Automata-based Model Counting for String Constraints

UCSB Technical Report Number 2015-03

Abdulbaki Aydin, Lucas Bang, Tevfik Bultan

University of California, Santa Barbara

Abstract. 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, 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. Our approach relies on the observation that, using an automata-based constraint representation, model counting reduces to path counting, which can be solved precisely. We demonstrate the effectiveness of our approach on a large set of string constraints extracted from real-world web applications.

1 Introduction

Since many computer security vulnerabilities are due to errors in string manipulating code, string analysis has become an active research area in the last decade [9, 40, 37, 17, 32, 3, 13, 39]. Symbolic execution is a well-known automated bug detection technique which has been applied to vulnerability detection [28]. In order to apply symbolic execution to analysis of string manipulating programs, it is necessary to check satisfiability of string constraints [6]. Several string constraint solvers have been proposed in recent years to address this problem [18, 21, 19, 41, 23, 1, 24, 33].

There are two recent research directions that aim to extend symbolic execution beyond assertion checking. One of them is quantitative information flow, where the goal is to determine how much secret information is leaked from a given program [10, 30, 27], and another one is probabilistic symbolic execution where the goal is to compute probability of the success and failure paths in order to establish reliability of the given program [14, 7]. Interestingly, both these approaches require the same basic extension to constraint solving: They require a model-counting constraint solver that not only determines if a constraint is satisfiable, but it also computes the number of satisfying instances.

In this paper, we present an automata-based model-counting technique for string constraints that consists of two main steps: 1) Given a string constraint and a variable, we construct an automaton that accepts all the string values for that variable for which the string constraint is satisfiable. 2) Given an automaton we generate a function that takes a length bound as input and returns the total number of strings that are accepted by the automaton that have a length that is less than or equal to the given bound.

Our constraint language can handle regular language membership queries, word equations that involve concatenation and replacement, and arithmetic constraints on string lengths. For a class of constraints that we call *pseudo-relational*, our approach gives the precise model-count. For constraints that are not in this class our approach computes an upper bound. We implemented a model-counting constraint solver based on the approach we present in this paper, and our experiments demonstrate that it is effective and efficient when applied to thousands of string constraints extracted from real-world web applications.

Related Work: Our inspiration for this work was the recently proposed model-counting string constraint solver SMC [25]. Similar to SMC, we also utilize generating functions in model-counting. However, due to some significant differences in how we utilize generating functions, our approach is strictly more precise than the approach used in SMC. For example, SMC cannot determine the precise model count for a regular expression constraint such as $x \in (a|b)^*|ab$, whereas our approach is precise for all regular expressions. More importantly, SMC cannot propagate string values across logical connectives which reduces its precision. For example, for a simple constraint such as $(x \in a|b) \ \vee \ (x \in a|b|c|d)$ SMC will generate a model-count range which consists of an upper bound of 6 and a lower bound of 2, whereas our approach will generate the exact count which is 4. Moreover, SMC always generates a lower bound of 0 for conjunctions that involve the same variable. So, the range generated for $(x \in a|b) \ \wedge \ (x \in a|b|c|d)$ would be 0 to 2, whereas our approach generates the exact count which is 2. The set of constraints we handle is also larger than the constraints that SMC can handle. In particular, we can handle constraints with replace operation which is common in server-side input sanitization code.

There has been significant amount of work on string constraint solving in recent years [18, 21, 19, 28, 41, 23, 1, 24, 33]. Some of these constraints solvers bound the string length [21, 28, 23] whereas our approach handles strings of arbitrary length. None of these string constraint solvers provide model-counting functionality. Our model-counting constraint solver is based on the automata-based string analysis tool Stranger [40, 37, 39], which was determined to be the best in terms of precision and efficiency in a recent empirical study for evaluating string constraint solvers for symbolic execution of Java programs [20]. In addition to checking satisfiability, our constraint solver also generates an automaton that accepts all possible solutions and provides model-counting capability. To the best of our knowledge, our string constraint solver is the only tool that supports all these. In addition to enabling quantitative and probabilistic analysis by model counting, our constraint solver also enables automated program repair synthesis by generating a characterization of all solutions [38, 2].

2 Automata Construction for String Constraints

In this section we discuss how to construct automata for string constraints. Given a constraint and a variable, our goal is to construct an automaton that accepts all strings, which, when assigned as the value of the variable in the given constraint, results in a satisfiable constraint.

2.1 String Constraint Language

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

$$F \rightarrow C \mid \neg F \mid F \ \wedge \ F \mid F \ \vee \ F \tag{1}$$
$$C \rightarrow S \in R \tag{2}$$
$$\mid \ S = S \tag{3}$$
$$\mid \ S = S \ . \ S \tag{4}$$
$$\mid \ \textsc{len}(S) \ O \ n \tag{5}$$
$$\mid \ \textsc{len}(S) \ O \ \textsc{len}(S) \tag{6}$$
$$\mid \ \textsc{contains}(S, s) \tag{7}$$
$$\mid \ \textsc{begins}(S, s) \tag{8}$$
$$\mid \ \textsc{ends}(S, s) \tag{9}$$
$$\mid \ n = \textsc{indexof}(S, s) \tag{10}$$
$$\mid \ S = \textsc{replace}(S, s, s) \tag{11}$$
$$S \rightarrow v \mid s \tag{12}$$
$$R \rightarrow s \mid \varepsilon \mid R \ R \mid R \mid R \mid R^* \tag{13}$$
$$O \rightarrow < \ \mid \ = \ \mid \ > \tag{14}$$

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, and the string functions are defined as follows:

