Parameterized Model Counting for String and Numeric Constraints[*]

Abdulbaki Aydin
Microsoft
USA
abakiaydinn@gmail.com

William Eiers
University of California Santa Barbara
USA
weiers@ucsb.edu

Lucas Bang
University of California Santa Barbara
USA
bang@ucsb.edu

Tegan Brennan
University of California Santa Barbara
USA
tegan@ucsb.edu

Miroslav Gavrilov
University of California Santa Barbara
USA
mvg@ucsb.edu

Tevfik Bultan
University of California Santa Barbara
USA
bultan@ucsb.edu

Fang Yu
National Chengchi University
Taiwan
yuf@nccu.edu.tw

ABSTRACT

Recently, symbolic program analysis techniques have been extended to quantitative analyses using model counting constraint solvers. Given a constraint and a bound, a model counting constraint solver computes the number of solutions for the constraint within the bound. We present a parameterized model counting constraint solver for string and numeric constraints. We first construct a multi-track deterministic finite state automaton that accepts all solutions to the given constraint. We limit the numeric constraints to linear integer arithmetic, and for non-regular string constraints we over-approximate the solution set. Counting the number of accepting paths in the generated automaton solves the model counting problem. Our approach is parameterized in the sense that, we do not assume a finite domain size during automata construction, resulting in a potentially infinite set of solutions, and our model counting approach works for arbitrarily large bounds. We experimentally demonstrate the effectiveness of our approach on a large set of string and numeric constraints extracted from software applications. We experimentally compare our tool to five existing model counting constraint solvers for string and numeric constraints and demonstrate that our tool is as efficient and as or more precise than other solvers. Moreover, our tool can handle mixed constraints with string and integer variables that no other tool can.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; • **Software and its engineering → Software verification**;

KEYWORDS

Model counting, constraint solving, quantitative program analysis

ACM Reference Format:
Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. 2018. Parameterized Model Counting for String and Numeric Constraints. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236064

1 INTRODUCTION

Quantitative program analysis arises in many contexts such as probabilistic analysis [11, 17], reliability analysis [15] and quantitative information flow [5, 8, 29, 30, 34]. Quantitative program analyses require efficient model counting constraint solvers to handle complex and diverse constraints generated during program analyses. A model counting constraint solver computes the number of solutions for a given constraint within a given bound [4, 7, 12–14, 27].

In this paper, we present a model counting constraint solver that can handle both numeric and string constraints and their combinations. Given a constraint, we construct a multi-track deterministic finite automaton (DFA) that accepts tuples of values that correspond to the solutions of the given constraint. For numeric constraints, we focus on linear integer arithmetic constraints, and the constructed automaton accepts a binary encoding of the numbers that satisfy the given numeric constraint. Since some string constraints can have non-regular solution sets, our automata construction approach over-approximates the solution set in such cases. Hence, our model

[*]This material is based on research supported by an Amazon Research Award and by DARPA under the agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

counting constraint solver provides a sound upper-bound for the number of solutions for a given constraint.

Since we use multi-track DFA, we can represent relational constraints that specify relationships among variables. Moreover, our approach handles interactions between numeric and string constraints in the presence of operations such as string *length* which can be used together with numeric variables in a constraint.

Automata-based constraint solving reduces the model counting problem to path counting. To count the number of values that satisfy the given constraint within a given domain bound, we count the number of accepting paths in the automaton within the path length bound that corresponds to said domain bound. We use techniques from algebraic graph theory to solve the path counting problem.

We implemented the techniques we present in this paper in a tool called Multi-Track Automata Based model Counter (MT-ABC). We experimented on a large set of constraints generated during symbolic execution of Java and JavaScript programs and compared MT-ABC with five existing model counting constraint solvers [4, 7, 14, 27, 38]. Our experiments demonstrate that MT-ABC is as efficient and as or more precise than existing tools. More importantly, MT-ABC is the only tool can handle the union of all constraints that existing tools can handle, and MT-ABC is the only tool that can handle mixed numeric and string constraints that contain both string and integer variables.

Our novel contributions in this paper are: 1) an extended constraint language that is more expressive than constraint languages supported by other model counting constraint solvers (Section 2), 2) handling of relational string constraints using multi-track automata (Section 3), 3) handling of mixed string and integer constraints using multiple automata (Section 3), 4) model counting for tuples of variables (Section 4), 5) heuristics for constraint simplification (Section 5), and 6) an extensive experimental evaluation (Section 6).

A Motivating Example. Let us give an example demonstrating the use of model counting constraint solvers for quantitative information flow analysis, and how this type of analysis can be used to quantify information leakage through side-channels. This example is based on a security vulnerability known as "Compression Ratio Info-leak Made Easy" (CRIME) [21, 32]. Many web server requests are compressed and encrypted for efficiency and security before transmission as a network packet. Despite the encryption, a malicious attacker who can observe network packet sizes can use the compression size to learn secret web-session information. Assume an attacker can inject and concatenate his own text with the secret text prior to compression. The smaller the resulting packet, the more compression must have occurred prior to encryption, and so the attacker-controlled input must contain substrings which match substrings of the secret text. In the CRIME attack, encryption does not significantly change the size of the packet, as many encryption protocols are size-preserving. Thus, by carefully crafting injected inputs, an attacker can incrementally reveal the secret text.

For instance, suppose the secret is the text: "`sessionkey:21620`" If the attacker is able to inject the text string: "`sessionkey:12345`" he will observe less compression than if he injects: "`sessionkey:21600`" because there is a longer prefix match between the attacker string and the secret string. In this way, the attacker is able to make repeated guesses and incrementally learn more information about prefixes of the secret.

