TRAU : SMT solver for string constraints

Parosh Aziz Abdulla
Uppsala University, Sweden
parosh@it.uu.se

Mohamed Faouzi Atig
Uppsala University, Sweden
mohamed_faouzi.atig@it.uu.se

Yu-Fang Chen
Academia Sinica, Taiwan
yfc@iis.sinica.edu.tw

Bui Phi Diep
Uppsala University, Sweden
bui.phi-diep@it.uu.se

Lukáš Holík
Brno University of Technology, Czech Republic
holik@fit.vutbr.cz

Ahmed Rezine
Linköping University, Sweden
ahmed.rezine@liu.se

Philipp Rümmer
Uppsala University, Sweden
philipp.ruemmer@it.uu.se

***Abstract*—We introduce TRAU, an SMT solver for an expressive constraint language, including word equations, length constraints, context-free membership queries, and transducer constraints. The satisfiability problem for such a class of constraints is in general undecidable. The key idea behind TRAU is a technique called flattening, which searches for satisfying assignments that follow simple patterns. TRAU implements a Counter-Example Guided Abstraction Refinement (CEGAR) framework which contains both an under- and an over-approximation module. The approximations are refined in an automatic manner by information flow between the two modules. The technique implemented by TRAU can handle a rich class of string constraints and has better performance than state-of-the-art string solvers.**

I. INTRODUCTION

The recent years have seen a wealth of research on *string constraints*, in particular in the form of SMT solvers that can efficiently check satisfiability of quantifier-free formulas over a background theory of strings and regular expressions (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]). String solvers can be applied in a variety of verification approaches, for instance in software model checking to take care of implication and path feasibility checks; the most widespread adoption has occurred in the area of *security analysis* for languages like JavaScript and PHP, for instance to discover information leaks or vulnerability to injection attacks (e.g., [12], [13], [14]). To process constraints from those domains, it is necessary for string solvers to handle a delicate combination of (theoretically and practically) highly challenging operations: *concatenation* in word equations, to model assignments in programs; *context-free grammar,* to model properties or attack patterns; *string length,* to express string manipulation in programs; and *transduction,* to express sanitisation, escape operations, and replacement operations in strings. Since the full combination of those theories is known to be undecidable, many SMT solvers are complete only for certain fragments of the full logic.

In this paper, we present TRAU, an SMT solver for string constraints, that can handle all of the above mentioned operations. TRAU implements the framework of Counter-Example Guided Abstraction Refinement (CEGAR) proposed in [8]. This framework contains both an under- and an over-approximation module. The key idea behind TRAU is a technique called flattening [8]. It is based on the observation that both satisfiability and unsatisfiability of common constraints can be shown through witnesses of simple patterns that can be captured by flat languages (i.e., a language consisting of the set of words in $w_1^* w_2^* \cdots w_n^*$ where $w_1, w_2, \ldots, w_n$ are finite words). Compared to [8], TRAU implements several optimizations that are keys to its current efficiency (namely, a precise and efficient over-approximation module and a better strategy for splitting equalities). Furthermore, TRAU can handle efficiently the case of *transduction,* which is the string operation that is currently least well supported in existing string solvers, albeit extremely important for security analysis, and often a bottleneck in applications. (Observe that the tool in [8] does not support transducer constraints.) We show that transduction can elegantly be reduced to context-free membership constrains. In fact, the technique implemented by TRAU can handle a rich class of string constraints and has better performance than state-of-the-art string solvers.