- $\textsc{contains}(v, s) \Leftrightarrow \exists s_1, s_2 \in \Sigma^* : v = s_1 s s_2$
- $\textsc{begins}(v, s) \Leftrightarrow \exists s_1 \in \Sigma^* : v = s s_1$
- $\textsc{ends}(v, s) \Leftrightarrow \exists s_1 \in \Sigma^* : v = s_1 s$
- $n = \textsc{indexof}(v, s) \Leftrightarrow (\exists s_1, s_2 \in \Sigma^* : \textsc{len}(s_1) = n \ \wedge \ v = s_1 s s_2) \ \wedge$ $(\forall i < n : \neg(\exists s_1, s_2 \in \Sigma^* : \textsc{len}(s_1) = i \ \wedge \ v = s_1 s s_2))$
- $v = \textsc{replace}(v', s_1, s_2) \Leftrightarrow (\textsc{contains}(v', s_1) \ \wedge$ $(\exists s_3, s_4, s_5 \in \Sigma^* : v' = s_3 s_1 s_4 \ \wedge \ v = s_3 s_2 s_5 \ \wedge \ s_5 = \textsc{replace}(s_4, s_1, s_2) \ \wedge$ $(\forall s_6, s_7 \in \Sigma^* : v' = s_6 s_1 s_7 \Rightarrow \textsc{len}(s_6) \geq \textsc{len}(s_3)))) \ \vee$ $(\neg\textsc{contains}(v', s_1) \ \wedge \ v = v')$

and the definitions of these functions when the string variable v is replaced with a string constant are similar.

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 formula F for variable v as $[\![F, v]\!] = \{s \mid F[s/v] \text{ is satisfiable}\}$.

We identify three classes of constraints: 1) *Single-variable constraints* are constructed using at most one string variable (i.e., $V_F = \{v\}$ or $V_F = \emptyset$), they do not contain constraints of type (4), (6), and (11), and have a single variable on the left hand side of constraints of type (3). 2) *Pseudo-relational constraints:* are a set of constraints that we define in the next section, for which the truth sets are regular (i.e., each $[\![F, v]\!]$ is a regular set). 3) *Relational constraints* are the constraints that are not pseudo-relational constraints (truth sets of relational constraints can be non-regular).

2.2 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 $\mathcal{L}(A) = [\![F, v]\!]$.

Automata Construction for Single-Variable Constraints: Let us define an automata constructor function $\mathcal{A}$ such that, given a formula 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 $\mathcal{A}$.

Let us give an example to demonstrate the automata construction algorithm for single-variable 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 sub-constraints of F as $C_1 \equiv x \in (01)^*$, $C_2 \equiv \textsc{len}(x) \geq 1$, $F_1 \equiv \neg C_1$, where $F \equiv F_1 \ \wedge \ C_2$. The automata construction algorithm starts from the basic constraints at the leaves of the syntax tree (C_1 and C_2), and constructs the automata for them. Then it traverses the syntax tree towards the root by constructing an automaton for each node using the automata constructed for its children (where the automaton for F_1 is constructed using the automaton for C_1 and the automaton for F is constructed using the automata for F_1 and C_2). Figure 1 demonstrates the automata construction algorithm on our running example.

Let $\mathcal{A}(\Sigma^*), \mathcal{A}(\Sigma^n), \mathcal{A}(s)$, and $\mathcal{A}(\emptyset)$ denote automata that accept the languages Σ^*, Σ^n, $\{s\}$, and $\emptyset$, respectively. We describe the construction of the automaton $\mathcal{A}(F, v)$ recursively on the structure of the single-variable constraint F as follows:

- case $V_F = \emptyset$ (i.e., there are no variables in F): Evaluate the constraint F. If $F \equiv \textbf{true}$ then $\mathcal{A}(F, v) = \mathcal{A}(\Sigma^*)$, otherwise $\mathcal{A}(F, v) = \mathcal{A}(\emptyset)$.
- case $F \equiv \neg F_1$: $\mathcal{A}(F, v)$ is constructed using $\mathcal{A}(F_1, v)$ and it is an automaton that accepts the complement language $\Sigma^* - \mathcal{L}(\mathcal{A}(F_1, v))$.
- case $F \equiv F_1 \ \wedge \ F_2$ or $F \equiv F_1 \ \vee \ F_2$: $\mathcal{A}(F, v)$ is constructed using $\mathcal{A}(F_1, v)$ and $\mathcal{A}(F_2, v)$ using automata product, and it accepts the language $\mathcal{A}(F_1, v) \cap \mathcal{A}(F_2, v)$ or $\mathcal{A}(F_1, v) \cup \mathcal{A}(F_2, v)$, respectively.
- case $F \equiv v \in R$: $\mathcal{A}(F, v)$ is constructed using regular expression to automata conversion algorithm and accepts all strings that match the regular expression R.
- case $F \equiv v = s$: $\mathcal{A}(F, v) = \mathcal{A}(s)$.
- case $F \equiv \textsc{len}(v) = n$: $\mathcal{A}(F, v) = \mathcal{A}(\Sigma^n)$.

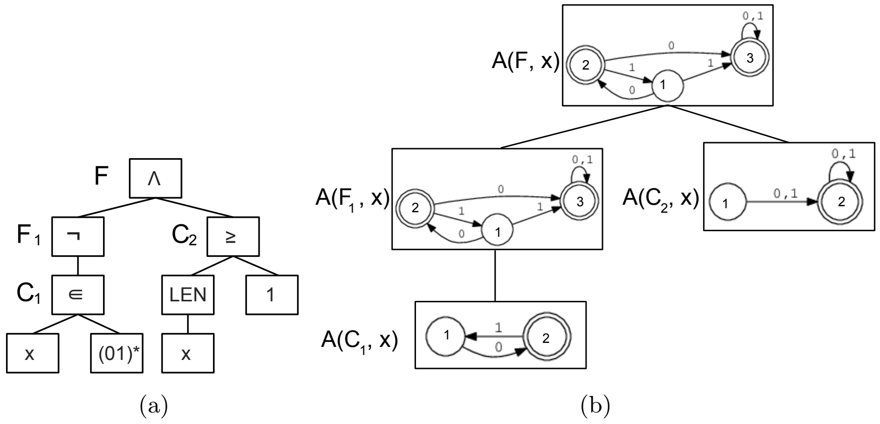


Fig. 1. (a) The syntax tree for the string constraint $\neg(x \in (01)^*) \ \wedge$ LEN$(x) \geq 1$ and (b) the automata construction that traverses the syntax tree from the leaves towards the root.