Consider a simple method for compressing the concatenation of two strings. For strings s and t, we compress their concatenation, $s \cdot t$, by checking if t is a prefix of s, and if so, encoding their concatenation as $s;[k]$ where k is the length of t. If t is not a prefix of s they are simply concatenated. The notation $s;[k]$ is interpreted as a pointer which indexes into s, indicating how many characters of s to expand in order to recover t. For example, if s is the string "Hello, World!" and t is the string "Hello", $s \cdot t$ is encoded as "Hello, World!;[5]". The following is a simple Java function for performing this combined concatenation and compression:

```
public String compress(String s, String t) {
if(s.startsWith(t)) return s + ";[" + t.\length() + "]";
else return s + t; }
```

This function results in an exploitable vulnerability similar to the CRIME attack. Suppose that s is a secret string of 5 numeric characters, and a malicious adversary has control over t. If the adversary is able to observe the size of the resulting compression, he can learn information about s by varying t.

We will assume that the alphabet for s and t is the set of numeric characters: '1', ..., '9'. By performing symbolic execution of `compress(s,t)`, we can determine path constraints which lead to different possible observations on the size of the result. For example, (using the constraint language we define in Section 2) we can see that if $(\textbf{length}(s) = 5) \wedge (\textbf{begins}(s,t) \vee \textbf{length}(t) = 4)$ then the length of the resulting string is 9. One may verify that there are 10,005 possible pairs of strings (s,t) that satisfy this constraint. If $\neg\textbf{begins}(s,t) \wedge \textbf{length}(s) = 5 \wedge \textbf{length}(t) = 5$ then the resulting string will have length 10, and there are 99,999 possible (s,t) which satisfy this constraint. Assuming that s is uniformly distributed, we can compute the probability of each observation by dividing the number of solutions by the total domain size.

Prior work in quantitative information flow has proposed using entropy as a measure of information leakage [5, 8, 29, 30, 34]. Given a probability distribution over program observables, the *Shannon entropy* of the distribution is defined as $H(p) = -\sum_{i=1}^{n} p_i \log p_i$. Applying this to the probabilities computed using a model counting constraint solver, we can quantify the amount of information leaked for our example as 0.52 bits. Note that, in addition to a standard symbolic execution tool, all we need in order to be able to perform this kind of quantitative information flow analysis is a model counting constraint solver, and for this particular example, we need a model counting constraint solver that can handle string constraints and numeric constraints together.

2 CONSTRAINT LANGUAGE

We define our constraint language using the grammar shown in Fig. 1, where φ denotes a formula, β denotes a numeric term, γ denotes a string term, $\varphi_{\mathbb{Z}}$ denotes a numeric constraint (an atomic formula) constructed from terms and expressions, $\varphi_{\mathbb{S}}$ denotes a string constraint (an atomic formula) constructed from terms and expressions, ρ denotes a constant regular expression, n denotes an integer constant, $\top$ and $\bot$ denote constants true and false, and v_i and v_s denote integer and string variables, respectively. We use α to denote φ, $\varphi_{\mathbb{Z}}$, $\varphi_{\mathbb{S}}$, β, or γ.

Given alphabet Σ, $s \in \Sigma^*$ denotes a string value and ε denotes the empty string. A character is a string that has length one. The string operations "$\cdot$", "$|$", and "$*$" correspond to regular expression

$$
\begin{array}{lll}
\varphi & \longrightarrow & \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi \mid \varphi_{\mathbb{Z}} \mid \varphi_{\mathbb{S}} \mid \top \mid \bot \\
\varphi_{\mathbb{Z}} & \longrightarrow & \beta = \beta \mid \beta < \beta \mid \beta > \beta \\
\varphi_{\mathbb{S}} & \longrightarrow & \gamma = \gamma \mid \gamma < \gamma \mid \gamma > \gamma \mid \textbf{match}(\gamma, \rho) \mid \textbf{contains}(\gamma, \gamma) \\
 & \mid & \textbf{begins}(\gamma, \gamma) \mid \textbf{ends}(\gamma, \gamma) \\
\beta & \longrightarrow & v_i \mid n \mid \beta + \beta \mid \beta - \beta \mid \beta \times n \\
 & \mid & \textbf{length}(\gamma) \mid \textbf{toint}(\gamma) \mid \textbf{indexof}(\gamma, \gamma) \mid \textbf{lastindexof}(\gamma, \gamma) \\
\gamma & \longrightarrow & v_s \mid \rho \mid \gamma \cdot \gamma \mid \textbf{reverse}(\gamma) \mid \textbf{tostring}(\beta) \mid \textbf{charat}(\gamma, \beta) \mid \\
 & \mid & \textbf{substring}(\gamma, \beta, \beta) \mid \textbf{replacefirst}(\gamma, \gamma, \gamma) \mid \textbf{replacelast}(\gamma, \gamma, \gamma) \\
 & \mid & \textbf{replaceall}(\gamma, \gamma, \gamma) \\
\rho & \longrightarrow & \varepsilon \mid s \mid \rho \cdot \rho \mid \rho \mid \rho \mid \rho^*
\end{array}
$$

Figure 1: Constraint language grammar

operations concatenation, alternation, and Kleene closure, respectively. Comparators "<" and ">" on string terms correspond to lexicographical comparisons. An *atomic constraint* refers to a formula without any boolean connectives. Notice that an integer term produced from the production rule β may contain string terms γ and vice versa; a constraint produced in this way is called a *mixed constraint*. The constraint language from Fig. 1 is rich enough to capture common constraints that appear in Java and PHP programs. Formal semantics of this constraint language is described in [3].

The set of variables present in φ is given by $\mathcal{V}(\varphi)$. A *model* for φ is an assignment of all variables in $\mathcal{V}(\varphi)$ where φ evaluates to *true*. The truth set of a formula φ, denoted $[\![\varphi]\!]$, is the set of all models of φ. The goal of model counting is to determine the size of $[\![\varphi]\!]$.