Related Work. During the last years, several SMT solvers for strings and related logics have been introduced. A number of tools handle string constraints, including context-free membership, by fixing an upper bound on the length of the possible solutions (e.g., [1], [12], [13], [15], [16]). In contrast, the under-approximation module of TRAU does not impose any bound on the length of solutions but rather limits the search only for solutions that belong to flat languages in a similar manner to [8]. More recently, DPLL(T)-based string solvers lift the restriction of strings of bounded length; this generation of solvers includes Z3-str [3], CVC4 [5], S3 [4], Norn [17], and Sloth [11]. Most of those solvers are more restrictive than TRAU in their support for language constraints. To the best of our knowledge, TRAU and Hampi [1] are the only string solvers which can handle context-free membership constraints. Observe that TRAU does not impose any bound on the length of the solutions while Hampi does. Furthermore, TRAU implements a DPLL(T)-style proof procedure for strings in a similar manner to [17] in order to gain in efficiency. Another related technique are automata-based solvers for analyzing string-manipulated programs (e.g., [2], [6], [18]). However, many kinds of constraints, including length constraints, word equations, and context-free grammars, cannot be handled by such automata-based solvers in a complete manner. Compared to [8], TRAU implements several optimizations, including a DPLL(T)-style proof procedure, that are keys to its current efficiency. Furthermore, TRAU supports transducer constraints which is not the case of [8].

II. PRELIMINARIES

Let Σ be a finite alphabet. We use Σ^* to denote the set of finite words over Σ, and use ϵ to denote the empty word. For a word $w \in \Sigma^*$, we use $\texttt{length}(w)$ to denote the length of w. We denote by w^R the reverse image of w. A language $L \subseteq \Sigma^*$ is said to be $(\mathbb{p},\mathbb{q})$-*flat*, for some $\mathbb{p},\mathbb{q} \in \mathbb{N}$, if there are words $w_1, w_2, \ldots, w_{\mathbb{q}} \in \Sigma^*$ such that $\texttt{length}(w_i) \leq \mathbb{p}$ for all $i : 1 \leq i \leq \mathbb{q}$, and $L = (w_1)^* \cdot (w_2)^* \cdots (w_{\mathbb{q}})^*$.

A *Context-Free Grammar* (CFG) is defined by a quadruple $\mathcal{G} = \langle N, T, P, S \rangle$ where N is a finite set of *non-terminals*, T is a finite set of *terminals*, P is a finite set of *productions*, and $S \in N$ is the *start symbol*. The language $\mathcal{L}(\mathcal{G})$ of the grammar $\mathcal{G}$ is defined in the standard manner.

A *Pushdown Automaton* (PDA) is defined by $\mathcal{P} = \langle Q, \Sigma, \Gamma, \Delta, q_{init}, q_{acc} \rangle$ where Q is a finite set of *states*, Σ is a finite input alphabet, Γ is a stack alphabet, $\Delta \subseteq (Q \times \Gamma^* \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \times Q)$ is a finite set of transitions, $q_{init} \in Q$ is the *initial state*, and $q_{acc} \in Q$ is the *accepting state*. The language $\mathcal{L}(\mathcal{P})$ of the pushdown automaton $\mathcal{P}$ is defined in the standard manner (where the stack content is empty at the initial and final configurations). It is well-known that the class of languages accepted by pushdown automata and the one accepted by context free grammars coincide (i.e., given a pushdown automaton $\mathcal{P}$ (resp. a context-free grammar $\mathcal{G}$), one can construct a context-free grammar $\mathcal{G}$ (resp. a pushdown automaton $\mathcal{P}$) such that $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{G})$).

A *Finite-State Transducer* is $\mathcal{T} = \langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$, where Q is a finite set of *states*, Σ is a finite alphabet, $\Delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times (\Sigma \cup \{\epsilon\}) \times Q$ is the transition relation, $q_{init} \in Q$ is the *initial state*, and $q_{acc} \in Q$ is the *accepting state*. For words $w_1, w_2 \in \Sigma^*$, we write $w_2 \in \mathcal{T}(w_1)$ to denote that there is a sequence $q_0 \langle a_1, b_1 \rangle q_1 \langle a_2, b_2 \rangle \cdots \langle a_n, b_n \rangle q_n$ such that $q_0 = q_{init}$, $q_n = q_{acc}$, $\langle q_i, \langle a_{i+1}, b_{i+1} \rangle, q_{i+1} \rangle \in \Delta$ for all $i : 0 \leq i < n$, $w_1 = a_1 a_2 \cdots a_n$, and $w_2 = b_1 b_2 \cdots b_n$.