- case $F \equiv$ LEN$(v) < n$: $\mathcal{A}(F, v)$ is an automaton that accepts the language $\{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^{n-1}$.
- case $F \equiv$ LEN$(v) > n$: $\mathcal{A}(F, v)$ is constructed using $\mathcal{A}(\Sigma^{n+1})$ and $\mathcal{A}(\Sigma^*)$ and then constructing an automaton that accepts the concatenation of those languages, i.e., $\Sigma^{n+1}\Sigma^*$.
- case $F \equiv$ CONTAINS(v, s): $\mathcal{A}(F, v)$ is an automaton that is constructed using $\mathcal{A}(\Sigma^*)$ and $\mathcal{A}(s)$ and it accepts the language $\Sigma^* s \Sigma^*$.
- case $F \equiv$ BEGINS(v, s): $\mathcal{A}(F, v)$ is constructed using $\mathcal{A}(\Sigma^*)$ and $\mathcal{A}(s)$, and it accepts the language $s\Sigma^*$.
- case $F \equiv$ ENDS(v, s): $\mathcal{A}(F, v)$ is constructed using $\mathcal{A}(\Sigma^*)$ and $\mathcal{A}(s)$, and it accepts the language $\Sigma^* s$.
- case $F \equiv n =$ INDEXOF(v, s): Let L_i denote the language $\Sigma^i s \Sigma^*$. Automata that accept the languages L_i can be constructed using $\mathcal{A}(\Sigma^i)$, $\mathcal{A}(s)$, and $\mathcal{A}(\Sigma)$. Then $\mathcal{A}(F, v)$ is the automaton that accepts the language $\Sigma^n s \Sigma^* - (\{\varepsilon\} \cup L_1 \cup L_2 \cup \ldots \cup L_{n-1})$ which can be constructed using $\mathcal{A}(\Sigma^n)$, $\mathcal{A}(s)$, $\mathcal{A}(\Sigma)$, and the automata that accept L_i.

Pseudo-Relational Constraints: Pseudo-relational constraints are multi-variable constraints. 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. Below, we define a class of constraints called pseudo-relational constraints for which $[\![F, v]\!]$ is regular.

We assume that constraint F is converted to DNF form 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. The constraint F is pseudo-relational if each F_i is pseudo-relational.

Given $F \equiv C_1 \ \wedge \ C_2 \ \wedge \ \ldots \ \wedge \ C_n$, where each C_i is either a basic constraint or negation of a basic constraint, for each C_i, let V_{C_i} denote the set of variables that appear in C_i. We call F pseudo-relational if the following conditions hold:

1. Each variable $v \in V_F$ appears in each C_i at most once.
2. There is only one variable, $v \in V_F$, that appears in more than one constraint C_i where $v \in V_{C_i} \ \wedge \ |V_{C_i}| > 1$, and in each C_i that v appears in, v is on the left hand side of the constraint. We call v the *projection variable.*
3. For all variables $v' \in V_F$ other than the projection variable, there is a single constraint C_i where $v' \in V_{C_i} \ \wedge \ |V_{C_I}| > 1$.
4. All constraints C_i where $|V_{C_i}| > 1$, C_i is not negated in the formula F.

Many string constraints extracted from programs via symbolic execution are pseudo-relational constraints, or can be converted to pseudo-relational constraints. The projection variable represents either the variable that holds the value of the user's input to the program (for example, user input to a web application that needs to be validated), or the value of the string expression at a program sink. A program sink is a program point (such as a security sensitive function) for which it is necessary to compute the set of values that reach to that program point in order to check for vulnerabilities.

For example, following constraint is a pseudo-relational constraint extracted from a web application (regular expressions are simplified):

$$(x = y \ . \ z) \ \wedge \ (\textsc{len}(y) = 0) \ \wedge \ \neg(z \in (0|1)) \ \wedge \ (x = t) \ \wedge \ \neg(t \in 0^*)$$

Automata Construction for Pseudo-Relational Constraints: Given a pseudo-relational constraint F and the projection variable v, we now discuss how to construct the automaton $\mathcal{A}(F, v)$ that accepts $[\![F, v]\!]$.

As above we assume that F is converted to DNF form 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. For each C_i, let V_{C_i} denote the set of variables that appear in C_i. If V_{C_i} is a singleton set, then we refer to the variable in it as v_{C_i}.

First, for each single-variable constraint C_i that is not negated, we construct an automaton that accepts the truth set of the constraint C_i, $[\![C_i, v_{C_i}]\!]$, using the techniques we discussed above for single-variable constraints. If C_i is negated, then we construct the automaton that accepts the complement language $\Sigma^* - [\![C_i, v_{C_i}]\!]$. Let us call these automata $\mathcal{A}(C_i, v_{C_i})$ (some of which may correspond to negated constraints).

Then, for any variable $v' \in V_F$ that is not the projection variable, we construct an automaton $\mathcal{A}(F, v')$ which accepts the intersection of the languages $\mathcal{A}(C_i, v')$ for all single-variable constraints that v' appears in, i.e., $\mathcal{L}(\mathcal{A}(F, v')) = \bigcap_{V_{C_i} = \{v'\}} \mathcal{L}(\mathcal{A}(C_i, v'))$.

