Solving String Constraints with Approximate Parikh Image ***

Petr Janků and Lenka Turoňová

Faculty of Information Technology, Brno University of Technology
{ijanku,ituronova}@fit.vutbr.cz

Abstract. In this paper, we propose a refined version of the Parikh image abstraction of finite automata to resolve string length constraints. We integrate this abstraction into the string solver SLOTH, where on top of handling length constraints, our abstraction is also used to speed-up solving other types of constraints. The experimental results show that our extension of SLOTH has good results on simple benchmarks as well as on complex benchmarks that are real-word combinations of transducer and concatenation constraints.

1 Introduction

Strings are a fundamental data type in many programming languages, especially owing to the rapidly growing popularity of scripting languages (e.g. JavaScript, Python, PHP, and Ruby) wherein programmers tend to make heavy use of string variables. String manipulations could easily lead to unexpected programming errors, e.g., cross-site scripting (a.k.a. XSS), which are ranked among the top three classes of web application security vulnerabilities by OWASP [11]. Some renowned companies like Google, Facebook, Adobe and Mozilla pay to whoever (hackers) finds a web application vulnerability such as cross-site scripting and SQL injection in their web applications [1], e.g., Google pays up to $10,000.

In recent years, there have been significant efforts on developing solvers for string constraints. Many rule-based solvers (such as Z3STR2 [15], CVC4 [8], S3P [12]) are quite fast for the class of simple examples that they can handle. They are sound but do not guarantee termination. Other tools for dealing with string constraints (such as NORN [1], SLOTH [6], OSTRICH [4]) are based on automata. They use decision procedures which work with fragments of logic over string constraints that are rich enough to be usable in real-world web applications. They are sound and complete. SLOTH was the first solver that can handle string

* This work has been supported by the Czech Science Foundation (project No. 17-12465S), the IT4Innovations Excellence in Science (project No. LQ1602), and the FIT BUT internal projects FIT-S-17-4014 and FEKT/FIT-J-19-5906.

** We thank you to Lukáš Holík for all the support and encouragement he gave us and also the time he spent with us during discussions.

[1] For more information, see https://www.netsparker.com/blog/web-security/google-increase-reward-vulnerability-program-xss/.

constraints including transducers, however, unlike NORN and OSTRICH it is not able to handle length constraints yet. Moreover, these tools are not efficient on simple benchmarks as the rule-based solvers above.

Example 1. The following JavaScript snippet is an adaptation of an example from [7, 2]:

```
var x = goog.string.htmlEscape(name);
var y = goog.string.escapeString(x);
nameElem.innerHTML = '<button onclick= "viewPerson(\'' + y +
    '\')">' + x + '</button>';
```

This is a typical example of string manipulation in a web application. The code attempts to first sanitise the value of name using the sanitization functions htmlEscape and escapeString from the Closure Library [5]. The author of this code accidentally swapped the order of the two first lines. Due to this subtle mistake, the code is vulnerable to XSS, because the variable y may be assigned an unsafe value. To detect such mistakes, we have to first translate the program and the safety property to a string constraint, which is satisfiable if and only if y can be assigned an unsafe value. However, if we would add the length constraints (e.g. `x.length == 2*y.lenght;`) to the code, none of SLOTH, OSTRICH, or NORN would be able to handle them.

The length constraints are quite common in programs like this. Hence, in this paper, we present how to extend the method of SLOTH to be able to cope with them. Our decision procedure is based on the computation of Parikh images for automata representing constraint functions. Parikh image maps each symbol to the number of occurrences in the string regardless to its position.

For one nonderministic finite automaton, one can easily computate the Parikh image by standard automata procedures. However, to compute an exact Parikh image for a whole formula of contraints is demanding. The existing solution proposes first to compute the product of the automata representing the subformulea and then compute the Parikh image of their product. Unfortunately, the exact computation of the Parikh images is computationally far too expensive. Even more importantly, the resulting semilinear expressions become exponential to the number of automata.