III. THE STRING CONSTRAINT LANGUAGE

In this section, we define string constraints over a finite alphabet Σ and a finite set of variables $\mathbb{X}$ ranging over Σ^*. The syntax of a formula ψ is given in Figure 1. ψ is given in the conjunctive normal form where each literal clause can be either a string (dis-)equality ϕ_s, a context-free membership ϕ_g, a transducer constraint ϕ_t or an arithmetic constraint ϕ_i. A string equality (resp. disequality) is of the form $tr_s = tr_s$ (resp. $(tr_s \neq tr_s)$) where tr_s is a (string) term. Each string term tr_s is a sequence composed of variables in $\mathbb{X}$ and symbols from Σ.

$$
\begin{array}{rcl}
\psi & ::= & \phi \mid \psi \wedge \psi \\
\phi & ::= & \phi_s \mid \phi_i \mid \phi_t \mid \phi_g \\
\phi_s & ::= & tr_s = tr_s \mid tr_s \neq tr_s \\
\phi_t & ::= & tr_s \in \mathcal{T}(tr_s) \\
\phi_g & ::= & tr_s \in \mathcal{G} \\
\phi_i & ::= & tr_i \geq tr_i \\
tr_s & ::= & w \mid x \mid tr_s \bullet tr_s \\
tr_i & ::= & \texttt{length}(tr_s) \mid k
\end{array}
$$

Fig. 1: Constraint Syntax

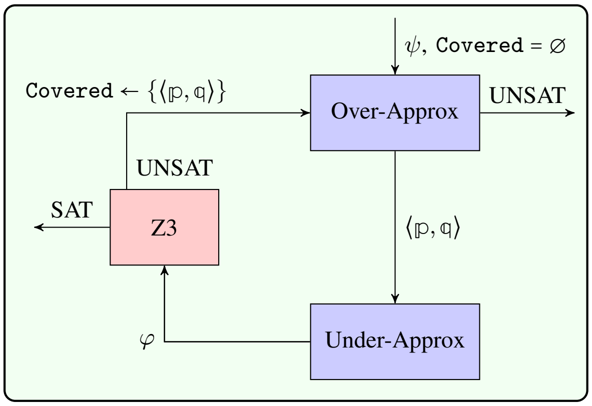
ψ, Covered = $\emptyset$
Covered ← $\{\langle \mathbb{p},\mathbb{q} \rangle\}$
Over-Approx
UNSAT
UNSAT
SAT
Z3
$\langle \mathbb{p},\mathbb{q} \rangle$
φ
Under-Approx

Fig. 2: Architecture of TRAU

Formally, a string term is either a word $w \in \Sigma^*$, a string variable $x \in \mathbb{X}$ or a concatenation of two string terms. A transducer constraint is of the form $tr_s \in \mathcal{T}(tr_s)$ where $\mathcal{T}$ is a transducer and tr_s is a string term. A context-free grammar membership constraint is of the form $tr_s \in \mathcal{G}$ where $\mathcal{G}$ is a context-free grammar and tr_s is a string term. An arithmetic constraint ϕ_i is a relational expression between two integer terms tr_i where an integer term is either the length of a string term $\texttt{length}(tr_s)$ or an integer k.

The formula ψ is said to be *satisfiable* iff there is an interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$ such that η satisfies ψ. Otherwise, it is said to be *unsatisfiable*.

IV. ARCHITECTURE OVERVIEW

In this section, we present the architecture of our tool TRAU which checks the satisfiability of string constraint formulae (as defined in Section III). The architecture of TRAU is shown in Figure 2. TRAU consists of two main modules, namely the *Over-Approx* module and the *Under-Approx* module. It uses the SMT solver Z3 to handle arithmetic constraints.