3 CONSTRAINT SOLVING VIA AUTOMATA

A multi-track DFA A is a 5-tuple $(Q, \vec{\Sigma}, \delta, q_0, F)$, where Q is the set of states, $\vec{\Sigma} = (\Sigma \cup \{\lambda\})^k$ is the k-track input alphabet where Σ is the set of alphabet symbols for one track, $\lambda \notin \Sigma$ is a padding symbol that appears only at the end of a string in each track, $\delta\colon Q \times \vec{\Sigma} \to Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. Multi-track DFA are closed under intersection, union and complement [42]. With each track of A, we associate a unique identifier v_i, which we refer to as the variable for track i. The set of track variables for A is denoted $\mathcal{V}(A)$. The language of all strings recognized by A is denoted $\mathcal{L}(A)$ where $\mathcal{L}(A) \subseteq \vec{\Sigma}^*$. Given a word $w \in \mathcal{L}(A)$, we use $w[v_i] \in \Sigma^*$ to denote the value of track i. Hence, $w \in \mathcal{L}(A)$ denotes a tuple of values $(w[v_1], w[v_2], \ldots, w[v_k])$, one value for each variable in $\mathcal{V}(A)$.

Given a formula φ, our goal is to construct an automaton A, such that $\mathcal{L}(A) = [\![\varphi]\!]$, where the tracks of A correspond to the variables of φ. We call this DFA the *solution automaton* for φ. Some mixed constraints and some pure string constraints have non-regular truth sets [42]. For such constraints we provide a sound over approximation by constructing an automaton A such that $[\![\varphi]\!] \subseteq \mathcal{L}(A)$.

During our construction, in addition to having one track for each variable of the formula in the multi-track automaton, we also create one track for each string term (shown as γ in Figure 1) and one track for each numeric term (shown as β in Figure 1). Actually, for the terms that correspond to addition, subtraction and multiplication with a constant we do not create separate tracks as we discuss in Section 3.3. Given a term γ or β, we use $\tau(\gamma)$ and $\tau(\beta)$ to denote the tracks that those terms are associated with.

We define a projection operation π such that, given an automaton A and a variable set V, $\pi(A, V)$ is an automaton A' where $\mathcal{V}(A') = V$. Let $x_1, \ldots, x_n \in V \setminus \mathcal{V}(A)$ be the variables in V but not in $\mathcal{V}(A)$ and $y_1, \ldots, y_m \in \mathcal{V}(A) \setminus V$ be the variables in $\mathcal{V}(A)$ but not in V. That is, we wish to add new unconstrained x_i tracks to A and remove y_j tracks from A. Then, we define $\pi(A, V)$ to be a multi-track DFA A' with $\mathcal{V}(A') = V$ such that:

$$w' \in \mathcal{L}(A') \Leftrightarrow \exists w \in \mathcal{L}(A), \forall v \in \mathcal{V}(A') \cap \mathcal{V}(A), w[v] = w'[v].$$

Algorithm 1 Solve(A, α)

Procedure operates on an automaton A which is passed by reference and has a track for each variable and term in α.

α is one of the following: a conjunction of numeric and string constraints, a string constraint, a numeric constraint, a string term, or a numeric term.

$\star \in \{=, \neq, <, \leq, >, \geq,$ **match**, $\neg$**match**, **contains**, $\neg$**contains**, **begins**, $\neg$**begins**, **ends**, $\neg$**ends**$\}$

$\odot \in \{-, +, \times,$ **length**, **toint**, **indexof**, **lastindexof**, **reverse**, **tostring**, **charat**, **substring**, **replacefirst**, **replacelast**, **replaceall**$\}$

```
1: if $\alpha \equiv \alpha_1 \wedge \alpha_2$ then
2:     Solve($A$, $\alpha_1$); Solve($A$, $\alpha_2$);
3:     Propagate($A$, $\alpha_1$); Propagate($A$, $\alpha_2$);
4: else if $\alpha \equiv \alpha_1 \star \alpha_2$ then
5:     Solve($A$, $\alpha_1$); Solve($A$, $\alpha_2$);
6:     Refine($A$, $\star$, $\tau(\alpha_1)$, $\tau(\alpha_2)$)          ▷ modifies tracks $\tau(\alpha_1)$ and $\tau(\alpha_2)$
7:     Propagate($A$, $\alpha_1$); Propagate($A$, $\alpha_2$);
8: else if $\alpha \equiv \odot(\alpha_1, \ldots, \alpha_n)$ then
9:     for all $\alpha_i \in \{\alpha_1, \ldots, \alpha_n\}$ do
10:        Solve($A$, $\alpha_i$);
11:    end for
12:    Restrict($A$, $\tau(\alpha)$, $\odot$, $\tau(\alpha_1)$, ..., $\tau(\alpha_n)$);          ▷ modifies track $\tau(\alpha)$
13: end if
```

Algorithm 2 Propagate(A, φ)

Procedure operates on an automaton A which is passed by reference and has a track for each variable and term in φ.

$\odot \in \{-, +, \times,$ **length**, **toint**, **indexof**, **lastindexof**, **reverse**, **tostring**, **charat**, **substring**, **replacefirst**, **replacelast**, **replaceall**$\}$

```
1: if $\varphi \equiv \odot(\alpha_1, \ldots, \alpha_n)$ then
2:     Refine($A$, $\tau(\alpha)$, $\odot$, $\tau(\alpha_1)$, ..., $\tau(\alpha_n)$);   ▷ modifies tracks $\tau(\alpha_1)$ to $\tau(\alpha_n)$
3:     for all $\alpha_i \in \{\alpha_1, \ldots, \alpha_n\}$ do
4:         Propagate($A$, $\alpha_i$);
5:     end for
6: else if $\varphi \equiv \varphi_1 \wedge \varphi_2$ then
7:     Propagate($A$, $\varphi_1$); Propagate($A$, $\varphi_2$);
8: else if $\varphi \equiv \varphi_1 \vee \varphi_2$ then
9:     $A_{\varphi_1} = A \cap A_{\varphi_1}$; $A_{\varphi_2} = A \cap A_{\varphi_2}$;
10:    Propagate($A_{\varphi_1}$, $\varphi_1$); Propagate($A_{\varphi_2}$, $\varphi_2$);
11: end if
```

3.1 Automata Construction

Since the negation operator is non-monotonic and since we sometimes over-approximate the solution sets of subformulas, before the automata construction, we convert the input formula to negation normal form by pushing negations to atomic formulas.

