Chain-Free String Constraints*

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Bui Phi Diep[1], Lukáš Holík[2], and Petr Janků[2]

[1] Uppsala University, Sweden
{parosh,mohamed_faouzi.atig,bui.phi-diep}@it.uu.se
[2] Brno University of Technology, Czech Republic
{holik,ijanku}@fit.vutbr.cz

Abstract. We address the satisfiability problem for string constraints that combine relational constraints represented by transducers, word equations, and string length constraints. This problem is undecidable in general. Therefore, we propose a new decidable fragment of string constraints, called weakly chaining string constraints, for which we show that the satisfiability problem is decidable. This fragment pushes the borders of decidability of string constraints by generalising the existing straight-line as well as the acyclic fragment of the string logic. We have developed a prototype implementation of our new decision procedure, and integrated it into in an existing framework that uses CEGAR with under-approximation of string constraints based on flattening. Our experimental results show the competitiveness and accuracy of the new framework.

Keywords: String Constraints · Satisfiability modulo theories · Program verification

1 Introduction

The recent years have seen many works dedicated to extensions of SMT solvers with new background theories that can lead to efficient analysis of programs with high-level data types. A data type that has attracted a lot of attention is *string* (for instance [10, 9, 17, 37, 38, 33, 18, 16, 4, 2, 20, 7, 14]). Strings are present in almost all programming and scripting languages. String solvers can be extremely useful in applications such as verification of string-manipulating programs [4] and analysis of security vulnerabilities of scripting languages (e.g., [29, 30, 37, 20]). The wide range of the commonly used primitives for manipulating strings in such languages requires string solvers to handle an expressive class of string logics. The most important features that a string solver have to model are *concatenation* (which is used to express assignments in programs), *transduction* (which can be used to model sanitisation and replacement operations), and *string length* (which is used to constraint lengths of strings).

It is well known that the satisfiability problem for the full class of string constraints with concatenation, transduction, and length constraints is undecidable in general [23,

* This work has been supported by the Czech Science Foundation (project No. 19-24397S), the IT4Innovations Excellence in Science (project No. LQ1602), and the FIT BUT internal projects FIT-S-17-4014 and FEKT/FIT-J-19-5906.

10] even for a simple formula of the form $\mathcal{T}(x,x)$ where $\mathcal{T}$ is a rational transducer and x is a string variable. However, this theoretical barrier did not prevent the development of numerous efficient solvers such as Z3-str3 [7], Z3-str2 [38], CVC4 [18], S3P [33, 34], and TRAU [2, 3]. These tools implement semi-algorithms to handle a large variety of string constraints, but do not provide completeness guarantees. Another direction of research is to find meaningful and expressive subclasses of string logics for which the satisfiability problem is decidable. Such classes include the acyclic fragment of Norn [5], the solved form fragment [13], and also the straight-line fragment [20, 14, 9].

In this paper, we propose an approach which is a mixture of the two above research directions, namely finding decidable fragments and making use of it to develop efficient semi-algorithms. To that aim, we define the class of *chain-free* formulas which strictly subsumes the acyclic fragment of Norn [5] as well as the straight-line fragment of [20, 14, 9], and thus extends the known border of decidability for string constraints. The extension is of a practical relevance. A straight-line constraint models a path through a string program in the single static assignment form, but as soon as the program compares two initialised string variables, the string constraint falls out of the fragment. The acyclic restriction of Norn on the other hand does not include transducer constraints and does not allow multiple occurrences of a variable in a single string constraint (e.g. an equation of the form $xy = zz$). Our chain-free fragment is liberal enough to accommodate constraints that share both these forbidden features (including $xy = zz$).

The following pseudo-PHP code (a variation of a code at [35]) that prompts a user to change his password is an example of a program that generates a chain-free constraint that is neither straight-line nor acyclic according to [20, 4].

```
$old=$database->real_escape_string($oldIn);
$new=$database->real_escape_string($newIn);
$pass=$database->query("SELECT password FROM users WHERE userID=".$user);
if($pass == $old)
   if($new != $old)
      $query = "UPDATE users SET password=".$new." WHERE userID=".$user;
      $database->query($query);
```

The user inputs the old password `oldIn` and the new password `newIn`, both are sanitized and assigned to `old` and `new`, respectively. The old sanitized password is compared with the value `pass` from the database, to authenticate the user, and then also with the new sanitized password, to ensure that a different password was chosen, and finally saved in the database. The sanitization is present to prevent SQL injection. To ensure that the sanitization works, we wish to verify that the SQL query `query` is safe, that is, it does not belong to a regular language *Bad* of dangerous inputs. This safety condition is expressed by the constraint

$$\texttt{new} = \mathcal{T}(\texttt{newIn}) \wedge \texttt{old} = \mathcal{T}(\texttt{oldIn}) \wedge \texttt{pass} = \texttt{old} \wedge \texttt{new} \neq \texttt{old} \\ \wedge \texttt{query} = u.\texttt{new}.v.\texttt{user} \wedge \texttt{query} \in \mathit{Bad}$$

The sanitization on lines 1 and 2 is modeled by the transducer $\mathcal{T}$, and u and v are the constant strings from line 7. The constraints fall out from the straight-line due to the test $\texttt{new} \neq \texttt{old}$. The main idea behind the chain-free fragment is to associate to the set of

relational constraints a *splitting graph* where each node corresponds to an occurrence of a variable in the relational constraints of the formula (as shown in Figure 1). An edge from an occurrence of x to an occurrence of y means that the source occurrence of x appears in a relational constraint which has in the opposite side an occurrence of y different from the target occurrence of y. The chain-free fragment prohibits loops in the graph, that we call *chains*, such as those shown in red in Figure 1.

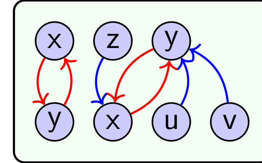

Fig. 1: The splitting graph of $x = z \cdot y \wedge y = x \cdot u \cdot v$.

Then, we identify the so called *weakly chaining* fragment which strictly extends the chain-free fragment by allowing *benign* chains. Benign chains relate relational constraints where each left side contains only one variable, the constraints are all *length preserving*, and all the nodes of the cycles appear exclusively on the left or exclusively on the right sides of the involved relational constraints (as is the case in Figure 1). Weakly chaining constraints may in practice arise from the checking that an encoding followed a decoding function is indeed the identity, i.e., satisfiability of constraints of the form $\mathcal{T}_{\text{enc}}(\mathcal{T}_{\text{dec}}(x)) = x$, discussed e.g. in [15]. For instance, in situations similar to the example above, one might like to verify that the sanitization of a password followed by the application of a function supposed to invert the sanitization gives the original password.