The Over-Approx module takes as input a formula ψ and a finite set $\texttt{Covered} \subseteq \mathbb{N}^2$ of (abstract) parameters. The set $\texttt{Covered}$ is empty at the beginning. This set stores abstract parameters used by the Under-Approx module to check the satisfiability in previous iterations. The Over-Approx then constructs an over-approximation ψ' of ψ. The formula ψ' is constructed such that it falls in the decidable fragment of the theory of strings with regular membership constraints and length constraints [7]. Thus, we are able to apply similar techniques as the ones used in Norn [7] to check the satisfiability of ψ'. If ψ' is unsatisfiable, then ψ is unsatisfiable, and TRAU terminates. If ψ' is satisfiable, a satisfying assignment for ψ' is returned. Then we extract an abstract parameter $\alpha = \langle \mathbb{p},\mathbb{q} \rangle \in \mathbb{N}^2$ from the satisfying interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$

as follows: α is one of minimal pairs such that for any variable $x \in \mathbb{X}$, the word $\eta(x)$ belongs to an *α-flat language* [8].

The Under-Approx module takes as input the abstract parameter α and the set of constraints ψ. It limits the search only for solutions of ψ that belong to an *α-flat language*. By [8], checking the existence of a solution ψ that belongs to an *α-flat language* can be reduced to the satisfiability problem of an existential Presburger formula. Therefore, the Under-Approx module produces as output an existential Presburger formula φ such that φ is satisfiable iff there is an interpretation $\eta : \mathbb{X} \mapsto \Sigma^*$ such that η satisfies ψ and for every variable $x \in \mathbb{X}$, we have that $\eta(x)$ belongs to an *α-flat language*.

Then, Z3 checks the satisfiability of the existential Presburger formula φ. If Z3 returns that φ is satisfiable, then we deduce that ψ is also satisfiable. In that case, we can even construct an interpretation η that satisfies ψ, and Trau terminates. In the case Z3 returns that φ is unsatisfiable, we are unable to find a solution of ψ that is accepted by an *α-flat language*. Thus, α is added to the set `Covered` and the control is given back to the *Over-Approx* module to produce a new pair α which is not in `Covered` (by requiring that the solutions do not belong to an *α-flat language*).

V. Efficient Handling of Transducer Constraints

Trau handles transducer constraints differently from the method presented in [8]. Rather than extending the Under-Approx module to transducers, we transform transducer constraints to context-free membership constraints. Let ψ be a string constraint and let ϕ_t be a transducer constraint appearing in ψ. Let us assume that ϕ_t is of the form $t' \in \mathcal{T}(t)$ where $\mathcal{T} = \langle Q, \Sigma, \Delta, q_{init}, q_{acc} \rangle$ is a transducer and t and t' are string terms. In order to construct the context-free membership constraints, we first construct a pushdown automaton $\mathcal{P}$ such that a word w is accepted by $\mathcal{P}$ iff there are two words u and v such that $u \in \mathcal{T}(v)$ and $w = v \cdot \sharp \cdot u^R$ where $\sharp$ is a fresh symbol (not in Σ). The pushdown automaton $\mathcal{P} = \langle Q \cup \{q_{final}\}, \Sigma \cup \{\sharp\}, \Sigma, \Delta', q_{init}, q_{final} \rangle$ has the same set of states as $\mathcal{T}$ plus one extra accepting state $q_{final} \notin Q$. Any accepting run of $\mathcal{P}$ can be split into two phases. In the first phase, the pushdown automaton simulates the transducer by: (i) performing the same changes on the state, (ii) reading the same input letter, and (iii) pushing into the stack the output letter read by the transducer. Formally, for each transition $\langle q, \langle a, b \rangle, q' \rangle$ of $\mathcal{T}$, the pushdown automaton $\mathcal{P}$ has a transition of the form $\langle q, \epsilon, a, b, q' \rangle$. At the end of this phase, the pushdown automaton reaches the same state as the transducer, reads the same input word, and stores the output word read by the transducer into its stack. The second phase of the pushdown automaton $\mathcal{P}$ starts, in non-deterministic manner, when its current state is q_{acc}. First, the pushdown moves its state from q_{acc} to q_{final} while reading the special $\sharp$ (i.e., the pushdown automaton $\mathcal{P}$ has the following transition $\langle q_{acc}, \epsilon, \sharp, \epsilon, q_{final} \rangle$). From the state q_{final}, the pushdown automaton $\mathcal{P}$ starts emptying its stack while reading each popped symbol (i.e., the pushdown automaton $\mathcal{P}$ has a transition of the form $\langle q_{final}, a, a, \epsilon, q_{final} \rangle$ for each letter $a \in \Sigma$). It is easy to see that a word w is in $\mathcal{L}(\mathcal{P})$ iff there are two words u and v such that $u \in \mathcal{T}(v)$ and $w = v \cdot \sharp \cdot u^R$.