We therefore propose a decision procedure which computes an over-approximation of the exact solution that is sufficiently close to the exact solution. We first compute the membership Parikh images of the automata representing the string constraints. Then we use concatenation and substitution to compute the over-approximation of the Parikh image of the whole formula. However, we will not get the same result as with the previous approach since the Parikh image forgets the ordering of the symbols in the world. This causes that we could accept even words that are not accepted by the first approach. But even though our method does not provide accurate results, it is able to handle the lenght constraints and solve also real-world cases.

Outline. Our paper is organized as follow. In Section 1, we introduce relevant notions from logic and automata theory. Section 3 presents an introduction to a string language. Section 4 explains the notion of Parikh image and operations on Parikh images. Section 5 presents the main decision procedure. In Section 6, the experimental results are presented.

2 Preliminaries

Bit vector. Let $\mathbb{B} = \{0, 1\}$ be a set of Boolean values and V a finite set of bit variables. *Bit vectors* are defined as functions $b : V \to \mathbb{B}$. In this paper, bit vectors are described by conjunctions of literals over V. We will denote the set of all bit vectors over V by $\mathbb{P}(V)$ and a set of all formulae over V by $\mathbb{F}_V$.

Further, let $k \geq 1$, and let $V\langle k\rangle = V \times [k]$ where $[k]$ denotes the set $\{1, \ldots, k\}$. Given a word $w = b_1^k \ldots b_m^k \in \mathbb{P}(V\langle k\rangle)^*$ over bit vectors, we denote by $b_j^k[i] \times \{i\} = b_j^k \cap (V \times \{i\})$, $1 \leq j \leq m$ where $b_j^k[i] \in \mathbb{P}(V)$ the j-th bit vector of the i-th track. Further, $w[i] \in \mathbb{P}(V)^*$ such that $w[i] = b_1^k[i] \ldots b_m^k[i]$ is the word which keeps the content of the i-th track of w only. For a bit vector $b \in \mathbb{P}(V)$, we denote by $\{b\}$ the set of variables in the vector.

Automata and transducers. A *succinct nondeterministic finite automaton* (NFA) over bit variables V is a tuple $\mathcal{A} = (V, Q, \Delta, q_0, F)$ where Q is a finite set of *states*, $\Delta \subseteq Q \times \mathbb{F}_V \times Q$ is *transition relation*, $q_0 \in Q$ is an *initial state*, and $F \subseteq Q$ is a finite set of *final states*. $\mathcal{A}$ accepts a word w iff there is a sequence $q_0 b_1^k q_1 \ldots b_m^k q_m$ where $b_i^k \in \mathbb{P}(V)$ for every $1 \leq i \leq m$ such that $(q_i, \varphi_i, q_{i-1}) \in \Delta$ for every $1 \leq i \leq m$ where $b_i^k \models \varphi_i$, $q_m \in F$, and $w = b_1^k \ldots b_m^k \in \mathbb{P}(V)^*$, $m \geq 0$, where each b_i^k, $1 \leq i \leq m$, is a bit vector encoding the i-th letter of w. The *language* of $\mathcal{A}$ is the set $L(\mathcal{A})$ of accepted words.

A k-track *succinct finite automaton* over V is an automaton $\mathcal{R}\langle k\rangle = (V\langle k\rangle, Q, \Delta, I, F)$, $k \geq 1$. The relation $R(\mathcal{R}\langle k\rangle) \subseteq (\mathbb{P}(V)^*)^k$ *recognised* by $\mathcal{R}$ contains a k-tuple of words $(x_1, \ldots, x_k)$ over $\mathbb{P}(V)$ iff there is a word $w \in L(\mathcal{R})$ such that $x_i = w[i]$ for each $1 \leq i \leq k$. A *finite transducer* (FT) $\mathcal{R}$ is a 2-track automaton.

Strings and languages. We assume a finite alphabet Σ. Σ^* represents a set of finite words over Σ, where the empty word is denoted by ϵ. Let x and y be finite words in Σ^*. The concatenation of x and y is denoted by $x \circ y$. We denote by $|x|$ the length of a word $x \in \Sigma^*$. A language is a subset of Σ^*. The concatenation of languages L, L' is the language $L \circ L' = x \circ x' | x \in L \wedge x' \in L'$, and the iteration L^* of L is the smallest language closed under $\circ$ and containing L and ϵ.