Our decision procedure for the weakly chaining formulas proceeds in several steps. The formula is transformed to an equisatisfiable chain-free formula, and then to an equisatisfiable concatenation free formula in which the relational constraints are of the form $\mathcal{T}(x,y)$ where x and y are two string variables and $\mathcal{T}$ is a transducer/relational constraint. Finally, we provide a decision procedure of a chain and concatenation-free formulae. The algorithm is based on two techniques. First, we show that the chain-free conjunction over relational constraints can be turned into a single equivalent transducer constraint (in a similar manner as in [6]). Second, consistency of the resulting transducer constraint with the input length constraints is checked via the computation of the Parikh image of the transducer.

To demonstrate the usefulness of our approach, we have implemented our decision procedure in SLOTH [14], and then integrated it in the open-source solver TRAU [2, 3]. TRAU is a string solver which is based on a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. These two modules interact together in order to automatically make these approximations more precise. We have implemented our decision procedure inside the over-approximation module which takes as an input a constraint and checks if it belongs to the weakly chaining fragment. If it is the case, then we use our decision procedure outlined above. Otherwise, we start by choosing a minimal set of occurrences of variables x that needs to be replaced by fresh ones such that the resulting constraint falls in our decidable fragment. We compare our prototype implementation against four other state-of-the-art string solvers, namely Ostrich [10], Z3-str3 [7], CVC4 [18, 19], and TRAU [1]. For our comparison with Z3-str3, we use the version that is part of Z3 4.8.4. Our experimental results show the competitiveness as as well as accuracy of the framework compared to the solver TRAU [2, 3]. Furthermore, the experimental results

show the competitiveness and generality of our method compared to the existing techniques. In summary, our main contributions are: (1) a new decidable fragment of string constraints, called chain-free, which strictly generalises the existing straight-line as well as the acyclic fragment [20, 4] and precisely characterises the decidability limitations of general relational/transducer constraints combined with concatenation, (2) a relaxation of the chain-free fragment, called weakly chaining, which allows special chains with length preserving relational constraints, (3) a decision procedures for checking the satisfiability problem of chain-free as well as weakly chaining string constraints, and (4) a prototype with experimental results that demonstrate the efficiency and generality of our technique on benchmarks from the literature as well as on new benchmarks.

2 Preliminaries

Sets and strings. We use $\mathbb{N}$, $\mathbb{Z}$ to denote the sets of natural numbers and integers, respectively. A finite set Σ of *letters* is an *alphabet*, a sequence of symbols $a_1 \cdots a_n$ from Σ is a *word* or a *string* over Σ, with its *length* n denoted by $|w|$, ε is the *empty word* with $|\varepsilon| = 0$, it is a neutral element with respect to string concatenation $\circ$, and Σ^* is the set of all words over Σ including ε.

Logic. Given a predicate formula, an occurrence of a predicate is *positive* if it is under an even number of negations. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of *clauses* that are themselves conjunctions of (negated) predicates. We write $\Psi[x/t]$ to denote the formula obtained by substituting in the formula Ψ each occurrence of the variable x by the term t.

(Multi-tape)-Automata and transducers. A *Finite Automaton* (FA) over an alphabet Σ is a tuple $\mathcal{A} = \langle Q, \Delta, I, F \rangle$, where Q is a finite set of *states*, $\Delta \subseteq Q \times \Sigma_\varepsilon \times Q$ with $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ is a set of *transitions*, and $I \subseteq Q$ (resp. $F \subseteq Q$) are the *initial* (resp. *accepting*) states. $\mathcal{A}$ accepts a word w iff there is a sequence $q_0 a_1 q_1 a_2 \cdots a_n q_n$ such that $(q_{i-1}, a_i, q_i) \in \Delta$ for all $1 \leq i \leq n$, $q_0 \in I$, $q_n \in F$, and $w = a_1 \circ \cdots \circ a_n$. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set all accepted words.

Given $n \in \mathbb{N}$, a *n-tape automaton* $\mathcal{T}$ is an automaton over the alphabet $(\Sigma_\varepsilon)^n$. It *recognizes* the relation $\mathcal{R}(\mathcal{T}) \subseteq (\Sigma^*)^n$ that contains vectors of words $(w_1, w_2, \ldots, w_n)$ for which there is $(a_{(1,1)}, a_{(2,1)}, \ldots, a_{(n,1)}) \cdots (a_{(1,m)}, a_{(2,m)}, \ldots, a_{(n,m)}) \in \mathcal{L}(\mathcal{T})$ with $w_i = a_{(i,1)} \circ \cdots \circ a_{(i,m)}$ for all $i \in \{1, \ldots, n\}$. A n-tape automaton $\mathcal{T}$ is said to be *length-preserving* if its transition relation $\Delta \subseteq Q \times \Sigma^n \times Q$. A *transducer* is a 2-tape automaton.

Let us recall some well-know facts about the class of multi-tape automata. First, the class of n-tape automata is closed under union but not under complementation nor intersection. However, the class of *length-preserving* multi-tape automata is closed under intersection. Multi-tape automata are closed under composition. Let $\mathcal{T}$ and $\mathcal{T}'$ be two multi-tape automata of dimension n and m, respectively, and let $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$ be two indices. Then, it is possible to construct a $(n+m-1)$-tape automaton $\mathcal{T} \wedge_{(i,j)} \mathcal{T}'$ which accepts the set of words $(w_1, \ldots, w_n, u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_m)$ if and only if $(w_1, \ldots, w_n) \in \mathcal{R}(\mathcal{T})$ and $(u_1, \ldots, u_{j-1}, w_i, u_{j+1}, \ldots, u_m) \in \mathcal{R}(\mathcal{T}')$. Furthermore, we can show that multi-tape automata are closed under permutations: Given a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ and a n-tape automaton $\mathcal{T}$, it is possible to construct

a n-tape automaton $\sigma(\mathcal{T})$ such that $\mathcal{R}(\sigma(\mathcal{T})) = \{(w_{\sigma(1)},\ldots,w_{\sigma(n)}) \mid (w_1,w_2,\ldots,w_n) \in \mathcal{R}(\mathcal{T})\}$. Finally, given a n-tape automaton $\mathcal{T}$ and a natural number $k \geq n$, we can construct a k-tape automaton s. t. $(w_1,\ldots,w_k) \in \mathcal{R}(\mathcal{T}')$ if and only if $(w_1,\ldots,w_n) \in \mathcal{R}(\mathcal{T})$.