We first describe our automata construction algorithm for constraints which are conjunctions of numeric and string constraints (i.e., $\varphi_{\mathbb{Z}}$ and $\varphi_{\mathbb{S}}$ in Fig. 1, respectively). We describe how we handle combinations of conjunctions and disjunctions later.

Let φ be a formula which is a conjunction of numeric and string constraints. The automata construction procedure Solve (Algorithm 1) recursively constructs a multi-track automaton A such that, when A is projected to the variables of φ (i.e., $\mathcal{V}(\varphi)$), it accepts an over approximation of φ solutions set, i.e., $[\![\varphi]\!] \subseteq \mathcal{L}(\pi(A, \mathcal{V}(\varphi)))$.

Procedure Solve passes the automaton A by reference, so there is a single automaton A that is being modified. Before the first call to the Solve procedure, A is initialized so that all tracks accept all strings, i.e., initially, $\mathcal{L}(A) = \vec{\Sigma}^*$.

The procedure Solve uses three other procedures during the construction of automaton A: Restrict, Refine and Propagate. Again, the automaton A is passed by reference, so all these procedures modify the same automaton A during construction.

The procedure Restrict is used to compute the result of a string or numeric operator. Note that, there is a track in A for each term in φ, so the result of each string or numeric operator has a track reserved for the corresponding term. Let us denote the string or numeric operator with the symbol $\odot$, where $\alpha \equiv \odot(\alpha_1, \ldots, \alpha_n)$. Then, Restrict($A$, $\text{T}(\alpha), \odot, \text{T}(\alpha_1), \ldots, \text{T}(\alpha_n)$) restricts the track in A that corresponds to the term $\odot(\alpha_1, \ldots, \alpha_n)$ based on the tracks of the arguments $\alpha_1, \ldots, \alpha_n$ in A. For this to work, we need to make sure that the arguments' tracks are processed first, and this is done in the for loop before Restrict is called.

For example, consider the term $\textbf{charat}(v, i)$. Restrict(A, $\text{T}(\textbf{charat}(v, i))$, **charat**, $\text{T}(v)$, $\text{T}(i)$) restricts track $\text{T}(\textbf{charat}(v, i))$ in A to string values that correspond to characters that can appear at location i of string v, where possible values for v and i are specified by the values recognized by tracks $\text{T}(v)$ and $\text{T}(i)$, respectively.

The procedure Refine is used to reflect the constraint imposed by a string or numeric predicate or its negation to its arguments. Let us denote the string or numeric predicate with the symbol $\star$, where $\alpha_1 \star \alpha_2$ and α_1 and α_2 are string or numeric terms. Then, Refine($A, \star, \text{T}(\alpha_1), \text{T}(\alpha_2)$) reflects the constraint imposed by the predicate $\alpha_1 \star \alpha_2$ to the tracks $\text{T}(\alpha_1)$ and $\text{T}(\alpha_2)$. Before Refine is called arguments of the predicate $\star$ are processed.

For example, for the equality predicate $\textbf{charat}(v, i) = \texttt{"a"}$, Refine($A, =, \text{T}(\textbf{charat}(v, i)), \text{T}(\texttt{"a"})$) restricts the set of values for track $\text{T}(\textbf{charat}(v, i))$, to the string "a". Note that, since "a" is a constant, we do not actually need a track for it, but for simplicity of presentation, let us assume that constants are also assigned a track which accept just the value that corresponds to the constant.

After Refine is called, the set of strings recognized by the arguments' tracks may have changed and must be propagated to the other tracks (as arguments can be terms constructed from other arguments). This is done using the Propagate procedure. For example, once we refine the set of values for track $\textbf{charat}(v, i)$ based on the predicate $\textbf{charat}(v, i) = \texttt{"a"}$ we have to propagate this change to the arguments of the operator **charat** and refine the values for $\text{T}(v)$ and $\text{T}(i)$. We call Propagate(A, $\textbf{charat}(v, i)$) to do this.

In general, we use the Propagate procedure when the result of a string or numeric operator is refined due to a string or numeric predicate, and this refinement has to be propagated to the arguments of the operator. As shown in Algorithm 2, Propagate(A, $\odot(\alpha_1, \ldots, \alpha_n)$) first calls Refine($A, \text{T}(\alpha), \odot, \text{T}(\alpha_1), \ldots \text{T}(\alpha_n)$) which refines the tracks for the arguments of the operator $\odot$ based on the track for the $\odot$ term. After this refinement, it recursively calls the procedure Propagate on the arguments of the $\odot$ term to further propagate the refinement.

As shown in Algorithm 3, we extend the Solve procedure to combinations of conjunctions and disjunctions. For conjunctions we use a single automaton. After a conjunction is solved, it is necessary to propagate the result to the children of the conjunction. After propagation, the conjunction is solved again so that the final automaton captures all the refinements.