Let $\mathcal{G}$ be a context-free grammar that accepts the same language as the pushdown automaton $\mathcal{P}$ (i.e., $\mathcal{L}(\mathcal{G}) = \mathcal{L}(\mathcal{P})$). Let $\mathcal{G}_1$ (resp. $\mathcal{G}_2$) be the context-free grammar that accepts exactly the following set of words $\{w \cdot \sharp \cdot w^R \,|\, w \in \Sigma^*\}$ (resp. Σ^*).

Now, we can replace the transducer constraint ϕ_t by the conjunction of the following context-free membership constraints: $t \cdot \sharp \cdot y \in \mathcal{G}$, $y \cdot \sharp \cdot t' \in \mathcal{G}_1$ and $t \cdot y \cdot t' \in \mathcal{G}_2$ where y is a fresh variable. Observe that we need the constraint $t \cdot y \cdot t' \in \mathcal{G}_2$ to enforce that the interpretations $\eta(y)$, $\eta(t)$, and $\eta(t')$ are over the alphabet Σ (since the alphabet of the newly constructed formulas is $\{\Sigma \cup \sharp\}$). Let us assume that ψ' is the string constraint obtained from ψ by replacing any transducer constraint by the conjunction of the three context-free membership constraints (constructed as described above). Then, it is easy to see that ψ is satisfiable iff ψ' is satisfiable.

VI. Optimizing the Over-Approximation Module

Suppose that we have a constraint formula ψ together with a set `Covered` $\subseteq \mathbb{N}^2$ of parameter values. We assume w.l.o.g. that ψ does not contain any transducer constraints (see Section V). The over-approximation module in [8] proceeds as follows: First, it replaces any context-free membership constraint of the form $tr_s \in \mathcal{G}$ in ψ by a constraint of the form $tr_s \in L$ where L is a regular language accepting the upward closure of $\mathcal{L}(\mathcal{G})$ [19], [20]. Then, it limits the search only for solutions that do not belong to any *α-flat language* with $\alpha \in$ `Covered`. Finally, it replaces any occurrence of a variable x by a fresh copy of x that satisfies the same word equation, membership and length constraints as x. The resulting string constraints falls in the decidable fragment of the theory of strings [7], [17]. In contrast, Trau adopts a lazy approach in the replacement of variables. More precisely, Trau starts by choosing an occurrence of a variable x to replace by a fresh copy that satisfies the same membership and length constraints. Then, Trau checks if the resulting string constraint satisfies the *acyclicity* condition of [7], [17]. If it is the case then the replacement procedure terminates. Otherwise, Trau chooses another occurrence of a variable to replace by a fresh copy.

VII. Optimizing the Under-Approximation Module

We present one important optimization that Trau implements. This optimization significantly improves the Under-Approx module (implemented in [8]) when applied to equality constraints. In practice, after flattening an equality constraint (i.e., computing a finite-state automaton that characterizes the intersection of flat languages), the size of the constructed automaton A could become fairly large. Consequently, the arithmetic SMT solver may have poor performance when checking the satisfiability of the constructed existential Presburger formula characterizing the Parikh image [21], [22] of A. We found that problem can be improved by combining the flattening technique proposed in [8] with the DPLL(T)-style proof procedure and the length-guided splitting of equalities