3 String Constraints

The syntax of a string formula Ψ over an alphabet Σ and a set of variables $\mathbb{X}$ is given in Figure 2. It is a Boolean combination of memberships, relational, and arithmetic constraints over string terms t_{str} (i.e., concatenations of variables in $\mathbb{X}$). *Membership constraints* denote membership in the language of a finite-state automaton $\mathcal{A}$ over Σ. *Relational constraints* denote either an equality of string terms, which we normally write as $t = t'$ instead of $=(t,t')$, or that the terms are related by a relation recognised by a transducer $\mathcal{T}$. (Observe that the equality relations can be also expressed using length preserving transducers.) Finally, arithmetic terms t_{ar} are linear functions over term lengths and integers, and arithmetic constraints are inequalities of arithmetic terms. String formulae allow using negation with one restriction, namely, constraints that are *not invertible* must have only positive occurrences. General transducers are not invertible, it is not possible to negate them. Regular membership, length preserving relations (including equality), and length constraints are invertible.

$$
\begin{aligned}
\Psi &::= \varphi \mid \Psi \wedge \Psi \mid \Psi \vee \Psi \mid \neg\Psi \\
\varphi &::= \mathcal{A}(t_{str}) \mid R(t_{str},t_{str}) \mid t_{ar} \geq t_{ar} \\
R &::= \mathcal{T} \mid = \\
t_{str} &::= \varepsilon \mid x \mid t_{str} \circ t_{str} \\
t_{ar} &::= k \mid |t_{str}| \mid t_{ar} + t_{ar}
\end{aligned}
$$

Fig. 2: Syntax of string formulae

To simplify presentation, we do not consider *mixed* string terms t_{str} that contain, besides variables of $\mathbb{X}$, also symbols of Σ. This is without loss of generality because a mixed term can be encoded as a conjunction of the pure term over $\mathbb{X}$ obtained by replacing every occurrence of a letter $a \in \Sigma$ by a fresh variable x and the regular membership constraints $\mathcal{A}_a(x)$ with $\mathcal{L}(\mathcal{A}_a) = \{a\}$. Observe also that membership and equality constraints may be expressed using transducers.

Semantics. We describe the semantics of our logic using a mapping η, called *interpretation*, that assigns to each string variable in $\mathbb{X}$ a word in Σ^*. Extended to string terms by $\eta(t_{s_1} \circ t_{s_2}) = \eta(t_{s_1}) \circ \eta(t_{s_2})$. Extended to arithmetic terms by $\eta(|t_s|) = |\eta(t_s)|$, $\eta(k) = k$ and $\eta(t_i + t_i') = \eta(t_i) + \eta(t_i')$. Extended to atomic constraints, η returns a truth value:

$$
\begin{aligned}
\eta(\mathcal{A}(t_{str})) = \top &\quad \text{iff} \quad \eta(t_{str}) \in \mathcal{L}(\mathcal{A}) \\
\eta(R(t_{str},t'_{str})) = \top &\quad \text{iff} \quad (\eta(t_{str}),\eta(t'_{str})) \in \mathcal{R}(R) \\
\eta(t_{i_1} \leq t_{i_2}) = \top &\quad \text{iff} \quad \eta(t_{i_1}) \leq \eta(t_{i_2})
\end{aligned}
$$

Given two interpretations η_1 and η_2 over two disjoint sets of string variables $\mathbb{X}_1$ and $\mathbb{X}_2$, respectively. We use $\eta_1 \cup \eta_2$ to denote the interpretation over $\mathbb{X}_1 \cup \mathbb{X}_2$ such that $(\eta_1 \cup \eta_2)(x) = \eta_1(x)$ if $x \in \mathbb{X}_1$ and $(\eta_1 \cup \eta_2)(x) = \eta_2(x)$ if $x \in \mathbb{X}_2$.

The truth value of a Boolean combination of formulae under η is defined as usual. If $\eta(\Psi) = \top$ then η is a *solution* of Ψ, written $\eta \models \Psi$. The formula Ψ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

A relational constraint is said to be *left-sided* if and only if it is on the form $R(x,t_{str})$ where $x \in \mathbb{X}$ is a string variable and t_{str} is a string term. Any string formula can be transformed into a formula where all the relational constraints are left-sided by replacing any relational constraint of the form $R(t_{str},t'_{str})$ by $R(x,t'_{str}) \wedge x = t$ where x is fresh.

A formula Ψ is said to be *concatenation free* if and only if for every relational constraint $R(t_{str},t'_{str})$, the string terms t_{str} and t'_{str} appearing in the parameters of any relational constraints in Ψ are variables (i.e., $t_{str},t'_{str} \in \mathbb{X}$).

4 Chain Free and Weakly Chaining Fragment

It is well known that the satisfiability problem for the class of string constraint formulas is undecidable in general [23, 10]. This problem is undecidable already for a single transducer constraint of the form $\mathcal{T}(x,x)$ (by a simple reduction from the Post-Correspondence Problem). In the following, we define a subclass called *weakly chaining fragment* for which we prove that the satisfiability problem is decidable.

Splitting graph. Let $\Psi ::= \bigwedge_{j=1}^{m} \varphi_j$ be a conjunction of relational string constraints with $\varphi_j ::= R_j(t_{2j-1},t_{2j}), 1 \leq j \leq m$ where for each $i : 1 \leq i \leq 2m$, t_i is a concatenation of variables $x_i^1 \circ \cdots \circ x_i^{n_i}$. We define the set of *positions* of Ψ as $P = \{(i,j) \mid 1 \leq j \leq 2m \ \wedge \ 1 \leq i \leq n_j\}$. The *splitting graph* of Ψ is then the graph $G_\Psi = (P, E, \mathtt{var}, \mathtt{con})$ where the positions in P are its nodes, and the mapping $\mathtt{var} : P \to \mathbb{X}$ labels each position (i,j) with the variable x_j^i appearing at that position. We say that $(i,2j-1)$ (resp. $(i,2j)$) is the ith *left* (resp. *right*) positions of the jth constraint, and that R_j is the predicate of these positions. Any pair of a left and a right position of the same constraint are called *opposing*. The set of edges E then consists of edges (p,p') between positions for which there is an intermediate position p'' (different from p') that is opposing to p and is labeled by the same variable as p' ($\mathtt{var}(p'') = \mathtt{var}(p')$). Finally, the labelling $\mathtt{con}$ of edges assigns to (p,p') the constraint of p, that is, $\mathtt{con}(p,p') = R_j$ where p is a position of the jth constraint. An example of a splitting graph is on Fig. 1.

Chains. A *chain*[3] in the graph is a sequence of the form $(p_0,p_1),(p_1,p_2),\ldots,(p_n,p_0)$ of edges in E. A chain is *benign* if (1) all the relational constraints corresponding to the edges $\mathtt{con}(p_0,p_1),\mathtt{con}(p_1,p_2),\ldots,\mathtt{con}(p_n,p_0)$ are left sided and and all the string relations involved in these constraints are length preserving, and (2) the sequence of positions $p_0,p_1,\ldots,p_n$ consists of left positions only, or from right positions only. Observe that if there is a benign chain that uses only right positions then there exists also a benign chain that uses only left positions. The graph is *chain-free* if it has no chains, and it is *weakly chaining* if all its chains are benign. A formula is *chain-free* (resp. *weakly chaining*) if the splitting graph of every clause in its DNF is chain-free (resp. weakly chaining). Benign chains are on Fig. 1 shown in red.