Algorithm 3 Solve(A, φ)

Procedure operates on an automaton A which is passed by reference.
Disjunctions create a separate automaton for each disjunct.
Conjunctions use a single automaton for all conjuncts.

```
1: if $\varphi \equiv \varphi_1 \lor \varphi_2$ then
2:     Solve($A_{\varphi_1}$, $\varphi_1$); Solve($A_{\varphi_2}$, $\varphi_2$) ;
3:     $A = A_{\varphi_1} \cup A_{\varphi_2}$          ▷ Union computed using automata product
4: else if $\varphi \equiv \varphi_1 \land \varphi_2$ then
5:     Solve($A$, $\varphi_1$); Solve($A$, $\varphi_2$);
6:     Propagate($A$, $\varphi_1$); Propagate($A$, $\varphi_2$);
7:     Solve($A$, $\varphi_1$); Solve($A$, $\varphi_2$);
8: end if
```

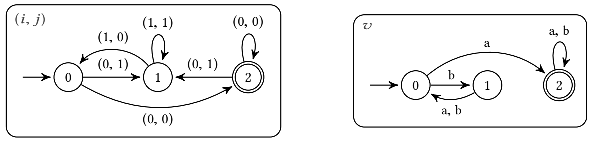


Figure 2: Automata constructed for Example 1

For disjunctions, each disjunct has its own automaton. Then, the automaton for the disjunction corresponds to the automaton that accepts the union of sets accepted by each disjunct automaton. We compute the union automaton using automata product.

Let us consider the following example constraint:

$$\textbf{charat}(v, i) = \texttt{"a"} \land i = 2 \times j \qquad (1)$$

We show the resulting automata in Figure 2. Note that, to make the example more readable, we split the automaton to two, one for string variables and one for integer variables. In fact, in our implementation we also split the automata to multiple automata based on the dependencies among variables, which we discuss later with other heuristics.

3.2 String Constraint Solving

We now discuss how Algorithm 1 handles atomic constraints $\alpha \equiv \alpha_1 \star \alpha_2$, when α is a $\varphi_{\mathbb{S}}$ term, $\star$ is a string predicate, and α_1 and α_2 are string terms (γ). In particular, we will focus on the Restrict and Refine procedures on string terms and string predicates, and discuss a representative subset of string terms and string predicates.

Let us use the notation introduced in Figure 1 where β denotes integer terms, γ denotes string terms, and ρ denotes regular expression terms. Given an automaton A, function $\text{T}(\alpha)$ represents the possible values of the term α that is encoded as a track in the given automaton. Let $\text{T}'(\alpha)$ represent the result of a Restrict or Refine procedure call on the corresponding track. The prefixes : $\Sigma^* \rightarrow \Sigma^*$ function computes the set of prefixes for a given set of strings and the suffixes : $\Sigma^* \rightarrow \Sigma^*$ function computes the set of suffixes for a given set of strings. Both functions can be implemented using projection, determinization, and minimization operations on DFAs.

Let us consider the operations **length**, **indexof**, **substring**, **charat**, and "·" (string concatenation) operations.

Restrict(A, $\text{T}(\textbf{length}(\gamma))$, **length**, $\text{T}(\gamma)$):

$$\text{T}'(\textbf{length}(\gamma)) = \{i \mid \exists s \in \text{T}(\gamma) : i = |s| \land i \in \text{T}(\textbf{length}(\gamma))\}$$

$\textsc{Restrict}(A, \mathrm{T}(\mathbf{indexof}(\gamma_1, \gamma_2)), \mathbf{indexof}, \mathrm{T}(\gamma_1), \mathrm{T}(\gamma_2))$:

$$\begin{aligned}\mathrm{T}'(\mathbf{indexof}(\gamma_1, \gamma_2)) = \{i \mid \exists s \in \text{prefixes}(\mathrm{T}(\gamma_1)), u \in \mathrm{T}(\gamma_2),\\ v \in \Sigma^* : suv \in \mathrm{T}(\gamma_1) \ \wedge \nexists s_1 \in \text{suffixes}(\text{prefixes}(s)) :\\ s_1 = u \wedge i = |s| \wedge i \in \mathrm{T}(\mathbf{indexof}(\gamma_1, \gamma_2))\}\end{aligned}$$

$\textsc{Restrict}(A, \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2)), \mathbf{substring}, \mathrm{T}(\gamma), \mathrm{T}(\beta_1), \mathrm{T}(\beta_1))$:

$$\begin{aligned}\mathrm{T}'(\mathbf{substring}(\gamma, \beta_1, \beta_2)) = \{s \mid \exists t \in \mathrm{T}(\gamma) : \exists t_1 \in \text{prefixes}(t),\\ t_2 \in \Sigma^* : t = t_1 t_2 \wedge |t_1| \in \mathrm{T}(\beta_1) \ \wedge \exists v \in \text{prefixes}(t_2) :\\ |v| \in \mathrm{T}(\beta_2) \wedge s = v \wedge s \in \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2))\}\end{aligned}$$

$\textsc{Restrict}(A, \mathrm{T}(\gamma_1 \cdot \gamma_2), \cdot, \gamma_1, \gamma_2)$:

$$\mathrm{T}'(\gamma_1 \cdot \gamma_2) = \{s \mid \exists s_1 \in \mathrm{T}(\gamma_1), s_2 \in \mathrm{T}(\gamma_2) : s = s_1 s_2 \wedge s \in \mathrm{T}(\gamma_1 \cdot \gamma_2)\}$$

Note that **charat** operation can be rewritten as $\mathbf{substring}(\gamma, \beta, 1)$ where the last parameter is the length of the substring, hence the Restrict and Refine for **charat** can be computed using corresponding operations for **substring**.