3 String Language

Let $\mathbb{X}$ be a set of variables and x, y be *string variables* ranging over Σ^*. A *string formula* over string terms $\{t_{str}\}^*$ is a Boolean combination of *word equations* $x =$

t_{str} whose right-hand side t_{str} might contain the concatenation operator, *regular constraints* P, *rational constraints* R and arithmetic inequalities:

$$\begin{aligned}\varphi &::= x = t_{str} \mid P(x) \mid R(x,y) \mid t_{ar} \geq t_{ar} \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \\ t_{str} &::= x \mid \epsilon \mid t_{str} \circ t_{str} \\ t_{ar} &::= k \mid |t_{str}| \mid t_{ar} + t_{ar}\end{aligned}$$

In the grammar, x ranges over string variables, $R \subseteq (\Sigma^*)^2$ is assumed to be a binary rational relation on words of Σ^*, and $P \subseteq \Sigma^*$ is a regular language. We will represent regular languages by succinct automata and tranducers denoted as $\mathcal{R}$ and $\mathcal{A}$, respectively. The arithmetic terms t_{ar} are linear functions over term lengths and integers, and arithmetic constraints are inequalities of arithmetic terms. The set of word variables appearing in a term is defined as follows: $\mathit{Vars}(\epsilon) = \emptyset$, $\mathit{Vars}(c) = \emptyset$, $\mathit{Vars}(u) = \{u\}$ and $\mathit{Vars}(t_1 \circ t_2) = \mathit{Vars}(t_1) \cup \mathit{Vars}(t_2)$.

To simplify the representation, 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 since a mixed term can be encoded as a conjunction of the pure terms over $\mathbb{X}$ obtained by replacing every occurrence of a letter $a \in \Sigma$ by a fresh variable x, and adding a regular membership constraint $\mathcal{A}_a(x)$ with $L(\mathcal{A}_a) = \{a\}$.

Semantics. A formula φ is interpreted over an *assignment* $\iota : \mathbb{X}_\varphi \to \Sigma^*$ of its variables $\mathbb{X}_\varphi$ to strings over Σ^*. ι is extended to string terms by $\iota(t_{s_1} \circ t_{s_2}) = \iota(t_{s_1}) \circ \iota(t_{s_2})$ and to arithmetic terms by $\iota(|t_s|) = |\iota(t_s)|$, $\iota(k) = k$ and $\iota(t_i + t_i') = \iota(t_i) + \iota(t_i')$. We formalize the satisfaction relation for word equations, regular constraints, rational constraints, and arithmetic inequalities, assuming the standard meaning of Boolean connectives:

$$\begin{aligned} x = t_{str} \quad &\text{iff} \quad \iota(x) = \iota(t_{str}) \\ \iota(P(x)) = \top \quad &\text{iff} \quad \iota(x) \in P \\ \iota(\mathcal{R}(x,y)) = \top \quad &\text{iff} \quad (\iota(x), \iota(y)) \in \mathcal{R} \\ \iota(t_{i_1} \leq t_{i_2}) = \top \quad &\text{iff} \quad \iota(t_{i_1}) \leq \iota(t_{i_2}) \end{aligned}$$

The truth value of Boolean combinations of formulae under ι is defined as usual. If $\iota(\varphi) = \top$ then ι is a *solution* of φ, written $\iota \models \varphi$. The formula φ is *satisfiable* iff it has a solution, otherwise it is *unsatisfiable*.

The unrestricted string logic is undecidable, e.g., one can easily encode Post Correspondence Problem (PCP) as the problem of checking satisfiability of the constraint $\mathcal{R}(x,x)$ for a rational transducer $\mathcal{R}$ [10]. Therefore, we restrict the formulae to be in so-called *straight-line form*. The definition of *straight-line fragment* as well as a linear-time algorithm for checking whether a formula φ falls into the straight-line fragment is defined in [9].