Next, for each multi-variable constraint C_i we construct an automaton that accepts the language $[\![C_i, v]\!]$ where v is the projection variable as follows:

- case $C_i \equiv v = v'$: $\mathcal{A}(C_i, v) = \mathcal{A}(F, v')$.
- case $C_i \equiv v = v_1 \ . \ v_2$: $\mathcal{A}(C_i, v)$ is constructed using the automata $\mathcal{A}(F, v_1)$ and $\mathcal{A}(F, v_2)$ and it accepts the concatenation of the languages $\mathcal{L}(\mathcal{A}(F, v_1))$ and $\mathcal{L}(\mathcal{A}(F, v_2))$.
- case $C_i \equiv \textsc{len}(v) = \textsc{len}(v')$: Given the automaton $\mathcal{A}(F, v')$, we construct an automaton $A_{\textsc{len}(F,v')}$ such that $s \in \mathcal{L}(A_{\textsc{len}(F,v')}) \Leftrightarrow \exists s' : \textsc{len}(s) = \textsc{len}(s') \wedge s' \in \mathcal{L}(\mathcal{A}(F, v'))$. Then, $\mathcal{A}(C_i, v) = A_{\textsc{len}(F,v')}$.
- case $C_i \equiv \textsc{len}(v) < \textsc{len}(v')$: Given the automaton $\mathcal{A}(F, v')$ we find the length of the maximum word accepted by $\mathcal{A}(F, v')$, which is infinite if $\mathcal{A}(F, v')$ has a loop that can reach an accepting state. If it is infinite then $\mathcal{A}(C_i, v) = A(\Sigma^*)$. If not, then given the maximum length m, $\mathcal{A}(C_i, v)$ is the automaton that accepts the language $\{\varepsilon\} \cup \Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^{m-1}$. Note that if $m = 0$ then $\mathcal{A}(C_i, v) = A(\emptyset)$.
- case $C_i \equiv \textsc{len}(v) > \textsc{len}(v')$: Given the automaton $\mathcal{A}(F, v')$ we find the length of the minimum word accepted by $\mathcal{A}(F, v')$. Given the minimum length m, $\mathcal{A}(C_i, v)$ is the automaton that accepts the concatenation of the languages accepted by $\mathcal{A}(\Sigma^{m+1})$ and $\mathcal{A}(\Sigma^*)$, i.e, $\Sigma^{m+1}\Sigma^*$.
- case $C_i \equiv v = \textsc{replace}(v', s, s)$: Given the the automaton $\mathcal{A}(F, v')$ we use the construction presented in [40, 39] for language based replacement to construct the automaton $\mathcal{A}(C_i, v)$.

The final step of the construction is to construct $\mathcal{A}(F, v)$ using the automata $\mathcal{A}(C_i, v)$ where $\mathcal{L}(\mathcal{A}(F, v)) = \bigcap_{v \in V_{C_i}} \mathcal{L}(\mathcal{A}(C_i, v))$.

For pseudo-relational constraints the automaton $\mathcal{A}(F, v))$ constructed based on the above construction accepts the truth set of the formula F for the projected variable, i.e., $\mathcal{L}(\mathcal{A}(F, v)) = [\![F, v]\!]$. However, the REPLACE function has different variations in different programming languages (such as first-match versus longest-match replace) and the match pattern can be given as a regular expression. The language-based replace automata construction we use [40, 39] over-approximates the replace operation in some cases, which would then result in over-approximation of the truth set: $\mathcal{L}(\mathcal{A}(F, v)) \supseteq [\![F, v]\!]$.

Automata Construction for Relational Constraints: For constraints that are not pseudo-relational, we can extend the above algorithm to compute an over approximation of $[\![F, v]\!]$. In relational constraints more than one variable can be involved in multi-variable constraints which creates a cycle in constraint evaluation.

Given a relational constraint in the form $F \equiv C_1 \ \wedge \ C_2 \ \wedge \ \ldots \ \wedge \ C_n$, we start with initializing each $\mathcal{A}(F, v)$ to $\mathcal{A}(\Sigma^*)$, i.e., initially variables are unconstrained. Then, we process each constraint as we described above to compute new automata for the variables in that constraint using the automata that are already available for each variable. We can stop this process at any time, and, for each variable v, we would get an over-approximation of the truth-set $\mathcal{A}(F, v) \supseteq [\![F, v]\!]$. We can state this algorithm as follows:

In order to improve the efficiency of the above algorithm, we first build a constraint dependency graph where, 1) a multi-variable constraint C_i depends on a single variable constraint C_j if $V_{C_j} \subseteq V_{C_i}$, and 2) a multi-variable constraint C_i depends on a multi-variable constraint C_j if $V_{C_j} \cap V_{C_i} \neq \emptyset$. We traverse the constraints based on their ordering in the dependency graph and iteratively refine

Algorithm 1 AutomataForConstraint($F \equiv C_1 \ \wedge \ C_2 \ \wedge \ \ldots \ \wedge \ C_n$)

```
for $v \in V_F$ do
    $\mathcal{A}(F, v) = \mathcal{A}(\Sigma^*)$;
end for
$count = 0$; $done =$ false;
while $count < bound \ \wedge \ \neg done$ do
    for each $C \in F$ and $v \in V_C$ do
        construct $A'$ where $\mathcal{L}(A') = \mathcal{L}(\mathcal{A}(F, v)) \cap \mathcal{L}(\mathcal{A}(C, v))$;
        $\mathcal{L}(\mathcal{A}(F, v)) = A'$;
    end for
    if none of the $\mathcal{L}(\mathcal{A}(F, v))$ changed during the current iteration of the while loop then
        $done =$ true;
    end if
    $count = count + 1$;
end while
```

the automata in case of cyclic dependencies. Note that, in the constructions we described above we only constructed automaton for the variable on the left-hand-side of a relational constraint using the automata for the variables on the right-hand-side of the constraint. In the general case we need to construct automata for variables on the right-hand-side of the relational constraints too. We do this using techniques similar to the ones we described above. Constructing automata for the right-hand-side variables is equivalent to the pre-image computations used during backward symbolic analysis as discussed in [36] and we use the constructions given there.

3 Automata-based Model Counting

Once we have translated a set of constraints into an automaton we can employ algebraic graph theory [5] and analytic combinatorics to perform model counting. In our method, model counting corresponds exactly to counting the accepting paths of the constraint DFA up to a given length bound k. This problem can be solved using dynamic programming techniques in $O(k \cdot |\delta|)$ where δ is the DFA transition relation [11, 16]. However, for each different bound, the dynamic programming technique requires another traversal of the DFA graph.

