Automata-based Model Counting String Solver

Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan

I. Introduction

Most common vulnerabilities in Web applications are due to string manipulation errors in input validation and sanitization code. String constraint solvers are essential components of program analysis techniques for detecting and repairing vulnerabilities that are due to string manipulation errors. For quantitative and probabilistic program analyses [1], [2], [3], [4], checking the satisfiability of a constraint is not sufficient, and it is necessary to count the number of solutions.

In this paper, we present a constraint solver that, given a string constraint, 1) constructs an automaton that accepts all solutions that satisfy the constraint, 2) generates a function that, given a length bound, gives the total number of solutions within that bound [5]. Our approach relies on the observation that, using an automata-based constraint representation, model counting reduces to path counting, which can be solved precisely.

II. A Model Counting String Constraint Solver

We first describe our string constraint language, then we discuss how to construct automata for string constraints and how to count number of solutions for a string constraint.

A. String Constraint Language

We define the set of string constraints using the following abstract grammar:

$$
\begin{array}{lcll}
F & \rightarrow & C \mid \neg F \mid F \wedge F \mid F \vee F & (1)\\
C & \rightarrow & S \in R & (2)\\
 & \mid & S = S & (3)\\
 & \mid & S = S \,.\, S & (4)\\
 & \mid & \textsc{len}(S)\ O\ n & (5)\\
 & \mid & \textsc{len}(S)\ O\ \textsc{len}(S) & (6)\\
 & \mid & \textsc{contains}(S, s) & (7)\\
 & \mid & \textsc{begins}(S, s) & (8)\\
 & \mid & \textsc{ends}(S, s) & (9)\\
 & \mid & n = \textsc{indexof}(S, s) & (10)\\
 & \mid & n = \textsc{lastIndexof}(S, s) & (11)\\
 & \mid & S = \textsc{charAt}(S, n) & (12)\\
 & \mid & S = \textsc{substring}(S, n, n) & (13)\\
 & \mid & S = \textsc{replace}(S, s, s) & (14)\\
S & \rightarrow & v \mid s & (15)\\
R & \rightarrow & s \mid \varepsilon \mid R\ R \mid R \mid R \mid R^* & (16)\\
O & \rightarrow & < \mid = \mid > \mid + \mid - & (17)
\end{array}
$$

where C denotes the basic constraints, n denotes integer values, $s \in \Sigma^*$ denotes string values, ε is the empty string, v denotes string variables, . is the string concatenation operator, $\textsc{len}(v)$ denotes the length of the string value that is assigned to variable v. Semantics of the string functions are consisted with Java language semantics.

This material is based on research sponsored by NSF under grant CCF-1423623 and by DARPA under agreement number FA8750-15-2-0087.

Given a constraint F, let V_F denote the set of variables that appear in F. Let $F[s/v]$ denote the constraint that is obtained from F by replacing all appearances of $v \in V_F$ with the string constant s. We define the truth set of the constraint F for variable v as $[\![F, v]\!] = \{s \mid F[s/v] \text{ is satisfiable}\}$.

B. Mapping Constraints to Automata

A Deterministic Finite Automaton (DFA) A is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where $Q = \{1, 2, \ldots, n\}$ is the set of n states, Σ is the input alphabet, $\delta \subseteq Q \times Q \times \Sigma$ is the state transition relation set, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final, or accepting, states.

Given an automaton A, let $\mathcal{L}(A)$ denote the set of strings accepted by A. Given a constraint F and a variable v, our goal is to construct an automaton A, such that $\mathcal{L}(A) = [\![F, v]\!]$.

Let us define an automata constructor function $\mathcal{A}$ such that, given a constraint F and a variable v, $\mathcal{A}(F, v)$ is an automaton where $\mathcal{L}(\mathcal{A}(F, v)) = [\![F, v]\!]$. In this section we discuss how to implement the automata constructor function $\mathcal{A}$.

We assume that F is converted to disjunctive normal form (DNF) where $F \equiv \ \vee_{i=1}^{n} F_i$, $F_i \equiv \ \wedge_{j=1}^{m} C_{ij}$, and each C_{ij} is either a basic constraint or negation of a basic constraint.