4 Parikh Image

The Parikh image of a string abstracts from the ordering in the string. Particularly, the Parikh image of a string x maps each symbol a to the number of its

occurrences in the string x (regardless to their position). Parikh image of a given language is then the set of Parikh images of the words of the language.

In this chapter, we present a construction of the Parikh image of a given NFA $\mathcal{A} = (V, Q, \Delta, q_0, F)$. The algorithm is modified version of the algorithm from [13] which computes the Parikh image for a given context-free grammar G. This algorithm contains a small mistake that has been fixed by Barner in a 2006 Master's thesis [3]. Since for every regular grammar there exists a corresponding NFA, we can easily customize the algorithm for NFA such that one can compute an existential Presburger formula $\phi_{\mathcal{A}}$ which characterizes the Parikh image of the language $L(\mathcal{A})$ recognized by $\mathcal{A}$ in the following way.

Let us define a variable $\#_\varphi$ for each $\varphi \in \mathbb{F}_V$, y_t for each $t \in \Delta$, and u_q for each $q \in Q$, respectively. The free variables of $\phi_{\mathcal{A}}$ are variables $\#_\varphi$ and we write $\mathit{Free}(\phi_{\mathcal{A}})$ for the set of all free variables in the formula $\phi_{\mathcal{A}}$. The formula $\phi_{\mathcal{A}}$ is the conjunction of the following three kinds of formulae:

- $u_q + \sum_{t=(q',\varphi,q)\in\Delta} y_t - \sum_{t=(q,\varphi,q')\in\Delta} y_t = 0$ for each $q \in Q$, where the variable u_q is restricted as follows: $u_{q_0} = 1$, $u_{q_F} \in \{0, -1\}$ for $q_F \in F$, and $u_q = 0$ for all other $q \in Q \setminus (\{q_0\} \cup F)$.
- $y_t \geq 0$ for each $t \in \Delta$ since the variable y_t cannot be assigned a negative value.
- $\#_\varphi = \sum_{t=(q,\varphi,q')\in\Delta} y_t$ for each $\varphi \in \mathbb{F}$ to ensure that the value x_φ are consistent with the y_t.
- To express the connectedness of the automaton, we use an additional variable z_q for each $q \in Q$ which reflects the distance of q from q_0 in a spanning tree on the subgraph of $\mathcal{A}$ induced by those $t \in \Delta$ with $y_t \geq 0$. To this end, we add for each $q \in Q$ a formulae $z_q = 1 \wedge y_t \geq 0$ if q is an initial state, otherwise $(z_q = 0 \wedge \bigwedge_{t\in\Delta_q^+} y_t = 0) \vee \bigvee_{t\in\Delta_q^+} (y_t \geq 0 \wedge z_{q'} \geq 0 \wedge z_q = z_{q'} + 1)$ where $\Delta_q^+ = \{(q', \varphi, q) \in \Delta\}$ is a set of ingoing transitions.

The resulting existential Presburger formula is then $\exists z_{q_1}, \ldots, z_{q_n}, u_{q_1}, \ldots, u_{q_n}, y_{t_1}, \ldots, z_{t_m} : \phi_{\mathcal{A}}$ where n is the number of states and m is the number of transitions of the given automaton. This algorithm can be directly applied to transducers where the free variables are $\#_\varphi$ such that $\varphi \in \mathbb{F}_{V\langle 2\rangle}$.

4.1 Operations on Parikh Images

In our decision procedure, we will need to use *projection* of the Parikh image of transducers and *intersection* of Parikh images. We have to find a way how to deal with alphabet predicates of transducers since our version of the intersection of Parikh images works only with alphabet predicates over a non-indexed set of bit variables. The intersection of Parikh images is needed since the alphabet predicates of one automaton can represent a set of symbols which may contains common symbols for more than one automaton. These operations can be implemented in linear space and time.