Let us now discuss the Refine procedure. Consider the string predicates =, **match**, and **contains**. Predicate operations create a boolean relation between the input tracks. We define the relation with tuples of strings that correspond to values from input tracks.
$\textsc{Refine}(A, =, \mathrm{T}(\gamma_1), \mathrm{T}(\gamma_2)) : \{(s, t) \mid s \in \mathrm{T}(\gamma_1) \wedge t \in \mathrm{T}(\gamma_2) \wedge s = t\}$
We can implement the semantics of the equality predicate using the multi-track DFAs precisely. Procedure Propagate must be called when tracks represent terms that include string term operations.
$\textsc{Refine}(A, \mathbf{match}, \mathrm{T}(\gamma), \mathrm{T}(\rho))$: $\{s \mid s \in \mathrm{T}(\gamma) \wedge s \in \mathrm{T}(\rho)\}$
Note that **match** operation takes a constant regular expression as an argument. We do not need to create a relation between a string term and a constant regular expression constant. Hence, we only refine the string term in the **match** predicate.
$\textsc{Refine}(A, \mathbf{contains}, \mathrm{T}(\gamma_1), \mathrm{T}(\gamma_2))$: $\{(s, t) \mid s \in \mathrm{T}(\gamma_1) \wedge t \in \mathrm{T}(\gamma_2) \ \wedge s \in \Sigma^*\mathrm{T}(\varphi_2)\Sigma^* \wedge t \in \text{suffixes}(\text{prefixes}(\mathrm{T}(\gamma_1))\}$
Here, semantics of the **contains** operation does not enforce the relation between the input tracks' values. In other words, if one of the tracks is updated by another operation, we need to propagate that update back to the **contains** operation. The Propagate procedure calls after conjunctions make sure that refinement for the **contains** operation is executed again once there is an update.

Next, we define the Refine semantics for the string term operations. Let us consider the operations **length**, **indexof**, **substring**, **charat**, and "·" again.
$\textsc{Refine}(A, \mathrm{T}(\mathbf{length}(\gamma)), \mathbf{length}, \mathrm{T}(\gamma))$:

$$\mathrm{T}'(\gamma) = \{s \mid \exists t \in \mathrm{T}(\mathbf{length}(\gamma)) : |s| = t \wedge s \in \mathrm{T}(\gamma)\}$$

$\textsc{Refine}(A, \mathrm{T}(\mathbf{indexof}(\gamma_1, \gamma_2)), \mathbf{indexof}, \mathrm{T}(\gamma_1), \mathrm{T}(\gamma_2))$:

$$\begin{aligned}\mathrm{T}'(\gamma_1) = \{s \mid \exists t, u, v \in \Sigma^* : |t| \in \mathrm{T}(\mathbf{indexof}(\gamma_1, \gamma_2)) \wedge\\ u \in \mathrm{T}(\gamma_2) \wedge s = tuv \wedge s \in \mathrm{T}(\gamma_1)\} \wedge\\ \mathrm{T}'(\gamma_2) = \{s \mid \exists t, v \in \Sigma^* : t \in \mathrm{T}(\mathbf{indexof}(\gamma_1, \gamma_2)) \wedge\\ tsv \in \mathrm{T}(\gamma_1) \wedge s \in \mathrm{T}(\gamma_2)\}\end{aligned}$$

$\textsc{Refine}(A, \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2)), \mathbf{substring}, \mathrm{T}(\gamma), \mathrm{T}(\beta_1), \mathrm{T}(\beta_1))$:

$$\begin{aligned}\mathrm{T}'(\gamma) = \{s \mid \exists t, u \in \Sigma^*, v \in \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2)) :\\ |t| \in \mathrm{T}(\beta_1) \wedge |v| \in \mathrm{T}(\beta_2) \wedge s = tvu \wedge s \in \mathrm{T}(\gamma)\} \wedge\\ \mathrm{T}'(\beta_1) = \{i \mid \exists t, u \in \Sigma^*, s \in \mathrm{T}(\gamma), v \in \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2)) :\\ |v| \in \mathrm{T}(\beta_2) \wedge s = tvu \wedge i = |t| \wedge i \in \mathrm{T}(\beta_1)\} \wedge\\ \mathrm{T}'(\beta_2) = \{i \mid \exists t, u \in \Sigma^*, s \in \mathrm{T}(\gamma), v \in \mathrm{T}(\mathbf{substring}(\gamma, \beta_1, \beta_2)) :\\ |t| \in \mathrm{T}(\beta_1) \wedge s = tvu \wedge i = |v| \wedge i \in \mathrm{T}(\beta_2)\}\end{aligned}$$

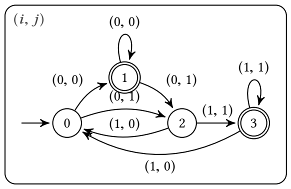


Figure 3: Automaton built for the constraint $\varphi_1 \equiv i = 2 \times j$

$\textsc{Refine}(A, \mathrm{T}(\gamma_1 \cdot \gamma_2), \cdot, \mathrm{T}(\gamma_1), \mathrm{T}(\gamma_2))$:

$$\begin{aligned}\mathrm{T}'(\gamma_1) = \{s \mid \exists t \in \mathrm{T}(\gamma_1 \cdot \gamma_2), v \in \mathrm{T}(\gamma_2) : t = sv \wedge s \in \mathrm{T}(\gamma_1)\} \wedge\\ \mathrm{T}'(\gamma_2) = \{s \mid \exists t \in \mathrm{T}(\gamma_1 \cdot \gamma_2), v \in \mathrm{T}(\gamma_1) : t = vs \wedge s \in \mathrm{T}(\gamma_2)\}\end{aligned}$$

The algorithms for the Restrict and Refine procedures are based on pre- and post-image computation in string analysis similar to the ones used in [2, 39, 40].