In the following sections, we will show decision procedures for the chain-free and weakly chaining fragments. Particularly, we will show how a weakly chaining formula can be transformed to a chain-free formula by elimination of benign cycles, how then

[3] We use chains instead of cycles in order to avoid confusion between our decidable fragment and the ones that exist in literatures.

concatenation can be eliminated from a chain-free formula, and finally how to decide a concatenation free-formula.

Undecidability of Chaining Formulae. Before presenting the decision procedures for weakly chaining formulae, we finish the current section by stating that the chain-free fragment is indeed the limit of decidability of general transducer constraints, in the following sense: We say that two conjunctive string formulae have the same *relation-concatenation skeleton* if one can be obtained from the other by removing membership and length constraints and replacing a constraint of the form $R(t,t')$ by another constraint of the form $R'(t,t')$. A *skeleton class* is then an equivalence class of string formulae that have the same relation-concatenation skeleton.

Lemma 1. *The satisfiability problem is undecidable for every given* skeleton class*.*

The proof of the above lemma can be done through a reduction from undecidability of general transducer constraints of the form $\mathcal{T}(x,x)$. Together with decidability of chain-free formulae, discussed in Sections 6 and 7, the lemma implies that the satisfiability problem for a skeleton class is decidable *if and only if* its splitting graph is chain-free. In other words, chain-freeness is the most precise criterion of decidability of string formulae based on relation-concatenation skeletons (that is, a criterion independent of the particular values of relational, membership, and length constraints).

5 Weakly Chaining to Chain-Free

In the following, we show that, given a weakly chaining formula, we can transform it to an equisatisfiable chain-free formula.

Theorem 1. *A weakly chaining formula can be transformed to an equisatisfiable chain-free formula.*

The rest of this section is devoted to the proof of Theorem 1 (which also provides an algorithm how to transform any weakly chaining formula into an equisatisfiable chain-free formula). In the following, we assume w.l.o.g. that the given weakly-chaining formula Ψ is conjunctive. The proof is done by induction on the number $\mathbf{B}$ of relational constraints that are labelling the set of benign chains in the splitting graph of Ψ.

Base Case ($\mathbf{B=0}$*).* Since there is no benign chain in G_Ψ, Theorem 1 holds.

Induction Case ($\mathbf{B>0}$*).* In the following, we will show how to remove one benign chain (and its set of labelling relational constraints) in the case where the splitting graph of Ψ does not contain nested chains. If nested chains are present, then the proof follows the same main ideas, but the reasoning is generalised from one benign chain to strongly connected components. Let $\rho = (p_0,p_1),(p_1,p_2),\ldots,(p_n,p_0)$ be a benign chain in the splitting graph G_Ψ. For every $i \in \{0,\ldots,n\}$, let $R_i(x_i,t_i)$ be the length preserving relational constraint to which the position p_i belongs. We assume w.l.o.g.[4] that all the positions $p_0,p_1,\ldots,p_n$ are left positions. Since ρ is a benign chain, we have that the

[4] This is possible since if there is benign chain that uses only right positions then there exists also a benign chain that uses only left positions.

variable x_i is appearing in the string term $t_{(i+n)mod(n+1)}$ for all $i \in \{0,\ldots,n\}$. Furthermore, we can use the fact that the relational constraints are length preserving to deduce that the variables $x_0,x_1,\ldots,x_n$ have the same length. This implies also that, for every $i \in \{0,1,\ldots,n\}$, the string term t'_i that is constructed by removing from t_i one occurrence of $x_{(i+1)mod(n+1)}$ is equivalent to the empty word. Therefore, the relational constraint $R_i(x_i,t_i)$ can be rewritten as $R_i(x_i,x_{(i+1)mod(n+1)})$ for all $i \in \{0,1,\ldots,n\}$.

Let $x_{i_1},x_{i_2},\ldots,x_{i_k}$ be the maximal subsequence of pairwise distinct variables in $x_0,x_1,\ldots,x_n$. Let `index` be a mapping that associates to each index $\ell \in \{0,\ldots,n\}$ the index $j \in \{1,\ldots,k\}$ such that $x_\ell = x_{i_j}$. We can transform the transducer R_i, with $i \in \{0,\ldots,n\}$, to a length preserving k-tape automaton A_i such that a word $(w_1,w_2,\ldots,w_k)$ is accepted by A_i if and only if $(w_{\texttt{index}(i)},w_{\texttt{index}((i+1)mod(n+1))})$ is accepted by R_i. Let then A be the k-tape automaton resulting from the intersection of $A_0,\ldots,A_n$. Observe that A is also length-preserving. Furthermore, we have that $(w_1,w_2,\ldots,w_k)$ is accepted by A if and only if $(w_{\texttt{index}(i)},w_{\texttt{index}((i+1)mod(n+1))})$ is accepted by R_i for all $i \in \{0,\ldots,n\}$ (i.e., the automaton A characterizes all possible solutions of $\bigwedge_{i=0}^n R_i(x_i,x_{(i+1)mod(n+1)})$). Ideally, we would like to replace the $\bigwedge_{i=0}^n R_i(x_i,x_{(i+n)mod(n+1)})$ by $A(x_{i_1},x_{i_2},\ldots,x_{i_k})$, however, our syntax forbids such k-ary relation. To overcome this problem, we first extend our alphabet Σ by all the letters in Σ^k and then we replace $\bigwedge_{i=0}^n R_i(x_i,x_{(i+1)mod(n+1)})$ by $\varphi := A(x) \wedge \bigwedge_{j=1}^k \pi_j(x_{i_j},x)$ where x is a fresh variable and for every $j \in \{1,\ldots,k\}$, π_j is the length preserving transducer that accepts all pairs of the form $(w_j,(w_1,w_2,\ldots,w_k))$. Finally, let Ψ' be the formula obtained from Ψ by replacing the subformula $\bigwedge_{i=0}^n R_i(x_i,t_i)$ in Ψ by $\varphi \wedge |t'_i| = 0$ (remember that the string term t'_i is t_i from which we have removed one occurrence of the variable $x_{(i+1)mod(n+1)}$). It is easy to see that Ψ' is satisfiable iff Ψ is also satisfiable. Furthermore, the number of relational constraints that are labelling the set of benign chains in the splitting graph of Ψ' is strictly less than $\mathbf{B}$ (since π_j can not be used to label any benign chain in Ψ').