In order to construct the automaton $\mathcal{A}(F, v)$ we first construct the automata $\mathcal{A}(F_i, v)$ for each F_i where $\mathcal{A}(F_i, v)$ accepts the language $[\![F_i, v]\!]$. Then we combine the $\mathcal{A}(F_i, v)$ automata using automata product such that $\mathcal{A}(F, v)$ accepts the language $[\![F_1, v]\!] \cup [\![F_2, v]\!] \cup \ldots \cup [\![F_m, v]\!]$.

Since we discussed how to handle disjunction, from now on we focus on constraints of the form $F \equiv C_1 \wedge C_2 \wedge \ldots \wedge C_n$ where each C_i is either a basic constraint or negation of a basic constraint. Since a basic constraint is also a constraint, we can use the same automata construct function for basic constraints. In order to construct $\mathcal{A}(F, v)$, we first construct the automata $\mathcal{A}(C_i, v)$ for each C_i where $\mathcal{A}(C_i, v)$ accepts the language $[\![C_i, v]\!]$. Then we combine the $\mathcal{A}(C_i, v)$ automata using automata product such that $\mathcal{A}(F, v)$ accepts the language $[\![C_1, v]\!] \cap [\![C_2, v]\!] \cap \ldots \cap [\![C_m, v]\!]$.

To describe automata construct function $\mathcal{A}(C, v)$ for basic constraints, consider the following string constraint $F \equiv \neg(x \in (01)^*) \ \wedge \ \textsc{len}(x) \geq 1$ over the alphabet $\Sigma = \{0, 1\}$. Let us name the basic constraints of F as $C_1 \equiv \neg(x \in (01)^*)$, $C_2 \equiv \textsc{len}(x) \geq 1$, where $F \equiv C_1 \ \wedge \ C_2$. Note that, since we push down all the negations in a DNF form, we treat negation of a basic constraint as a basic constraint. Let us define a term t to be a node at any level of tree rooted with a basic constraint C. As an example, Figure 1a shows terms of basic constraints C_1 and C_2. We extend the automata constructor function for terms such that $\mathcal{A}(C_i, t_j)$ constructs an automaton for each term $t \in C_i$. The automata construction algorithm traverses syntax tree (terms) in a post-order manner, and constructs the automata for each term using

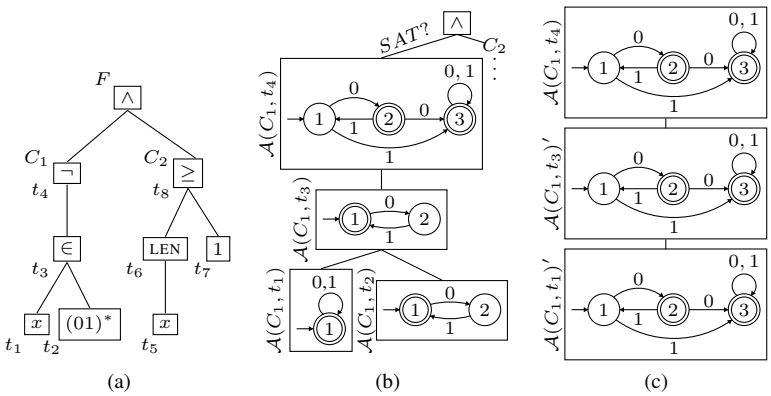

Figure 1: (a) The syntax tree for the string constraint $\neg(x \in (01)^*) \ \wedge \ \text{LEN}(x) \geq 1$ (b) the automata construction that traverses the syntax tree from the leaves towards the root (c) pre-image computation to construct automaton for the variable.