Let us consider the string constraint example $\mathbf{charat}(v, i) =$ "a" again. Initially $\mathrm{T}(v)$ and $\mathrm{T}(i)$ are unconstrained. Based on the semantics, $\textsc{Restrict}(A, \mathrm{T}(\mathbf{charat}(v, i)), \Sigma^*, \Sigma^*)$ computes the set for $\mathrm{T}'(\mathbf{charat}(v, i))$ as Σ^*. Next, the $\textsc{Refine}(A, =, \Sigma^*, \text{"a"})$ refines $\mathrm{T}(\mathbf{charat}(v, i))$ as {"a"}. Note that, we are not able keep the relation between $\mathbf{charat}(v, i)$, v, and i once they are computed. As equality predicate updates the $\mathrm{T}(\mathbf{charat}(v, i))$, we need to propagate the result back to v and i. In the final step, $\textsc{Refine}(A, \{\text{"a"}\}, \mathbf{charat}, \Sigma^*, \Sigma^*)$ is called to refine v and i. The final refinement sets the $\mathrm{T}(v)$ as Σ^*"a"Σ^* and $\mathrm{T}(i)$ as $\{i \mid i >= 0\}$.

3.3 Integer Constraint Solving

We now focus out attention to the branch of Algorithm 1 for $\alpha \equiv \alpha_1 \star \alpha_2$, when α is a $\varphi_{\mathbb{Z}}$ term, $\star$ is an integer term comparison operator, and α_1 and α_2 are linear combinations of atomic integer terms. Any such integer term constraint can be rewritten by moving all terms to one side of $\star$ and decomposing it into a semantically equivalent conjunction of constraints in which $\star$ is $\leq$. Thus, without loss of generality, we focus on integer term constraints of the form

$$\varphi_{\mathbb{Z}} \equiv 0 \leq \sum_{i=1}^{n} c_i \beta_i \tag{2}$$

where c_i denotes an integer constant coefficient and β_i is an atomic integer term.

Algorithm 1 is written in a way that it would process each binary + term separately. However, in the case of integer constraints of the form in expression 2, we construct a DFA for all terms of $\varphi_{\mathbb{Z}}$ at once. That is, when we call $\textsc{Refine}(A, \leq, \mathrm{T}(\beta_1), \ldots, \mathrm{T}(\beta_n))$ we use an automaton construction that updates all β_i tracks simultaneously. This automata construction method is based on algorithms that construct a binary adder state machine [9]. Given $\varphi_{\mathbb{Z}}$ as in expression 2, we use those algorithms to directly construct a multi-track automaton A over the binary alphabet $\{0, 1\}$ such that each track corresponds a β_i, and $\mathcal{L}(A)$ is the set of tuples of satisfying assignments for $(\beta_1, \ldots, \beta_n)$, encoded as binary integers in 2's complement form, reads from least to most significant bit.

For instance, consider the constraint $i = 2 \times j$ for integer variables i and j. The binary DFA for this constraint is depicted in Figure 3. One possible accepting sequence of states is 0, 2, 3, 0, 1. By taking the right-hand concatenation (as the DFA reads least significant bits first) of the pairs of bits along the corresponding transitions, we get

(0110, 0011) in binary which is (6, 3) in decimal. The DFA captures all possible integer solutions in this way, with leading 1's indicating negative numbers in the standard 2's complement encoding.

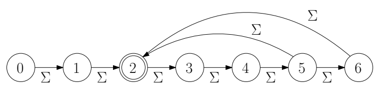
3.4 Binary and Unary Encodings

A string term can have integer sub-terms and a integer terms can contain string sub-terms. As described in the earlier discussion of Algorithm 1, we call Propagate, Refine, and Restrict to update the relationship between the string and integer variables. However, our binary integer arithmetic representation is not directly compatible with automaton operations over standard string automata.

As just described, we can precisely solve multi-variable linear integer arithmetic constraints by constructing a multi-track binary integer automaton that recognizes tuples of solutions. However, integer variable solutions can be related to string variables through operations that have both string and integer parameters such as **length** or **indexof**. Given the DFA representing the solutions for integer variables, we must propagate the constraints imposed by the integer solutions to each related string variable. We do so by first converting the binary DFA solution representation A for an integer variable i to a set comprehension representation S.

Our conversion from binary integer DFA A to a set comprehension S uses algorithms from [23, 24, 41], which show how to construct a description of a *semilinear set* from a binary DFA, which we now describe at a high level. A linear set L_i is given by $\{a_0 + a_1k_1 + \ldots a_nk_n : k_j \in \mathbb{Z}\}$ where the a_j constant integers are called the *periods* of the linear set. A semilinear set S is a finite union of linear sets, $S = \cup_i L_i$. For any binary integer DFA A constructed from linear integer arithmetic constraints, the accepted integers for each track of A form a semilinear set. Furthermore, for any track (which corresponds to an integer term), we can recover a set comprehension for the semilinear set S that it represents [23, 24, 41]. Intuitively, this works by examining the periods of the loops in the strongly connected components of the binary DFA in order to find the periods for a linear set $L_i \subseteq \mathcal{L}(A)$. A DFA representing the set L_i is then subtracted from A using DFA complement and intersection, and we iterate this procedure until $\mathcal{L}(A) = \emptyset$.

Once we have S, for a single track of the binary DFA A, which corresponds to a single integer terms, we then convert S into a unary DFA A', which for any integer $m \in S$ accepts all strings of length m. The unary DFA A' is then compatible with string automata and can be used to restrict or refine the set of string models. For example, if $S = \{2 + 5k_1 + 4k_2\}$ the corresponding unary DFA is shown below, which has an initial segment of length 2 and two interleaved loops of periods 4 and 5.



We described how to propagate solutions from binary integer DFA to string DFA. In order to propagate solutions from string DFA to binary integer DFA, we reverse this process by converting a string DFA into a unary length DFA, extracting the semilinear set, and constructing the corresponding binary integer DFA.