6 Chain-Free to Concatenation Free

In the following, we show that we can reduce the satisfiability problem for a chain free formula to the satisfiability problem of a concatenation free formula. To that aim, we describe an algorithm that eliminates concatenation from relational constraints by iterating simple splitting steps. When it terminates, it returns a formula over constraints that are concatenation free. The algorithm can be applied if the string constraints in the formula *allow splitting*, and it is guaranteed to terminate if the formula is *chain-free*. We will explain these two notions below together with the description of the algorithm.

Splitting. The *split* of a relational constraint $\varphi ::= R(x \circ t, y \circ t')$ with $t,t' \neq \varepsilon$ is the formula $\Phi_L \vee \Phi_R$ where

$$\Phi_L ::= \bigvee_{i=1}^n R_i(x_1,y) \wedge R'_i(x_2 \circ t,t')\ [x/x_1 \circ x_2]$$
$$\Phi_R ::= \bigvee_{j=1}^m R_j(x,y_1) \wedge R'_j(t,y_2 \circ t')\ [y/y_1 \circ y_2]$$

$m,n \in \mathbb{N}$, x_1,x_2,y_1,y_2 are fresh variables, and $\eta \models \varphi$ if and only if there is an assignment $\eta' : \{x_1,x_2,y_1,y_2\} \to \Sigma^*$ such that $\eta \cup \eta' \models (\Phi_L \wedge x = x_1 \circ x_2) \vee (\Phi_R \wedge y = y_1 \circ y_2)$. The

formula Φ_L is called the *left split* and Φ_R is called the *right split* of φ. In case $t' = \varepsilon$, the split is defined in the same way but with Φ_L left out, and if $t = \varepsilon$, then Φ_R is left out. If both t and t' equal ε, then φ is concatenation free and does not have a split. A simple example is the equation $xy = zz$ with the split $(x_1 = z \wedge x_2 y = z) \vee (x = z_1 \wedge y = z_2 z_1 z_2)$. A class of relational constraints $\mathcal{C}$ *allows splitting* if for every constraint in $\mathcal{C}$ that is not concatenation free, it is possible to compute a split that belongs to $\mathcal{C}$. Equalities as well as transducer constraints allow splitting. A left split of an equality $x \circ t = y \circ t'$ is $x_1 = y \wedge x_2 \circ t = t'$. A left split of a transducer constraint $\mathcal{T}(x \circ t, y \circ t')$ is the formula

$$\bigvee_{q \in Q} \mathcal{T}_q(x_1, y) \wedge {}_q\mathcal{T}(x_2 \circ t, t')$$

where Q is the set of states of $\mathcal{T}$, and ${}_q\mathcal{T}$ and $\mathcal{T}_q$ are the $\mathcal{T}$ with the original set of initial and final states, respectively, replaced by $\{q\}$ (this is the automata splitting technique of [4] extended to transducers in [20]). The right splits are analogous.

Splitting algorithm. A *splitting algorithm* for eliminating concatenation iterates *splitting steps* on a formula in DNF. A *splitting step* can be applied to one of the clauses if it can be written in the form $\Upsilon ::= \varphi \wedge \Psi$ where $\varphi ::= R(x \circ t, y \circ t')$. It then replaces the clause by a DNF of the disjunction

$$(\Phi_L \wedge \Psi[x/x_1 \circ x_2] \wedge |x| = |x_1| + |x_2|) \vee (\Phi_R \wedge \Psi[y/y_1 \circ y_2] \wedge |y| = |y_1| + |y_2|)$$

where Φ_L and Φ_R are the left and the right split, respectively, of φ. The left or the right disjunct is omitted if $t' = \varepsilon$ or $t = \varepsilon$, respectively. The splitting step is not applied when both t and t' equal ε, i.e. φ is concatenation free. In order to ensure termination, the algorithm applies splitting steps under the following regimen consisting of two phases.

In Phase 1, the algorithm maintains each clause Υ of a DNF of the string formula annotated with a *reminder*, a sub-graph H_Υ of its splitting graph G_Υ. The reminders restrict the choice of splitting steps so that a splitting step can be applied to a clause $\Upsilon = \varphi \wedge \Psi$ only if φ is a *root constraint* in H_Υ, meaning that all positions at one of the sides of φ are root nodes of H_Υ. The reminder graphs are assigned to clauses as follows. The algorithm is initialised with $H_\Upsilon ::= G_\Upsilon$ for each clause Υ. After taking a splitting step, the reminder graph of each new clause Υ' is a sub-graph $H_{\Upsilon'}$ of its splitting graph $G_{\Upsilon'}$. Particularly, $H_{\Upsilon'}$ contains only those constraints of Υ' (their positions that is) that are non-concatenation-free successors of the constraints of Υ that appear in H_Υ. The newly created concatenation free constraints do not propagate to $H_{\Upsilon'}$. Here, by saying that a constraint φ' of Υ' is a *successor* of a constraint φ of Υ means that either $\varphi' = \varphi[x/x_1 \circ x_2]$ or $\varphi' = \varphi[y/y_1 \circ y_2]$, or that they are the constraints explicitly mentioned in the definition of left or right split. Phase 1 terminates when the reminder graphs of all clauses are empty.

Phase 2 then performs splitting steps in any order until all constraints are concatenation free.

Theorem 2. *When run on a chain-free formula, the splitting algorithm terminates with an equisatisfiable chain and concatenation-free formula.*

Hereafter, we provide a brief overview of the proof of Theorem 2. The main difficulty with proving termination of splitting is the substitution of variables involved in

the left and right split. The left split makes a step towards concatenation freeness by removing one concatenation operator $\circ$ from the clause, since the terms $x \circ t$ and $y \circ t'$ are replaced by x_1, y, t', and $x_2 \circ t$. However, the substitution of x by $x_1 \circ x_2$ in the reminder of the clause introduces as many new concatenations as there are occurrences of x other than the one explicit in the definition of the left split (and similarly for the right split). Therefore, to guarantee termination of splitting, we must limit the effect of substitution by enforcing chain-freeness.

Why chain-freeness is the right property here may be intuitively explained as follows. The splitting graph of a clause is in fact a map of how chains of substitutions may increase the number of concatenations in the clause. Consider an edge in the splitting graph from a position p to a position p'. By definition, there is an intermediate position p'' opposite p and carrying the same variable as p'. This means that when splitting decreases the number of concatenations on the side of p by one (the label of p may be y referred to in the left split), the substitution of the label of p'' (this would be x in the left split) would cause that the position p' also labeled by x is replaced by the concatenation $x_1 \circ x_2$. Moreover, since the length of the side of p' is now larger, it is possible to perform more splitting steps that follow edges starting at the side of p' and increase numbers of $\circ$ at positions reachable from p' and consequently also further along the path in the splitting graph starting at (p, p'). Hence the intuitive meaning of the edge is that decreasing the number of $\circ$ at the side of p might increase the number of $\circ$ at the side of p'. Chain-freeness now guarantees that it can happen only finitely many times that decreasing the number of $\circ$ at the side of a position p can through a sequence of splitting steps lead to increasing this number.