		CVC4	Z3-str3	S3P	TRAU-PRE	TRAU
Kaluza suite	sat	35235	34495	**35264**	35202	**35264**
	unsat	**12014**	11799	**12014**	12019	**12014**
	timeout	35	350	6	63	6
	error/unknown	0	640	0	0	0
PISA suite	sat	7	**8**	6	-	**8**
	unsat	4	**4**	1	-	**4**
	timeout	0	0	5	-	0
	error/unknown	1	0	0	-	0
AppScan suite	sat	7	**8**	6	-	**8**
	unsat	0	0	0	-	0
	timeout	1	0	1	-	0
	error/unknown	0	0	1	-	0
Transducer suite	sat	-	-	3	-	**11**
	unsat	-	-	10	-	2
	timeout	-	-	0	-	4
	error/unknown	-	-	4	-	0
StringFuzz suite	sat	618	605	-	-	**723**
	unsat	160	190	-	-	**261**
	timeout	247	207	-	-	0
	error/unknown	0	23	-	-	41

TABLE I: Experimental results. All *satisfying* results of TRAU are cross-checked by S3P to guarantee correct solutions. Runtime was limited to 20s for the Kaluza, PISA, AppScan, StringFuzz suites and to 100s for the Transducer suite. The row "(*un*)*sat*" indicates the number of benchmarks for which the solvers report (un)satisfiable.

procedure used in [7]. This is mainly due to the fact that we limit the search for solutions that belong to α-flat languages.

Fix a set of constraints ψ, a finite set of variables $\mathbb{X}$, and an abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$. To handle the equality constraints efficiently, we proceed as follows: First, we construct the string constraint ψ' by replacing any occurrence of a variable x in ψ, that belongs to an $(\mathbb{p}, \mathbb{q})$-flat language, by $x_1 \cdot x_2 \cdots x_{\mathbb{q}}$ where $x_1, x_2, \ldots, x_{\mathbb{q}}$ are fresh variables that belong to $(\mathbb{p}, 1)$-flat languages. Assume w.l.o.g that ψ' contains an equality constraint ϕ_s of the form $x_1 \cdots x_m = y_1 \cdot y_2 \cdots y_n$. Observe that $x_1, \ldots, x_m, y_1, \ldots, y_n$ belong to $(\mathbb{p}, 1)$-flat languages. Then, for every $j : 1 \leq j \leq m$ (resp. $i : 1 \leq i \leq n$), we construct a string constraint φ (resp. φ') from ψ' by: (1) deleting the equality constraint ϕ_s from ψ', (2) replacing any occurrence of the variable y_1 (resp. x_1) by $x_1 \cdot x_2 \cdots x_j$ (resp. $y_1 \cdot y_2 \cdots y_i$), and (3) adding the equality constraint $x_{j+1} \cdots x_m = y_2 \cdots y_n$ (resp. $x_2 \cdots x_m = y_{i+1} \cdots y_n$). For each string constraint φ (resp. ψ'), we repeat the procedure of splitting of the equality constraints until the obtained string constraint does not contain equality constraints. Finally, we declare the string constraint ψ to be satisfiable if one of the constructed string constraints is satisfiable; otherwise we add the abstract parameter $\alpha = \langle \mathbb{p}, \mathbb{q} \rangle$ to the set `Covered`.

Observe that such a splitting strategy will limit the search space for solutions to a subset of $(\mathbb{p}, \mathbb{q})$-flat languages. However, this is not a restriction since if ψ is satisfiable then for the abstract parameter $\alpha = \langle 1, \mathbb{q} \rangle$, with $\mathbb{q}$ is the maximal length of the strings appearing in a satisfying assignment of ψ, the splitting strategy will lead to a satisfiable string constraint.

This splitting strategy is also significantly improved by using a DPLL(T)-Style proof procedure and a length-guided splitting procedure as in [7].