A preferable solution is to derive 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* [31, 15] to produce an ordinary generating function which in turn yields a linear recurrence relation that is used to count constraint solutions. We will briefly review the necessary background first and then describe the model counting algorithm.

Given a DFA A, consider its corresponding language $\mathcal{L}$. let $\mathcal{L}_i = \{w \in \mathcal{L} : |w| = i\}$, the language of strings in $\mathcal{L}$ with length i. We can then write $\mathcal{L} = \bigcup_{i \geq 0} \mathcal{L}_i$. Further, define $|\mathcal{L}_i|$ to be the cardinality of $\mathcal{L}_i$. Thus, the cardinality of $\mathcal{L}$ can be computed by the sum of a series $a_0, a_1, \ldots, a_i, \ldots$ where each a_i is the cardinality of the corresponding language $\mathcal{L}_i$, i.e., $a_i = |\mathcal{L}_i|$.

For example, recall the automaton in Fig. 1. Let $\mathcal{L}^x$ be the language over $\Sigma = \{0, 1\}$ that satisfies the formula $(x \notin (01)^* \wedge \mathrm{LEN}(x) \geq 1)$. Then $\mathcal{L}^x$ can be described by the expression $\Sigma^* - (01)^*$. In the language $\mathcal{L}^x$, we have zero strings of length 0 ($\varepsilon \notin \mathcal{L}^x$), two strings of length 1 ($\{0, 1\}$), three strings of length 3 ($\{00, 10, 11\}$), and so on. So in this case, adding a few more terms, our sequence becomes $a_0 = 0, a_1 = 2, a_2 = 3, a_3 = 8, a_4 = 15$, etc. It turns out that for any length i, the number of strings in $\mathcal{L}^x_i$, is given by a fairly simple 3^{rd} order linear recurrence relation:

$$\begin{array}{l} a_0 = 0, a_1 = 2, a_2 = 3 \\ a_i = 2a_{i-1} + a_{i-2} - 2a_{i-3} \text{ for } i \geq 3 \end{array} \tag{15}$$

In fact, using standard techniques for solving linear homogeneous recurrences, we can derive a closed form solution to determine that

$$|\mathcal{L}^x_i| = (1/2)(2^{i+1} + (-1)^{i+1} - 1). \tag{16}$$

In the following discussion we give a general method based on generating functions for deriving a recurrence relation and closed form solution that we can use for model counting.

Generating Functions: Given the representation of the size of a language $\mathcal{L}$ as a sequence $\{a_i\}$ we can encode each $|\mathcal{L}_i|$ as the coefficients of a polynomial, an ordinary generating function (GF). The *ordinary generating function* of the sequence $a_0, a_1, \ldots, a_i, \ldots$ is the infinite polynomial [31, 15]

$$g(z) = \sum_{i \geq 0} a_i z^i \tag{17}$$

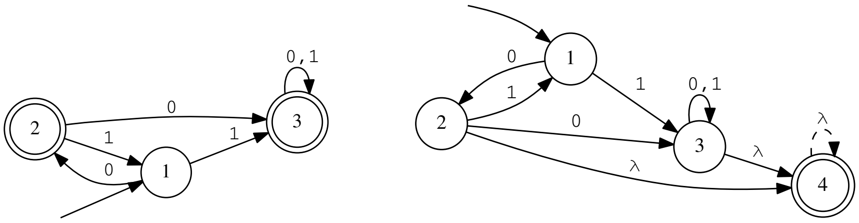
Although $g(z)$ is an infinite polynomial, in our application $g(z)$ can be interpreted as the Taylor series of some finite rational expression. I.e., we can also write $g(z) = p(z)/q(z)$, where $p(z)$ and $q(z)$ are finite degree polynomials. If $g(z)$ is given as such a finite rational expression, each a_i can be computed from the Taylor series expansion of $g(z)$:

$$a_i = \frac{g^{(i)}(0)}{i!} \tag{18}$$

where $g^{(i)}(z)$ is the i^{th} derivative of $g(z)$. We write $[z^i]g(z)$ for the i^{th} Taylor series coefficient of $g(z)$. Returning to our example, we can write the generating function for $\mathcal{L}^x$ both as a rational function and as an infinite Taylor series polynomial. The reader can verify the following equivalence by computing the RHS coefficients via equation (18).

$$g(z) = \frac{2z - z^2}{1 - 2z - z^2 + 2t^3} = 0z^0 + 2z^1 + 3z^2 + 8z^3 + 15z^4 + \ldots \tag{19}$$

Generating Function for a DFA: Given a DFA A and length k we can compute the generating function $g_A(z)$ such that the k^{th} Taylor series coefficient of $g_A(z)$ is equal to $|\mathcal{L}_k(A)|$.



(a)

(b)

Fig. 2. (a) The original DFA A, and (b) the augmented DFA A' used for model counting.

The transfer-matrix method [31, 15] allows us to compute a generating function for the number of accepting paths for an arbitrary DFA. We first apply a transformation and add an extra state, s_{n+1}. The resulting automaton is a DFA A' with λ-transitions from each of the accepting states of A to s_{n+1} where λ is a new padding symbol that is not in the alphabet of A. Thus, $\mathcal{L}(A') = \mathcal{L}(A) \cdot \lambda$ and furthermore $|\mathcal{L}_i(A)| = |\mathcal{L}_{i+1}(A')|$. That is, the augmented DFA A' preserves both the language and count information of A. Recalling the automaton from Fig. 1, the corresponding augmented DFA is shown in Fig. 2(b). (Ignore the dashed λ transition for the time being.)

From A' we construct the $(n+1) \times (n+1)$ transfer matrix T in the following way. Now, A' has $n+1$ states $s_1, s_2, \ldots s_{n+1}$. The matrix entry $T_{i,j}$ is the number of transitions from state s_i to state s_j. Then the generating function for A is

$$g_A(z) = (-1)^{n+1} \frac{\det(I - zT : n+1, 1)}{z \det(I - zT)}, \tag{20}$$

where $(M : i, j)$ denotes the matrix obtained by removing the i^{th} row and j^{th} column from M, I is the identity matrix, and $\det M$ is the matrix determinant. The number of accepting paths of A with length exactly k, i.e. $|\mathcal{L}_k(A)|$, is then given by $[z^k]g_A(z)$ which can be computed through symbolic differentiation via equation 18.