Projection. Let $\mathcal{R} = (V\langle 2\rangle, Q, \Delta, I, F)$ be a transducer representing a constraint $R(x, y)$ and let $\varphi \in \mathbb{F}_{V\langle 2\rangle}$ be a formula over $\{b^k\} \in 2^{V\langle 2\rangle}$ where $b^k \in \mathbb{P}(V\langle 2\rangle)$. We write $\varphi[x]$ to denote a alphabet projection of φ where $\varphi[x]$ is the subformula of φ such that only contains bits from $b^k[i]$ and i is the position of x in R. Given the Parikh image $\phi_{\mathcal{R}}$ of $\mathcal{R}$, we denote by $\phi_{\mathcal{R}}[x]$ a *projection* of $\phi_{\mathcal{R}}$ where the set of free variables is $\mathit{Free}(\phi_{\mathcal{R}}[x]) = \{\#_{\varphi[x]} \mid \#_{\varphi} \in \mathit{Free}(\phi_{\mathcal{R}})\}$. Further, we need to introduce the auxiliary function λ that assigns to each variable $\#_{\varphi[x]}$ a set $\{\#_{\varphi'} \mid \varphi'[x] = \varphi[x]\}$. The resulting formula of projection $\phi_{\mathcal{R}}$ has then the form $\phi_{\mathcal{R}}[x] = \exists \#_{\varphi_1}, \ldots, \#_{\varphi_n} : \phi_{\mathcal{R}} \wedge \bigwedge_{\#_{\varphi[x]} \in \mathit{Free}(\phi_{\mathcal{R}}[x])} \left(\#_{\varphi[x]} = \sum_{\#_{\varphi} \in \lambda(\#_{\varphi[x]})} \#_{\varphi}\right)$ where $\#_{\varphi_i} \in \mathit{Free}(\phi_{\mathcal{R}})$ for $1 \leq i \leq n$.

Intersection. We assume that both Parikh images have alphabet predicates $\mathbb{F}_V$ over the same set of bit variables V. Given two Parikh images ϕ_1 and ϕ_2, their intersection $\phi_{\curlywedge} = \phi_1 \curlywedge \phi_2$ can be constructed as follows. First, we compute a set of fresh variables $\mathcal{I} = \{\#_{\varphi_1 \curlywedge \varphi_2} \mid \#_{\varphi_1} \in \mathit{Free}(\phi_1) \wedge \#_{\varphi_2} \in \mathit{Free}(\phi_2) \wedge \exists b \in \mathbb{P}(V) : b \models \varphi_1 \wedge \varphi_2\}$ representing the number of common symbols for ϕ_1 and ϕ_2. Next, we define for each Parikh image ϕ_i a function $\tau_i : \mathit{Free}(\phi_i) \to 2^{\mathcal{I}}$ such that $\tau_1(\#_{\varphi_1}) = \{\#_{\varphi_1 \curlywedge \varphi_2} \in \mathcal{I}\}$ and $\tau_2(\#_{\varphi_2}) = \{\#_{\varphi_1 \curlywedge \varphi_2} \in \mathcal{I}\}$. Finally, the intersection is define as $\phi_{\curlywedge} = \phi_1 \wedge \phi_2 \wedge \bigwedge_{\#_{\varphi_1} \in \mathit{Free}(\phi_1)} \left(\#_{\varphi_1} = \sum_{\#_{\varphi_1'} \in \tau_1(\#_{\varphi_1})} \#_{\varphi_1'}\right) \wedge \bigwedge_{\#_{\varphi_2} \in \mathit{Free}(\phi_2)} \left(\#_{\varphi_2} = \sum_{\#_{\varphi_2'} \in \tau_2(\#_{\varphi_2})} \#_{\varphi_2'}\right)$.

5 Decision Procedure

Our decision procedure is based on computation of the Parikh images of the automata representing string constraints. Let $\varphi := \varphi_{cstr} \wedge \varphi_{eq} \wedge \varphi_{ar}$ be a formula in straight-line form where φ_{cstr} is a conjunction of regular constraints (or their negation) and rational constraints, φ_{eq} is a conjunction of word equations of the form $x = y_1 \circ y_2 \circ \cdots \circ y_n$, and φ_{ar} is a conjunction of arithmetic inequalities. The result of the decision procedure is an existential Presburger formula ϕ_{φ} which represents an over-approximation of the Parikh image of φ.