VIII. EXPERIMENTAL RESULTS

In this section, we describe the experimental evaluation of the TRAU solver to validate the effectiveness of the techniques presented in the paper. We have implemented TRAU as an open source solver and used Z3 [23] as the SMT solver to handle generated arithmetic constraints from the Under-Approx module. TRAU takes inputs in SMTLIB format. TRAU does not run any parts concurrently to boost the performance. We compare TRAU against four other state-of-the-art string solvers, namely Z3-str3 [10], CVC4 [5], [24] (the newest version), S3P [25], and TRAU-PRE [26]. We do not compare with Sloth [11] since it does not support length constraints which disqualifies it in a majority of our test cases. For our comparison with Z3-str3, we use the version that is part of Z3 4.6. Each benchmark suite draws from real world applications with diverse characteristics. The summary of the results is given in Table I. All experiments were performed on an Intel Core i7 2.7Ghz with 8 GB of RAM. In most experiments, the time limit is 20s since it is widely used in the evaluation of other string solvers.

Kaluza suite. The Kaluza suite [12] is generated by a JavaScript symbolic execution engine. It consists of 47284 test cases, including length, regular and (dis)equality constraints. For this suite, CVC4 times out on 35 cases while TRAU-PRE times out on 63 cases. Z3-str3 times out on 350 cases and cannot answer on 640 cases. TRAU and S3P have the same performance, which is better than the other solvers as they time out only in 6 cases. When increasing the timeout to 40s, TRAU can solve all the remaining cases (they all are *sat* cases) while other solvers cannot.

PISA and AppScan suite. The PISA suite includes constraints from real-world Java sanitizer methods that were used in the evaluation of the PISA system [27]. The suite has 12 tests, including transducer constraints such as Substring, IndexOf, and Replace operations. The AppScan suite is derived from security warnings output by IBM Security AppScan Source Edition [28]. The suite has 8 tests, including transducer constraints and (dis)equality constraints. In both suites, the performance of TRAU is comparable to Z3-str3 (they are able to solve all test cases). CVC4 cannot give an answer for 1 test case in each suite. TRAU-PRE cannot run these suites since it does not support transducer constraints.

Transducer suite. The Transducer suite is inspired by the Google closure library [29], which supports sanitizing strings to protect websites from vulnerabilities. The suite has 17 tests, including transducer constraints such as Replace and ReplaceAll. Since only S3P and TRAU support ReplaceAll constraints, we do not include Z3-str3, CVC4, and TRAU-PRE in this comparison. Within the time limit, TRAU showed the satisfiability of 11 tests while S3P did it only for 3 tests.

StringFuzz suite. StringFuzz [30] is a fuzzer for automatically generating SMT-LIB string constraints. StringFuzz can help in exposing bugs and performance issues for string solvers. We use StringFuzz to generate 1025 tests including word (dis)equalities and regular membership constraints. These generated tests consist of a combination of small and large examples (in terms of the number of used variables and expected lengths of satisfying string assignments). TRAU can solve 984 tests (of them 723 tests are *sat* and 261 tests are *unsat*) in the suite. CVC4 and Z3-str3 can determine the satisfiability of only 778 and 795 tests, respectively. We do not run S3P and TRAU-PRE because they do not support some constraints in the suite. TRAU gives up in 41 tests containing non-membership constraints that are currently not supported.

ACKNOWLEDGEMENTS

This research has been partially supported by the Swedish Research Council (VR) under grant 2014-5484, by the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), the Czech Science Foundation project 16-24707Y, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), the FIT BUT internal project FIT-S-17-4014, and the Ministry of Science and Technology of Taiwan (project 106- 2221-E-001-009-MY3).

REFERENCES

[1] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A Solver for String Constraints," in *ISTA'09*. ACM, 2009, pp. 105–116.

[2] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *TACAS'10*, ser. LNCS, vol. 6015. Springer, 2010, pp. 154–157.

[3] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A Z3-based string solver for web application analysis," in *ESEC/FSE'13*. ACM, 2013, pp. 114–124.

[4] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "S3: A symbolic string solver for vulnerability detection in web applications," in *CCS'14*. ACM, 2014, pp. 1232–1243.

[5] 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*, ser. LNCS, vol. 8559. Springer, 2014, pp. 646–662.

[6] S. Kausler and E. Sherman, "Evaluation of string constraint solvers in the context of symbolic execution," in *ASE '14*. ACM, 2014, pp. 259–270.