7 Satisfiability of Chain and Concatenation-Free Formula

In this section, we explain a decision procedure of a chain and concatenation-free formula. The algorithm is essentially a combination of two standard techniques. First, concatenation and chain-free conjunction over relational constraints is a formula in the "acyclic fragment" of [6] that can be turned into a single equivalent transducer constraint (an approach used also in e.g. [14]). Second, consistency of the resulting transducer with the input length constraints may be checked via computation of the Parikh image of the transducer.

We will now describe the two steps in a more detail. For simplicity, we will assume only transducer and length constraints. This is without loss of generality because the other types of constraints can be encoded to transducers.

Transducer constraints. A conjunction of transducer constraints may be decided through computing an equisatisfiable multi-tape transducer constraint and checking emptiness of its language. The transducer constraint is computed by synchronizing pairs of constraints in the conjunction. That is, synchronization of two transducer constraints $\mathcal{T}_1(x_1, \ldots, x_n)$ and $\mathcal{T}_2(y_1, \ldots, y_m)$ is possible if they share at most one variable (essentially the standard automata product construction where the two transducers synchronise on the common variable). The result of their synchronization is then a constraint $\mathcal{T}_1 \wedge_{(i,j)} \mathcal{T}_2(x_1, \ldots, x_n, y_1 \ldots y_{j-1}, y_{j+1}, \ldots, y_m)$ where y_j is the common variable equal to x_i for some $1 \leq i \leq n$ or a constraint $\mathcal{T}_1 \wedge \mathcal{T}_2(x_1, \ldots, x_n, y_1, \ldots, y_m)$ if there is no common

variable. The $\mathcal{T}_1 \wedge \mathcal{T}_2$ is a lose version of $\wedge_{(i,j)}$ that does not synchronise the two transition relations (see e.g. [14, 20] for details on implementation of a similar construction). Since the original constraint is chain and concatenation-free, two constraints may share at most one variable. This property is an invariant under synchronization steps, hence they may be preformed in any order until only single constraint remains. Termination of this procedure is immediate because every step decreases the number of constraints.

Length constraints. A formula of the form $\Psi_r \wedge \Psi_l$ where Ψ_r is a conjunction of relational constraints and Ψ_l is a conjunction of length constraints may be decided through replacing Ψ_r by a Presburger formula Ψ'_r over length constraints that captures the length constraints implied by Ψ_r. That is, an assignment $\nu : \{|x| \mid x \in \mathbb{X}\} \to \mathbb{N}$ is a solution of Ψ'_r if and only if there is a solution η of Ψ_r such that $|\eta(x)| = \nu(|x|)$ for all $x \in \mathbb{X}$. The conjunction $\Psi'_r \wedge \Psi_l$ is then an existential Presburger formula equisatisfiable to the original conjunction, solvable by an of-the-shelf SMT solver.

Construction of Ψ'_r is based on computation of the Parikh image of the synchronised constraint $\mathcal{T}(x_1,\ldots,x_n)$ equivalent to Ψ_r. Since $\mathcal{T}$ is a standard finite automaton over the alphabet of n-tuples Σ^n_ε, its Parikh image can be computed in the form of a semi-linear set represented as an existential Presburger formula Ψ_{Parikh} by a standard automata construction (see e.g. [32]). The formula captures the relationship between the numbers of occurrences of letters of Σ^n_ε in words of $\mathcal{L}(\mathcal{T})$. Particularly, the numbers of letter occurrences are represented by the *Parikh variables* $\mathbb{P} = \{\#\alpha \mid \alpha \in \Sigma^n_\varepsilon\}$ and it holds that The formula$\nu \models \Psi_{\text{Parikh}}$ iff there is a word $w \in \mathcal{L}(\mathcal{T})$ such that for all $\alpha \in \Sigma^n_\varepsilon$, α appears $\nu(\#\alpha)$ times in w.

The formula Ψ'_r is then extracted from Ψ_{Parikh} as follows. Let $\mathbb{A} = \{\#a_i \mid a \in \Sigma, 1 \leq i \leq n\}$ be a set of *auxiliary variables* expressing how many times the letter $a \in \Sigma$ appears on the ith position of a symbol from Σ^n_ε in a word from $\mathcal{L}(\mathcal{T})$. Let $\alpha[i]$ denotes the ith component of the tuple $\alpha \in \Sigma^n_\varepsilon$. We construct the formula Φ that uses variables $\mathbb{A}$ to describe the relation between values of $|x_1|,\ldots,|x_n|$ and variables of $\mathbb{P}$:

$$\Phi := \bigwedge_{i=1}^{n} \left(|x_i| = \sum_{a\in\Sigma} \#a_i \wedge \bigwedge_{a\in\Sigma} \left(\#a_i = \sum_{\alpha\in\Sigma^n_\varepsilon \text{ s.t. } \alpha[i]=a} \#\alpha\right)\right)$$

We the obtain Ψ'_r by eliminating the quantifiers from $\exists\mathbb{P}\exists\mathbb{A} : \Phi \wedge \Psi_{\text{Parikh}}$.

8 Experimental Results

We have implemented our decision procedure in SLOTH [14] and then used it in the over-approximation module of the string solver TRAU+, which is an extension of TRAU[3]. TRAU+ is as an open source string solver and used Z3 [11] as the SMT solver to handle generated arithmetic constraints. TRAU+ is based on a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. These two modules interact together in order to automatically make these approximations more precise. The extension of SLOTH in the over-approximation module of TRAU+ takes as an input a constraint and checks if it belongs to the weakly-chaining fragment. If it is the case, then we use our decision procedure outlined above. Otherwise, we start by choosing a minimal set of occurrences

		Ostrich	Z3-str3	CVC4	TRAU	TRAU+
Chain-Free (26)	sat	0	-	-	-	**5**
	unsat	0	-	-	-	**14**
	timeout	6	-	-	-	7
	error/unknown	20	-	-	-	0
ReplaceAll (120)	sat	**106**	-	-	26	105
	unsat	**14**	-	-	4	**14**
	timeout	0	-	-	0	1
	error/unknown	0	-	-	90	0
Replace (3392)	sat	1250	298	1278	1174	**1287**
	unsat	2022	2075	2079	2080	**2081**
	timeout	3	903	9	24	23
	error/unknown	117	116	26	114	1
PyEx-td (5569)	sat	36	839	4178	4244	**4245**
	unsat	299	1477	1281	1287	**1287**
	timeout	0	3027	105	35	35
	error/unknown	5234	226	5	3	2
PyEx-z3 (8414)	sat	35	1211	5617	6680	**6681**
	unsat	466	**1870**	1346	1357	1357
	timeout	0	4760	1449	374	374
	error/unknown	7913	573	2	3	2
PyEx-zz (11438)	sat	38	2840	**9817**	8966	8967
	unsat	141	1974	**1202**	1192	1193
	timeout	0	5988	416	1277	1276
	error/unknown	11259	636	3	3	2
Total (28959)	solved	4407	12730	26798	27010	**27236**
	unsolved	24552	16229	2161	1949	**1723**

Table 1: Results of running solvers over Chain-Free, two sets of the SLOG, and four sets of PyEx suite.

of variables x that needs to be replaced by fresh ones such that the resulting constraint falls in our decidable fragment.