Applying this to our running example, we show the transition matrix T, and the terms $(I - zT)$ and $(I - zT : n, 1)$. In the example, $T_{1,2}$ is 1 because there is a single transition from state 1 to state 2, $T_{3,3}$ is 2 because there are two transitions from state 3 to itself, $T_{2,4}$ is 1 because there is a single (λ) transition from state 2 to state 4, and so on for the remaining entries.

$$T = \begin{bmatrix} 0&1&1&0 \\ 1&0&1&1 \\ 0&0&2&1 \\ 0&0&0&1 \end{bmatrix}, I - zT = \begin{bmatrix} 1 & -z & -z & 0 \\ -z & 1 & -z & -z \\ 0 & 0 & 1-2z & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}, (I - zT : n, 1) = \begin{bmatrix} -z & -z & 0 \\ 1 & -z & -z \\ 0 & 1-2z & -z \end{bmatrix}$$

Applying equation (20) results in the generating function

$$g_{A'}(z) = -\frac{\det(I - zT : n, 1)}{z \det(I - zT)} = \frac{2z - z^2}{1 - 2z - z^2 + 2z^3}. \tag{21}$$

This is precisely the same GF that counts $\mathcal{L}_i(A)$ given in equation (19). Suppose we now want to know the number of solutions of length six. We can compute the sixth Taylor series coefficient to find that $\mathcal{L}_6^x(A) = [z^6]g(z) = 63$.

Deriving a Recurrence Relation: We would like a way to compute the Taylor series coefficients that is more direct than symbolic differentiation. Here we describe how a linear recurrence for these coefficients can be extracted from the GF. Before we describe how to accomplish this in general, we first demonstrate the procedure for our specific example. Combining equations (20) and (21) and multiplying through by the denominator, we can write

$$2z - z^2 = (1 - 2z - z^2 + 2z^3)\sum_{i\geq 0} a_i z^i.$$

Expanding the sum for $0 \leq i < 3$ and collecting terms,

$$2z - z^2 = a_0 + (a_1 - 2a_0)z + (a_2 - 2a_1 - a_0)z^2 + \sum_{i\geq 3}(a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3})z^i.$$

Now by comparing the coefficient of z^i on the LHS to the coefficient of z^i on the RHS, we have the set of equations

$$\begin{aligned}
&a_0 = 0\\
&a_1 - 2a_0 = 2\\
&a_2 - 2a_1 - a_0 = -1\\
&a_i - 2a_{i-1} - a_{i-2} + 2a_{i-3} = 0, \text{ for } i \geq 3
\end{aligned}$$

One can see that this results in the same solution given in equation (15).

This idea is easily generalized. Recall that $g(z) = p(z)/q(z)$ for some finite degree polynomials p and q. Suppose that the maximum degree of p and q is m. With possibly some $b_i = 0$ and $c_i = 0$, we can write

$$g(z) = \frac{b_m z^m + \ldots + b_1 z + b_0}{c_m z^m + \ldots + c_1 z + c_0} = \sum_{i\geq 0} a_i z^i$$

Multiplying by the denominator, expanding the sum up to m terms, by comparison of coefficients we have the resulting system of equations which can be solved for $\{a_i : 0 \leq i \leq m\}$ by standard linear algebra:

$$\sum_{j=0}^{i} c_j a_{i-j} = \begin{cases} b_i, \text{ for } 0 \leq i \leq m\\ 0, \text{ for } i > m \end{cases}$$

For any DFA A, once the recurrence relation has been derived, and since each coefficient a_i is associated with $|\mathcal{L}_i(A)|$, given a string length size k this gives us a simple $O(kn)$ method to compute $|\mathcal{L}_i(A)|$ for any k. In addition, standard techniques for solving linear homogeneous recurrence relations can be used to derive a closed form solution for $|\mathcal{L}_i(A)|$ [22].

Counting All Solutions within a Given Bound: The above described method gives a generating function that encodes each $|\mathcal{L}_i(A)|$ *separately.* Instead, we seek a generating function that encodes the number of *all solutions within a bound*. To this end we define the automata model counting function

$$\mathcal{MC}_A(k) = \sum_{i \geq 0}^{k} |\mathcal{L}_i(A)|. \tag{22}$$

In order to compute $\mathcal{MC}_A(k)$ we make a simple adjustment. All that is needed is to add a single λ-cycle (the dashed transition in Fig. 2(b)) to the accepting state of the augmenting DFA A'. Then $\mathcal{L}_{k+1}(A') = \bigcup_{i=0}^{k} \mathcal{L}_i(A) \cdot \lambda^{k-i}$ and the accepting paths of strings in $\mathcal{L}_{k+1}(A')$ are in one-to-one correspondence with the accepting paths of strings in $\bigcup_{i=0}^{k} \mathcal{L}_i(A)$. Consequently, $|\mathcal{L}_{k+1}(A')| = \sum_{i=0}^{k} |\mathcal{L}_i(A)|$ and so $\mathcal{MC}_A(k) = |\mathcal{L}_{k+1}(A')|$. Hence we can compute $\mathcal{MC}_A$ using recurrence for $|\mathcal{L}_i(A')|$ with the additional λ-transition.

4 Implementation

We implemented the techniques we presented in earlier sections in a tool called Automata-Based model Counter for string constraints (ABC). We implemented ABC using the symbolic string analysis library provided by the Stranger tool [40, 37, 39]. We used the symbolic DFA representation of the MONA DFA library [8] to implement the constructions described in Section 2. In MONA's DFA library, the transition relation of the DFA is represented as a Multi-terminal Binary Decision Diagram (MBDD) which results in a compact representation of the transition relation. ABC implementation supports more operations (such as trim, substring) than the ones listed in Section 2 using constructions similar to the ones given in that section.