the automata constructed for its children. A term for a variable is initialized with the latest automaton computed for the variable (initially all variables are initialized with $\mathcal{A}(\Sigma^*)$, i.e., initially all variables are unconstrained) and a term for literal is initialized with automaton A where $L(A)$ is the literal value. Figure 1b demonstrates the automata construction algorithm on our running example. An automaton for t_1 is constructed from the latest value of the variable x and an automaton for t_2 is constructed based on regular expression literal. An automaton for t_3 is computed by using the automata for t_1 and t_2 with the semantics of the operation $\in$. Automata construction algorithm does a satisfiability check whenever a term that is root of a basic constraint is reached. In Figure 1b, we stop at term t_4 and check satisfiability of C_1 by checking if $L(\mathcal{A}(C_1, t_4))$ is not $\emptyset$. If the constraint is unsatisfiable, the algorithm sets $\mathcal{A}(C, v)$ (which is $\mathcal{A}(C_1, v)$ in our example) to $\mathcal{A}(\emptyset)$. If the constraint is satisfiable, we update the value of variable v by doing pre-image computations on the path from the root term of the basic constraint to a term that corresponds to a variable using the pre-image computations discussed in [6]. In Figure 1c, we start with $\mathcal{A}(C_1, t_4)$ and continue to calculate automata for the terms on the path to variable x ($t_4 \rightarrow t_3 \rightarrow t_1$). Finally, $\mathcal{A}(C_1, v)$ returns the automaton generated by $\mathcal{A}(C_1, t_1)'$ as new automaton for v.

A constraint F may have more than one variable where V_F denotes the set of variables that appear in F. In that case, we use the same algorithm describe above to construct the automata for each variable $v \in V_F$. If there are two variables appear in the same basic constraint, we do a pre-image computation for each of them. In a multi-variable constraint, for each variable v, we would get an over-approximation of the truth-set $\mathcal{A}(F, v) \supseteq [\![F, v]\!]$. We can eliminate over-approximation by solving the constraint iteratively. At each iteration, we initialize each $\mathcal{A}(F, v)$ to automaton that is obtain in previous iteration for the same v. We stop iteration when there is no more change in any $\mathcal{A}(F, v)$. Note that, using multiple variables, one can specify constraints with non-regular truth sets. For example, given the constraint $F \equiv x = y \ . \ y$, $[\![F, x]\!]$ is not a regular set, so we cannot construct an automaton precisely recognizing its truth set. In that case, we put a bound on the number of iterations for constraint solver and return an over-approximation of the truth set when bound is reached.

C. Automata-based Model Counting

Once we have translated a set of constraints into an automaton we reduce the model counting problem into path counting problem of graphs. We solve path counting problem by deriving a symbolic function that given a length bound k outputs the number of solutions within bound k. To achieve this, we use the *transfer matrix method* [7], [8] to produce an ordinary generating function which in turn yields a linear recurrence relation that is used to count constraint solutions.

III. Experiments and Applications

We implemented this approach as tool called ABC and conducted experiments to evaluate its effectiveness [5]. We used constraints generated from real world Java (116164 constraints) and JavaScript (44252 constraints) applications. Our experiments show that ABC solves wide range of constraints effectively.

To test effectiveness of our model counting approach, we do a comparison with another string model counter called SMC [9]. We used 6 examples that are listed on SMCs web page. For all the cases ABC generates a precise count given the bound whereas SMC generates an upper bound and a lower bound. ABCs count is exactly equal to SMCs upper bound for four of the examples and is exactly equal to SMCs lower bound for one example. For one example ABC reports a count that is between the lower and upper bound produced by SMC.

ABC is integrated with Symbolic Java Path Finder (a.k.a SPF) [10] and used in worst case complexity analysis and information leakage analysis with SPF.

References

[1] D. Clark, S. Hunt, and P. Malacaria, "A static analysis for quantifying information flow in a simple imperative language," *J. Comput. Secur.*, vol. 15, no. 3, 2007.

[2] G. Smith, "On the foundations of quantitative information flow," in *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures (FOSSACS)*, 2009, pp. 288–302.

[3] A. Filieri, C. S. Pasareanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 622–631.

[4] M. Borges, A. Filieri, M. d'Amorim, C. S. Pasareanu, and W. Visser, "Compositional solution space quantification for probabilistic software analysis," in *Proceedigns of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[5] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *CAV*, 2015, pp. 255–272.

[6] F. Yu, "Automatic verification of string manipulating programs," Ph.D. dissertation, University of California, Santa Barbara, 2010.

[7] R. P. Stanley, *Enumerative Combinatorics: Volume 1*, 2nd ed. New York, NY, USA: Cambridge University Press, 2011.

[8] P. Flajolet and R. Sedgewick, *Analytic Combinatorics*, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[9] L. Luu, S. Shinde, P. Saxena, and B. Demsky, "A model counter for constraints over unbounded strings," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, p. 57.

[10] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," ser. ISSTA '08. ACM, 2008, pp. 15–26.