable performance to Z3str2 although CVC4 times out far more often than Z3str2, whereas S3 is significantly slower. These results establish the efficacy of both techniques presented in this paper.

Detection of overlapping variables. Z3str2 can decide either sat or unsat on 98.7%, 100% and 100% of the instances in the Kaluza, PISA and AppScan suites, respectively. CVC4, in comparison, achieves 94.8%, 50% and 50%. For unknowns reported by Z3str2 on the Kaluza instances, which occur in merely 1.3% of the cases, CVC4 times out on all of them. This lends support to our design choice of purposely pruning away parts of the solution space (those with overlapping arrangements) to avoid nontermination.

String and Integer Theory Integration. As the comparisons between Z3str2 versions with and without the integration clearly demonstrate, there is significant gain thanks to tightening the integer and string theory integration, which enables generation of implied constraints in both domains for more aggressive elimination of assignments unsatisfying for combined string-integer constraints.

7 Conclusion

We have described two techniques that dramatically improve the efficiency of word-based string solvers: (i) a sound and complete procedure to detect overlapping variables, thereby automatically identifying and avoiding sources of nontermination; and (ii) tight bi-directional integer/string theory integration, thereby pruning a vast array of inconsistent search candidates. We have implemented both of these techniques on top of Z3-str as Z3str2. We show the efficacy of these techniques through an extensive set of experiments, comparing Z3str2 with the CVC4, S3, Kaluza, PISA and Stranger solvers over four benchmark suites derived from real-world applications.

References

1. Z3str2 String Constraint Solver. https://sites.google.com/site/z3strsolver/.
2. IBM Security AppScan Source. http://www-03.ibm.com/software/products/en/appscan-source.
3. Kausler Suite. https://github.com/BoiseState/string-constraint-solvers.
4. LibStranger. https://github.com/vlab-cs-ucsb/LibStranger.
5. Personal communications with the Stranger team. 2015.
6. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 150–166, 2014.
7. Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 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 '09, pages 307–321, 2009.
8. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 1–18, 2003.
9. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, 2008.
10. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 519–531, 2007.
11. Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin Rinard. Word equations with length constraints: what's decidable? In *HVC'12*, 2012.
12. Indradeep Ghosh, Nastaran Shafiei, Guodong Li, and Wei-Fan Chiang. JST: An Automatic Test Generation Tool for Industrial Java Applications with Strings. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 992–1001, 2013.
13. Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 377–386, 2010.
14. J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to automata theory, languages, and computation*. Pearson/Addison Wesley, 2007.
15. Artur Jeż. Recompression: Word equations and beyond. In *Developments in Language Theory*, Lecture Notes in Computer Science, pages 12–26. 2013.
16. Juhani Karhumäki, Filippo Mignosi, and Wojciech Plandowski. The expressibility of languages and relations by word equations. *J. ACM*, 47(3):483–505, May 2000.
17. Scott Kausler. Evaluation of string constraint solvers using dynamic symbolic execution. Master's thesis, Boise State University, 2014.
18. Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 259–270, New York, NY, USA, 2014. ACM.
19. Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 105–116, 2009.
20. Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, 2014.

21. Guodong Li and Indradeep Ghosh. PASS: String solving with parameterized array and interval automaton. In *9th International Haifa Verification Conference*, HVC '13, pages 15–31. 2013.
22. Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A dpll(t) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV'14, pages 646–662. 2014.
23. G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English transl. in Math USSR Sbornik 32 (1977).
24. Yu. Matiyasevich. Word equations, fibonacci numbers, and hilbert's tenth problem. In *Workshop on Fibonacci Words*, 2007.
25. Wojciech Plandowski. Satisfiability of word equations with constants is in pspace. *J. ACM*, 51(3):483–496, May 2004.
26. Wojciech Plandowski. An efficient algorithm for solving word equations. In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing*, STOC '06, pages 467–476, 2006.
27. Gideon Redelinghuys, Willem Visser, and Jaco Geldenhuys. Symbolic execution of programs with strings. In *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, SAICSIT '12, pages 139–148, 2012.
28. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, 2010.
29. K. Schulz. Makanin's algorithm for word equations-two improvements and a generalization. In K. Schulz, editor, *Word Equations and Related Topics*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150. Springer Berlin / Heidelberg, 1992.
30. Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.*, 22(4):33:1–33:33, October 2013.
31. Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1232–1243, 2014.
32. Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: an automata-based string analysis tool for php. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 154–157, 2010.
33. Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 322–336, 2009.
34. Yunhui Zheng, Vijay Ganesh, Sanu Subramanian, Omer Tripp, Julian Dolby, and Xiangyu Zhang. Effective Search-space Pruning for Solvers of String Equations, Regular Expressions and Length Constraints, https://sites.google.com/site/z3strsolver/publications. Technical report, 2015.
35. Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, 2013.