We compare TRAU+ performance against the performance of four other state-of-the-art string solvers, namely Ostrich [10], Z3-str3 [7], CVC4 1.6 [18, 19], and TRAU [1]. For our comparison with Z3-str3, we use the version that is part of Z3 4.8.4. The goal of our experiments is twofold:

- TRAU+ handles transducer constraints in an efficient manner TRAU+ can handle more cases than TRAUsince the new over-approximation of TRAU+ supports more and new transducer constraints that the one of TRAU.
- TRAU+ performs either better or as well as existing tools on transducer-less benchmarks.

We carry experiments on suites that draw from the real world applications with diverse characteristics. The first suite is our new suite Chain-Free. Chain-Free is ob-

tained from variations of various PHP code, including the introductory example. The second suite is SLOG [36] that is derived from the security analysis of real web applications. The suite was generated by Ostrich group. The last suite is PyEx [27] that is derived from PyEx - a symbolic executor designed to assist Python developers to achieve high coverage testing. The suite was generated by CVC4 group on 4 popular Python packages: `httplib2`, `pip`, `pymongo`, and `requests`. The summary of experimenting Chain-Free, SLOG, and PyEx is given in Table 1. All experiments were performed on an Intel Core i7 2.7Ghz with 16 GB of RAM. The time limit is 30s for each test which is widely used in the evaluation of other string solvers. Additionally, we use 2700s for Chain-Free suite - much larger than usual as its constraints are difficult. Rows with heading "sat"("unsat") indicate the number of times the solver returned satisfiable (unsatisfiable). Rows with heading "timeout" indicate the number of times the solver exceeded the time limit. Rows with heading "error/unknown" indicate the number of times the solver either crashed or returned unknown.

Chain-Free suite consists of 26 challenging chain-free tests, 6 of them being also straight-line. The tests contain Concatenation, ReplaceAll, and general transducers constraints encoding various JavaScript and PHP functions such as htmlescape, escapeString. Since Z3-str3, CVC4, and TRAU do not support the language of general transducers, we skip performing experiments on those tools in the suite. Ostrich returns 6 times "timeout" for straight-line tests, and times 20 "unknown" for the rest. TRAU+ handles well most cases, and gets "timeout" for only 7 tests.

SLOG suite consists of 3512 tests which contain transducer constraints such as Replace and ReplaceAll. Since Z3-str3 and CVC4 do not support the ReplaceAll function, we skip doing experiments on those tools in the ReplaceAll set. In both sets, the result shows that TRAU+ clearly improved TRAU. In particular, TRAU+ can handle most cases where TRAU returns either "unknown" and "timeout". TRAU+ has also better performance than other solvers.

PyEx suite consists of 25421 tests which contain diverse string constraints such as IndexOf, CharAt, SubString, Concatenation. TRAU and CVC4 have similar performance on the suite. While TRAU is better on PyEx-dt and PyEx-z3 sets (3 less error/unknown results, roughly 1000 less timeouts), CVC4 is better on PyEx-zz set (about 800 less timeouts). CVC4 and TRAU clearly have an edge over Z3-str3 in all aspects. Comparing with Ostrich on this benchmark is problematic because it mostly fails due to unsupported syntactic features. TRAU+ is better than TRAU on all three benchmark sets. This shows that our proposed procedure is efficient in solving not only transducer examples, but also in transducer-less examples.

To summarise our experimental results, we can see that:

- TRAU+ handles more transducer examples in an efficient manner. This is illustrated by the Chain-Free and Slog suites. The experiment results on these benchmarks show that TRAU+ outperforms TRAU. Many tests on which TRAU returns "unknown" are now successfully handled by TRAU+ .
- TRAU+ performs as well as existing tools on transducer-less benchmarks and in fact sometimes TRAU+ outperforms them. This is illustrated by the PyEx suite. In fact, this benchmark is handled very well by TRAU, but nevertheless, as evident from the table, our tool is doing better than TRAU. In fact, As TRAU+ out-

performs TRAU in some examples in the PyEx suite. In those examples, TRAU returned "unknown' while TRAU+ returned "unsat". This means that the new over-approximation not only improves TRAU in transducer benchmarks, but it also improves TRAU in transducer-less examples. Furthermore, observe that the PyEx suite has only around 4000 "unsat" cases out of 25k cases.

9 Related Work

Already in 1946, Quine [26] showed that the first order theory of string equations is undecidable. An important line of work has been to identify subclasses for which decidability can be achieved. The pioneering work by Makanin [21] proposed a decision procedure for quantifier-free word equations, i.e., Boolean combinations of equalities and disequalities, where the variables may denote words of arbitrary lengths. The decidability and complexity of different subclasses have been considered by several works, e.g. [24, 25, 22, 28, 31, 13, 12]. Generalizations of the work of Makanin by adding new types of constraints have been difficult to achieve. For instance, the satisfiability of word equations combined with length constraints of the form $|x| = |y|$ is open [8]. Recently, regular and especially relational transducers constraints were identified as a strongly desirable feature of string languages especially in the context software analysis with an emphasis on security. Adding these to the mix leads immediately to undecidability [23] and hence numerous decidable fragments were proposed [4, 6, 20, 9, 10]. From these, the straight line fragment of [20] is the most general decidable combination of concatenation and transducers. It is however incomparable to the acyclic fragment of [4] (which does not have transducers but could be extended with them in a straightforward manner). Some works add also other syntactic features, such as [9, 10], but the limit of decidable combinations of the core string features—transducers/regular constraints, length constraints, and concatenation stays at [20] and [4]. The weakly chaining decidable fragment present in this paper significantly generalises both these fragments in a practically relevant direction.