We assume that each variable $x \in \mathit{Vars}(\varphi)$ is restricted by an automaton or a transducer. Note that the function $\mathit{Vars}(\varphi)$ denotes a set of variables appearing in the formula φ. We write $\mathbb{T}$ to denote a set of Parikh images. The procedure is divided into three steps as follows.

- **Step 1**: First, we compute Parikh images of automata and transducers representing the constraints from φ_{cstr} using the algorithm from Sec. 4. We define a mapping $\rho_{cstr} : \mathit{Vars}(\varphi_{cstr}) \to \mathbb{T}$ that maps each string variable $x \in \mathit{Vars}(\varphi_{cstr})$ to the over-approximation of its Parikh image. Let $P_1(x), \ldots, P_n(x)$ and $R_1(x, y), \ldots, R_m(x, y)$ be constraints from φ_{cstr} restricting x. A formula ϕ_x representing the Parikh image of x is then computed using the algorithm from Sec. 4.1 as $\phi_x = \phi_{\mathcal{A}_x} \curlywedge \phi_{\mathcal{A}_1} \curlywedge \ldots \curlywedge \phi_{\mathcal{A}_n} \curlywedge \phi_{\mathcal{R}_1}[x] \curlywedge \ldots \curlywedge \phi_{\mathcal{R}_m}[x]$ where $\phi_{\mathcal{A}_i}$, $0 \leq i \leq n$, is the Parikh image of the automaton $\mathcal{A}_i$ representing $P_i(x)$ and $\phi_{\mathcal{R}_j}$, $0 \leq j \leq m$, is the Parikh image of the transducer $\mathcal{R}_j$ representing $R_j(x, y)$.

– **Step 2**: We define a mapping $\rho_{eq} : \mathit{Vars}(\varphi_{eq}) \to \mathbb{T}$ that maps each string variable $x \in \mathit{Vars}(\varphi_{eq})$ to the over-approximation of its Parikh image as $\phi_x = (\bigwedge_{i=1}^{k} \rho_{cstr}(y_i) \wedge \bigwedge_{j=k+1}^{n} \rho_{eq}(y_j)) \curlywedge \rho_{cstr}(x)$. We assume that $\mathit{Free}(y_1) \cap \cdots \cap \mathit{Free}(y_n) = \emptyset$. This can be done by adding double negation to the alphabet predicates which helps to distinguish free variables of individual y_i. Parikh image does not preserve the ordering of the symbols in the string, therefore, we can reorder the right side of the equation $y_1 \circ \cdots \circ y_k \circ \cdots \circ y_n$ such that $\forall 1 \leq i \leq k : y_i \in \varphi_{cstr}$ and $\forall k \leq j \leq n : y_j \in \varphi_{eq}$. Moreover, the reordering can be done in such a way that each variable on the right side of the equation is already defined since φ falls into the straight-line fragment.
– **Step 3**: Finally, we build the final formula ϕ_φ using mappings ρ_{cstr} and ρ_{eq}. Let $\mathbb{X}_{eq} \subseteq \mathit{Vars}(\varphi_{eq})$ be a set of all variables that are on the left side of the equations. The resulting formula ϕ_φ is then a conjunction $\phi_\varphi = (\curlywedge_{x \in \mathbb{X}_{eq}} \rho_{eq}(x)) \curlywedge (\curlywedge_{x \in \mathit{Vars}(\varphi) \setminus \mathbb{X}_{eq}} \rho_{cstr}(x))$.

6 Experiments

We have implemented our decision procedure extending the method of Sloth [6] as a tool, called PICoSo. Sloth is a decision procedure for the straight-line fragment and acyclic formulas. It uses succinct alternating finite-state automata as concise symbolic representation of string constraints. Like Sloth, PICoSo was implemented in Scala.

To evaluate its performance, we compared PICoSo against Sloth. We performed experiments on benchmarks with diverse characteristics.