ABC supports the SMT-LIB 2 language syntax. We specifically added support for CVC4 string operations [24]. In string constraint benchmarks provided by CVC4, boolean variables are used to assert the results of subformulas. In our automata-based constraint solver, we check the satisfiability of a formula by checking if its truth set is empty or not. We eliminated the boolean variables that are only used to check the results of string operations (such as string equivalence, string membership) and instead substituted the corresponding expressions directly. We converted if-then-else structures into disjunctions. We also searched for several patterns between length equations and word equations to infer the values of the string variables whenever possible (for example when we see the constraint $\textsc{len}(x) = 0$ we can infer that the string variable x must be equal to empty string). These transformations allow us to convert some constraints to pseudo-relational constraints that we can precisely solve. If these transformations do not resolve all the cyclic dependencies in a constraint then the resulting DFA may recognize an over-approximation of all possible solutions.

We implemented the automata-based model counting algorithm of Section 3 by passing the automaton transfer matrix to Mathematica for computing the generating function, corresponding recurrence relation, and the model count for a specific bound. Because the DFAs we encountered in our experiments typically have sparse transition graphs, we make use of Mathematica's powerful and efficient implementations of symbolic sparse matrix determinant functions [34].

5 Experiments

To evaluate ABC we experimented with a set of Java application benchmarks, SMT-LIB 2 translation of Kaluza JavaScript benchmarks and several examples from SMC distribution. In our experiments we compared ABC to SMC [25] and CVC4 [24]. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 14.04[1].

Table 1. Constraint Characteristics

Benchmarks	# Constraints	Frequency of Operations Per 1000 Formulas									
		$\in$	.	=	LEN	REPLACE	INDEXOF	CONTAINS	BEGINS	ENDS	SUBSTRING
ASE	116164	0.42	386.10	129.39	382.54	639.28	4.11	7.52	16.91	7.51	41.17
Kaluza Small	368433	30.29	93.89	224.87	46.84	0	0	0	0	0	0
Kaluza Big	5138323	38.12	129.53	164.64	60.46	0	0	0	0	0	0

Table 1 shows the frequency of string operations from our string constraint grammar that are contained in the ASE, Kaluza Small, and Kaluza Big Benchmark sets. ASE benchmarks are from Java programs and represent server-side code [20]. The Kaluza benchmarks are taken from JavaScript programs and represent client-side code [29]. All three benchmarks have regular expression membership ($\in$), concatenation (.), string equality (=), and length constraints. However, the ASE benchmark contains additional string operations that are typically used for input sanitization, like replace and substring.

Java Benchmarks. String constraints in these benchmarks are extracted from 7 real-world Java applications: Jericho HTML Parser, jxml2xql (xml-to-sql converter), MathParser (math expression solver), MathQuizGame (math study tool), Natural CLI (a natural language command line tool), Beasties (a command line game), HtmlCleaner (html to clean xml converter), and iText (PDF library) [20]. These benchmarks represent server-side code and employ many input-sanitizing string operators such as replace and substring as seen in Table 1. These string constraints were generated by extracting program path constraints through dynamic symbolic execution [20].

In [20] an empirical evaluation of several string constraint solvers is presented. As a part of this empirical evaluation, the authors use the symbolic string analysis library of Stranger [40, 37, 39] to construct automata for path constraints on strings. In order to evaluate the model counting component of ABC, we ran their tool on the 7 benchmark sets and output the resulting automata whenever the constraint is satisfiable. Out of 116,164 string path constraints, 66,236 were found to be satisfiable and we performed model counting on those cases. The constraints in Java benchmarks are all single-variable or pseudo-relational constraints. The resulting automata do not have any over-approximation caused by relational constraints. As a measure of the size of the resulting automata, we can give the number of BDD nodes used in the symbolic transition relation representation of MONA. The average number of BDD nodes for the satisfiable path constraints is 69.51 and the size of the each BDD node is 16 bytes. For these benchmarks our model-counter is very efficient, where the average running time of model counter per path constraint is 0.0015 seconds and the model-counting

[1] Results of our experiments are available at http://www.cs.ucsb.edu/˜vlab/ABC/

function that our model-counter produces is precise, i.e., gives the exact count for any given bound.

SMC and CVC4 are not able to handle the constraints in this data set since they do not support sanitization operations such as replace.

Table 2. Log scaled comparison between SMC and ABC

	bound	SMC	SMC	ABC
		lower bound	upper bound	count
nullhttpd	500	3752	3760	3760
ghttpd	620	4880	4896	4896
csplit	629	4852	4921	4921
grep	629	4676	4763	4763
wc	629	4281	4284	4281
obscure	6	0	3	2

SMC Examples. For a comparative evaluation of our tool with SMC, we used the examples that are listed on SMC's web page. We translated the 6 example constraints listed in table 2 into SMT-LIB2 language format that we support. We inspected the examples to confirm that they are pseudo-relational, i.e., our analysis generates a precise model-counting function for these constraints. We compare our results with the SMC results reported in the SMC's web page. The first column of the Table 2 shows the file names of these example constraints. The second column shows the bounds used for obtaining the model counts. The next two columns show the log-scale SMC lower and upper bound values for the model counts. The last column shows the log-scale model count produced by ABC. We omit the decimal places of the numbers to fit them on the page. For all the cases ABC generates a precise count given the bound. ABC's count is exactly equal to SMC's upper bound for four of the examples and is exactly equal to SMC's lower bound for one example. For the last example ABC reports a count that is between the lower and upper bound produced by SMC. Note that these are log scaled values and actual differences between a lower and an upper-bound values are huge. Although SMC is unable to produce an exact answer for any of these examples, ABC produces an exact count for each of them.