The strong practical motivation in string solving led to a rise of a number of SMT solvers that do not always provide completeness guarantees but concentrate on solving practical problem instances, through applying a variety of calculi and algorithms. A number of tools handle string constraints by means of *length-based under-approximation* and translation to bit-vectors [17, 29, 30], assuming a fixed upper bound on the length of the possible solutions. Our method on the other hand allows to analyse constraints without a length limit and with completeness guarantees. More recently, also *DPLL(T)-based* string solvers lift the restriction of strings of bounded length; this generation of solvers includes Z3-str3 [7], Z3-str2 [38], CVC4 [18], S3P [33, 34], Norn [5], Trau [3], Sloth [14], and Ostrich [10]. DPLL(T)-based solvers handle a variety of string constraints, including word equations, regular expression membership, length constraints, and (more rarely) regular/rational relations; the solvers are not complete for the full combination of those constraints though, and often only decide a (more or less well-defined) fragment of the individual constraints. Equality constraints are normally handled by means of splitting into simpler sub-cases, in combination with powerful techniques for Boolean reasoning to curb the resulting exponential search space. Our

implementation is combining strong completeness guarantees of [14] extended to handle the fragment proposed in this paper with an efficient approximation techniques of [3] and its performance on existing benchmarks compares favourably with the most efficient of the above tools.

A further direction is *automata-based* solvers for analyzing string-manipulated programs. Stranger [37] soundly over-approximates string constraints using regular languages, and outperforms DPLL(T)-based solvers when checking single execution traces, according to some evaluations [16]. It has recently also been observed [36, 14] that automata-based algorithms can be combined with model checking algorithms, in particular IC3/PDR, for more efficient checking of the emptiness for automata. However, many kinds of constraints, including length constraints and word equations, cannot be handled by automata-based solvers in a complete manner.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau String Solver. https://github.com/diepbp/FAT, https://github.com/diepbp/FAT
2. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Flatten and conquer: a framework for efficient analysis of string constraints. In: PLDI. ACM (2017)
3. Abdulla, P.A., Atig, M.F., Chen, Y., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: Trau: SMT solver for string constraints. In: FMCAD. IEEE (2018)
4. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: CAV'14. LNCS, vol. 8559. Springer (2014)
5. Abdulla, P.A., Atig, M.F., Chen, Y., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: CAV'15. LNCS, vol. 9206. Springer (2015)
6. Barceló, P., Figueira, D., Libkin, L.: Graph logics with rational relations. Logical Methods in Computer Science **9**(3) (2013). https://doi.org/10.2168/LMCS-9(3:1)2013
7. Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: A string solver with theory-aware branching. CoRR **abs/1704.07935** (2017)
8. Büchi, J.R., Senger, S.: Definability in the existential theory of concatenation and undecidable extensions of this theory. Z. Math. Logik Grundlagen Math. **34**(4) (1988)
9. Chen, T., Chen, Y., Hague, M., Lin, A.W., Wu, Z.: What is decidable about string constraints with the replaceall function. Proc. ACM Program. Lang. **2**(POPL) (Dec 2018). https://doi.org/10.1145/3158091, http://doi.acm.org/10.1145/3158091
10. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290362, http://doi.acm.org/10.1145/3290362
11. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. LNCS, vol. 4963. Springer (2008)
12. Ganesh, V., Berzish, M.: Undecidability of a theory of strings, linear arithmetic over length, and string-number conversion. CoRR **abs/1605.09442** (2016)
13. Ganesh, V., Minnes, M., Solar-Lezama, A., Rinard, M.: Word equations with length constraints: Whats decidable? In: HVC'13, LNCS, vol. 7857. Springer (2013)
14. Holík, L., Janku, P., Lin, A.W., Rümmer, P., Vojnar, T.: String constraints with concatenation and transducers solved efficiently. PACMPL **2**(POPL) (2018). https://doi.org/10.1145/3158092, http://doi.acm.org/10.1145/3158092
15. Hu, Q., D'Antoni, L.: Automatic program inversion using symbolic transducers. SIGPLAN Not. **52**(6) (Jun 2017)

16. Kausler, S., Sherman, E.: Evaluation of string constraint solvers in the context of symbolic execution. In: ASE '14. ACM (2014)
17. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A Solver for String Constraints. In: ISTA'09. ACM (2009)
18. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV'14. LNCS, vol. 8559. Springer (2014)
19. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: CVC4 (2016), http://cvc4.cs.nyu.edu/papers/CAV2014-strings/
20. Lin, A.W., Barceló, P.: String solving with word equations and transducers: towards a logic for analysing mutation XSS. In: POPL'16. ACM (2016)
21. Makanin, G.: The problem of solvability of equations in a free semigroup. Mathematics of the USSR-Sbornik **32**(2) (1977)
22. Matiyasevich, Y.: Computation paradigms in light of hilberts tenth problem. In: New Computational Paradigms. Springer, New York (2008)
23. Morvan, C.: On rational graphs. In: Foundations of Software Science and Computation Structures. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
24. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. J. ACM **51**(3) (2004)
25. Plandowski, W.: An efficient algorithm for solving word equations. In: STOC'06. ACM (2006)
26. Quine, W.V.: Concatenation as a basis for arithmetic. J. Symb. Log. **11**(4) (1946)
27. Reynolds, A., Woo, M., Barrett, C., Brumley, D., Liang, T., Tinelli, C.: Scaling up dpll(t) string solvers using context-dependent simplification. In: Majumdar, R., Kunčak, V. (eds.) CAV'17. Springer (2017)
28. Robson, J.M., Diekert, V.: On quadratic word equations. In: STACS (1999)
29. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A Symbolic Execution Framework for JavaScript. In: IEEE Symposium on Security and Privacy. IEEE (2010)
30. Saxena, P., Hanna, S., Poosankam, P., Song, D.: FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications. In: NDSS. The Internet Society (2010)
31. Schulz, K.U.: Makanin's algorithm for word equations - two improvements and a generalization. In: IWWERT (1990)
32. Seidl, H., Schwentick, T., Muscholl, A., Habermehl, P.: Counting in trees for free. In: Automata, Languages and Programming. Springer Berlin Heidelberg (2004)
33. Trinh, M.T., Chu, D.H., Jaffar, J.: S3: A symbolic string solver for vulnerability detection in web applications. In: CCS'14. ACM (2014)
34. Trinh, M., Chu, D., Jaffar, J.: Progressive reasoning over recursively-defined strings. In: CAV'16. LNCS, vol. 9779. Springer (2016)
35. TwistIt.tech: PHP tutorials. https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php (2019), https://www.makephpsites.com/php-tutorials/user-management-tools/changing-passwords.php, [Online; accessed 2019-04-29]
36. Wang, H., Tsai, T., Lin, C., Yu, F., Jiang, J.R.: String analysis via automata manipulation with logic circuit representation. In: CAV'16. LNCS, vol. 9779. Springer (2016)
37. Yu, F., Alkhalaf, M., Bultan, T.: Stranger: An automata-based string analysis tool for PHP. In: TACAS. LNCS, vol. 6015. Springer (2010)
38. Zheng, Y., Zhang, X., Ganesh, V.: Z3-str: A Z3-based string solver for web application analysis. In: ESEC/FSE'13. ACM (2013)