		Sloth	PICoSo
Norn (1027)	sat (sec)	**314 (545)**	313 (566)
	unsat (sec)	353 (624)	**356 (602)**
	timeout	0	0
	error/un	360	**358**
SLOG (3392)	sat (sec)	922 (**5526**)	**923** (5801)
	unsat (sec)	2033 (5950)	**2080 (4382)**
	timeout	437	**389**
	error/un	0	0
SLOG-LEN (394)	sat (sec)	0	0
	unsat (sec)	266 (659)	**296 (773)**
	timeout	**4**	15
	error/un	124	**83**

Table 1. Performance of PICoSo in comparison to Sloth.

The first set of benchmarks is obtained from Norn group [1] and implements string manipulating functions such as the Hamming and Levenshtein distances. It consists of small test case that is combinations of concatenations, regular constraints, and length constraints. The second set SLOG [14] is derived from the security analysis of real web applications. It contains regular constraints, concatenations, and transducer constraints such as `Replace` but no length constraints. The last set is obtained from SLOG by selecting 394 examples containing `Replace` operation. It was extended by `RelaceAll` operation and since in practice, it is common to restrict the size of string variables in web applica-

tions, we added length constraints of the form $|x| + |y| Rn$, where $R \in \{=, <, >\}$, $n \in \{4, 8, 12, 16, 20\}$, and x, y are string variables.

The summary of the experiments is shown in Table 1. All experiments were executed on a computer with Intel Xeon E5-2630v2 CPU @ 2.60 GHz and 32 GiB RAM. The time limit was 30 seconds was imposed on each test case. The rows indicate the number of times the solver returned satisfiable/unsatisfiable (sat/unsat), the number of times the solver ran out of 30-second limit (timeout), and the number of times the solver either crashed or returned unknown (error/un).

The results show that PICoSo outperforms Sloth on all of unsat examples. Sloth is however slightly better in case of sat examples due to the addition computation of the over-approximation of the Parikh image. Sloth timed out on 441 cases while PICoSo run out of time only in 404 cases. This shows that our proposed procedure is efficient in solving not only length constraints, but also other types of constraints.

References

1. P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *CAV'14*, pages 150–166, 2014.
2. P. Barceló, D. Figueira, and L. Libkin. Graph logics with rational relations. volume 9, 2013.
3. S. Barner. H3 mit gleichheitstheorien. Master's thesis, Technical University of Munich, Germany, 2006.
4. T. Chen, M. Hague, A. W. Lin, P. Rümmer, and Z. Wu. Decision procedures for path feasibility of string-manipulating programs with complex operations. volume 3, pages 49:1–49:30, 2019.
5. G. co. 2015. Google closure library (referred in nov 2015). `https://developers.google.com/closure/library/`, 2015.
6. L. Holík, P. Janků, A. W. Lin, P. Rümmer, and T. Vojnar. String constraints with concatenation and transducers solved efficiently. *PACMPL*, 2(POPL), 2018.
7. C. Kern. Securing the tangled web. In *ACM 57*, pages 38–47, 2014.
8. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV'14*, 2014.
9. A. W. Lin and P. Barceló. String solving with word equations and transducers: towards a logic for analysing mutation XSS. In *POPL*, pages 123–136, 2016.
10. C. Morvan. On rational graphs. In *FoSSaCS*, pages 252–266, 2000.
11. OWASP. The ten most critical web application security risks. `https://www.owasp.org/images/f/f8/OWASP_Top_10_-_2013.pdf`, 2013.
12. M. Trinh, D. Chu, and J. Jaffar. Progressive reasoning over recursively-defined strings. In *CAV'16*, pages 218–240, 2016.
13. K. N. Verma, H. Seidl, and T. Schwentick. On the complexity of equational horn clauses. In *CADE'05*, pages 337–352, 2005.
14. H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang. String analysis via automata manipulation with logic circuit representation. In *CAV'16*, volume 9779 of *LNCS*, pages 241–260. Springer, 2016.
15. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, M. Berzish, J. Dolby, and X. Zhang. Z3str2: an efficient solver for strings, regular expressions, and length constraints. *Formal Methods in System Design*, 50(2-3):249–288.