JavaScript Benchmarks. We also experimented with Kaluza benchmarks which were extracted from JavaScript code via dynamic symbolic execution [29]. These benchmarks are divided to a small and large set based on the sizes of the constraints. These benchmarks have been used by both SMC and CVC4 tools. ABC handles 19,731 benchmark constraints in satisfiable small set with an average of 0.32 seconds per constraint for model counting, whereas SMC handles 17,559 constraints with an average of 0.26 seconds per constraint. ABC handles and 1,587 benchmark constraints in satisfiable big set with an average of 0.34 seconds per constraint for model counting, whereas SMC handles 1,342 constraints with an average of 5.29 seconds per constraint. We were not able to do a one-to-one timing and precision comparison between ABC and SMC for each constraint due to an error in the SMC data file (the mapping between file names and results is wrong).

Satisfiability Checking Evaluation. We ran ABC on SMT-LIB 2 translation of the full set of JavaScript benchmarks. We put 20-seconds CPU timeout limit to our solver. Table 3 shows the comparison between ABC and CVC4 [24]

Table 3. Constraint-Solver Comparison

	ABC	CVC4	ABC	CVC4	ABC	CVC4	ABC	CVC4	ABC	CVC4
	sat - sat		unsat-unsat		sat-unsat		unsat-sat		sat-timeout	
sat/small	19728		3		0		0		0	
sat/big	1587		0		0		0		0	
unsat/small	8139		3013		74		0		0	
unsat/big	3419		5904		2385		0		2359	

constraint solver based on the CVC4 results that are available online. The first column shows the initial satisfiability classification of the data set by the creators of the benchmarks [29]. The next two columns show the number of results that ABC and CVC4 agree. The last three columns show the cases where ABC and CVC4 differ. Note that, since ABC over-approximates the solution set, if the given constraint is not single-valued or pseudo-relational, it is possible for ABC to classify a constraint as satisfiable even if it is unsatisfiable. However, it is not possible for ABC to classify a constraint unsatisfiable if it is satisfiable. Out of 47,284 benchmark constraints ABC and CVC4 agree on 41,793 of them. As expected ABC never classifies a constraint as unsatisfiable if CVC4 classifies it as satisfiable. However, due to over-approximation of relational constraints, ABC classifies 2,459 constraints as satisfiable although CVC4 classifies them as unsatisfiable. A practical approach would be to use ABC together with a satisfiability solver like CVC4, and, given a constraint, first use the satisfiability solver to determine the satisfiability of the formula, and then use ABC to generate its trust set and the model counting function.

The average automata construction time for big benchmark constraints is 2.43 seconds and for small benchmark constraints is 0.001 seconds. ABC never timeouts and reports that the constraint is satisfiable for 2359 constraints that CVC4 timeouts. ABC is unable to handle 672 constraints and for these 672 constraints CVC4 timeouts in 29 of them, reports unsat for 246 of them, and reports sat for 397 of them. There are also a few thousand constraints from the Kaluza benchmarks that CVC4 is unable to handle.

6 Conclusions and Future Work

We presented a string constraint solver that, given a constraint, generates: 1) An automaton that accepts all solutions to the given string constraint; 2) A model-counting function that, given a length bound, returns the number of solutions within that bound. Our experiments on thousands of constraints extracted from real-world web applications demonstrates the effectiveness and efficiency of the proposed approach. Our string constraint solver can be used in quantitative information flow, probabilistic analysis and automated repair synthesis. We plan to extend out automata-based model-counting approach to Presburger arithmetic constraints using an automata-based representation for Presburger arithmetic constraints [35, 4].

References

1. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 150–166, 2014.
2. Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. Semantic differential repair for input validation and sanitization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 225–236, 2014.
3. Muath Alkhalaf, Tevfik Bultan, and Jose L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 947–957, 2012.
4. Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.
5. N. Biggs. *Algebraic Graph Theory*. Cambridge Mathematical Library. Cambridge University Press, 1993.
6. Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. Path feasibility analysis for string-manipulating programs. In *Proceeding of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 307–321, 2009.
7. Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Pasareanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedigns of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
8. BRICS. The MONA project. http://www.brics.dk/mona/.
9. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
10. David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3), 2007.
11. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
12. CVE. Common Vulnerabilities and Exposures. http://www.cve.mitre.org.
13. Loris DAntoni and Margus Veanes. Static analysis of string encoders and decoders. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'13)*, pages 209–228. Springer, 2013.
14. Antonio Filieri, Corina S. Pasareanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 622–631, 2013.
15. Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, New York, NY, USA, 1 edition, 2009.
16. Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2013.
17. Pieter Hooimeijer, Benjamin Livshits, David Molnar, Prateek Saxena, and Margus Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

18. Pieter Hooimeijer and Westley Weimer. A decision procedure for subset constraints over regular languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 188–198, 2009.
19. Pieter Hooimeijer and Westley Weimer. Solving string constraints lazily. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2010.
20. Scott Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *29th ACM/IEEE International Conference on Automated software engineering (ASE)*, pages 259–270, 2014.
21. Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116, 2009.
22. Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.
23. Guodong Li and Indradeep Ghosh. PASS: string solving with parameterized array and interval automaton. In *Proceedings of the 9th International Haifa Verification Conference (HVC)*, pages 15–31, 2013.
24. Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Proceedings of the 26th International Conference (CAV)*, pages 646–662, 2014.
25. Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 57, 2014.
26. Open Web Application Security Project (OWASP). Top ten project. https://www.owasp.org/index.php/Category OWASP_Top_Ten_Project.
27. Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.
28. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland 2010)*, 2010.
29. Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. *Security and Privacy, IEEE Symposium on*, 0:513–528, 2010.
30. Geoffrey Smith. On the foundations of quantitative information flow. In *FOSSACS '09*, pages 288–302, 2009.
31. Richard P. Stanley. *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition, 2011.
32. Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 166–176, 2011.
33. Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1232–1243, 2014.
34. Inc. Wolfram Research. Mathematica, 2014.
35. Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS*, pages 1–19, 2000.

36. Fang Yu. *Automatic Verification of String Manipulating Programs*. PhD thesis, University of California, Santa Barbara, 2010.
37. Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, 2010.
38. Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In *International Conference on Software Engineering (ICSE)*, pages 131–134, 2011.
39. Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.
40. Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN*, pages 306–324, 2008.
41. Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 114–124, 2013.