[7] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "String constraints for verification," in *CAV'14*, ser. LNCS, vol. 8559. Springer, 2014, pp. 150–166.

[8] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, and P. Rümmer, "Flatten and conquer: a framework for efficient analysis of string constraints," in *PLDI*. ACM, 2017, pp. 602–617.

[9] A. W. Lin and P. Barceló, "String solving with word equations and transducers: towards a logic for analysing mutation XSS," in *POPL'16*. ACM, 2016, pp. 123–136.

[10] M. Berzish, Y. Zheng, and V. Ganesh, "Z3str3: A string solver with theory-aware branching," *CoRR*, vol. abs/1704.07935, 2017.

[11] L. Holík, P. Janku, A. W. Lin, P. Rümmer, and T. Vojnar, "String constraints with concatenation and transducers solved efficiently," *PACMPL*, vol. 2, no. POPL, pp. 4:1–4:32, 2018.

[12] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2010, pp. 513–528.

[13] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic discovery of client-side validation vulnerabilities in rich web applications," in *NDSS*. The Internet Society, 2010.

[14] F. Yu, M. Alkhalaf, and T. Bultan, "Stranger: An automata-based string analysis tool for PHP," in *TACAS*, ser. LNCS, J. Esparza and R. Majumdar, Eds., vol. 6015. Springer, 2010, pp. 154–157.

[15] J. D. Scott, P. Flener, J. Pearson, and C. Schulte, "Design and implementation of bounded-length sequence variables," in *CPAIOR*, ser. LNCS, D. Salvagnin and M. Lombardi, Eds., vol. 10335. Springer, 2017, pp. 51–67.

[16] J. D. Scott, P. Flener, and J. Pearson, "Constraint solving on bounded string variables," in *CPAIOR*, ser. LNCS, vol. 9075. Springer, 2015, pp. 375–392.

[17] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman, "Norn: An SMT solver for string constraints," in *CAV'15*, ser. LNCS, vol. 9206. Springer, 2015, pp. 462–469.

[18] H. Wang, T. Tsai, C. Lin, F. Yu, and J. R. Jiang, "String analysis via automata manipulation with logic circuit representation," in *CAV'16*, ser. LNCS, vol. 9779. Springer, 2016, pp. 241–260.

[19] J. van Leeuwen, "Effective constructions in well-partially- ordered free monoids," *Discrete Mathematics*, vol. 21, no. 3, pp. 237 – 252, 1978.

[20] M. F. Atig, A. Bouajjani, and T. Touili, "On the reachability analysis of acyclic networks of pushdown systems," in *CONCUR'08*, ser. LNCS, vol. 5201. Springer, 2008, pp. 356–371.

[21] R. Parikh, "On context-free languages," *J. ACM*, vol. 13, no. 4, 1966.

[22] J. Esparza, P. Ganty, S. Kiefer, and M. LuttenbergSer, "Parikh's theorem: A simple and direct automaton construction," *Inf. Process. Lett.*, vol. 111, no. 12, pp. 614–619, 2011.

[23] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS'08*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[24] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters, "CVC4," 2016. [Online]. Available: http://cvc4.cs.nyu.edu/papers/CAV2014-strings/

[25] M. Trinh, D. Chu, and J. Jaffar, "Progressive reasoning over recursively-defined strings," in *CAV'16*, ser. LNCS, vol. 9779. Springer, 2016, pp. 218–240.

[26] "Trau Solver." [Online]. Available: https://github.com/diepbp/Trau

[27] T. Tateishi, M. Pistoia, and O. Tripp, "Path- and index-sensitive string analysis based on monadic second-order logic," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, pp. 1–33, 2013.

[28] "IBM Security AppScan Tool and Source." [Online]. Available: https://www.ibm.com/us-en/marketplace/ibm-appscan-source.

[29] "Google Closure Library." [Online]. Available: https://github.com/google/closure-library/.

[30] "StringFuzz." [Online]. Available: https://github.com/dblotsky/stringfuzz