Consider the following example constraint:

$$i = 2 \times j \wedge \textbf{length}(v) = i \tag{3}$$

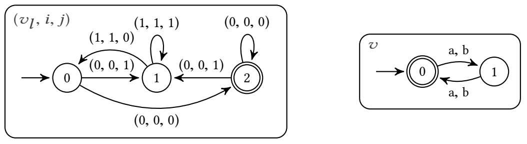


Figure 4: Final automata constructed for Example 3

Example 3 is a conjunction of an atomic integer constraints $\varphi_1 \equiv i = 2 \times j$ and $\varphi_2 \equiv \textbf{length}(v) = i$. The constraint φ_2 is also a mixed constraint as it contains both a string and an integer variable.

Figure 4 shows the final automata constructed for the input formula $\varphi \equiv i = 2 \times j \wedge \textbf{length}(v) = i$. The auxiliary variable v_l represents bitwise encodings of the lengths of the strings that are represented with the variable v.

4 MODEL COUNTING

In this section, we describe how to perform parameterized model counting by making use of the automata constructed by our constraint solving procedure. The *model counting problem* is to determine the size of $[\![\varphi]\!]$, which we denote $\#[\![\varphi]\!]$. While a formula can have infinitely many models, we can count the number of models in an infinite space of solutions restricted to a finite range for the free variables. Hence, we perform *parameterized model counting* for string and integer constraints, where $\#[\![\varphi]\!](b_{\mathbb{S}}, b_{\mathbb{Z}})$ is a function over parameters $b_{\mathbb{S}}$, which bounds the length of string solutions, and $b_{\mathbb{Z}}$, which bounds the bit-length representation of integer solutions.

The constraint solving procedure produces a final DFA, A, that contains multi-track solution sub-automata $A_{\mathbb{S}}$ and $A_{\mathbb{Z}}$. The separation of string and integer automata may lose some relational information between string and integer variables; we can multiply the model counts for each automaton in order to give a sound upper bound on the number of models for tuples of integer and string variables. We use functions $\#F_{A_{\mathbb{S}}}(k)$ and $\#f_{A_{\mathbb{Z}}}(k)$ to count string and integer models respectively.

We rely on the observation that counting the number of strings of length k in a regular language, $\mathcal{L}$, is equivalent to counting the number of accepting paths of length k in the DFA that accepts $\mathcal{L}$. That is, by using a DFA representation, we reduce the parameterized model counting problem to counting the number of paths of a given length in a graph. In a DFA, there is exactly one accepting path for every recognized string. Thus, if we are interested in computing only string models or only integer models, there is no loss of precision due to the the model counting procedure; any loss of precision for strings comes from the over-approximations of non-regular constraints in the solving phase, and for pure integer constraints, the model counting procedure is precise because integer solution automata construction is precise.

Given a string automaton $A_{\mathbb{S}}$, computation of $\#f_{A_{\mathbb{S}}}(k)$, the number of accepted strings of length k, can be done by constructing the transfer matrix of the automaton based on its transition relation [31, 35]. Let $A_{\mathbb{S}}$ be a DFA with n states. The transfer matrix T of A is a matrix where $T_{i,j}$ is the number of transitions from state i to state j. The number of paths of length k that start in state i and end in state j is given by $(T^k)_{i,j}$. Then the number of strings of length k accepted by A can be computed using matrix multiplication. We compute uT^kv, where u is the row vector such that $u_i = 1$ if and

$$\begin{array}{rrr}
\varphi \wedge \varphi \rightarrow \varphi & \varphi \vee \varphi \rightarrow \varphi & \varphi \vee \top \rightarrow \top \\
\varphi \wedge \top \rightarrow \varphi & \varphi \vee \bot \rightarrow \varphi & \varphi \wedge \bot \rightarrow \bot \\
0 \times \beta \rightarrow 0 & 1 \times \beta \rightarrow \beta & \beta + 0 \rightarrow \beta \\
\beta - 0 \rightarrow \beta & \beta = \beta \rightarrow \top & i \neq j \rightarrow \top \\
-(-\beta) \rightarrow \beta & \neg(\neg\beta) \rightarrow \beta & \beta \neq \beta \rightarrow \bot \\
i = j \rightarrow \bot & |\epsilon| \rightarrow 0 & |v_{s_1} . v_{s_2}| \rightarrow |v_{s_1}| + |v_{s_2}| \\
\gamma . \epsilon \rightarrow \gamma & \epsilon . \gamma \rightarrow \gamma & \mathbf{match}(\gamma, t) \rightarrow \gamma = t \\
\gamma = \gamma \rightarrow \top & \gamma_1 . t = \gamma_2 . v \rightarrow \bot & \mathbf{contains}(\gamma_2 . \gamma_1 . \gamma_3, \gamma_1) \rightarrow \top \\
\gamma \neq \gamma \rightarrow \bot & \gamma_1 . t \neq \gamma_2 . v \rightarrow \top & \mathbf{begins}(\gamma_1 . \gamma_2, \gamma_1) \rightarrow \top \\
t = v \rightarrow \bot & t . \gamma_1 = v . \gamma_2 \rightarrow \bot & \mathbf{ends}(\gamma_2 . \gamma_1, \gamma_1) \rightarrow \top \\
t \neq v \rightarrow \top & t . \gamma_1 \neq v . \gamma_2 \rightarrow \top & t . \gamma_1 = t . \gamma_2 \rightarrow \gamma_1 = \gamma_2 \\
t_1 . t_2 \rightarrow t_1 t_2 & & \gamma_1 . t = \gamma_2 . t \rightarrow \gamma_1 = \gamma_2
\end{array}$$

Figure 5: Term reduction rules

only if i is the start state and 0 otherwise, and v is the column vector where $v_i = 1$ if and only if i is an accepting state and 0 otherwise. Note that for relational string constraints, the transition alphabet is over tuples of characters and the method described here will count the number of tuples of solutions of a given length. Our counting method is parameterized in the following sense: after a constraint is solved, we can count the number of solutions of any desired size k by computing $uT^k v$, without re-solving the constraint.

The method described above computes $\#f_{A_\mathbb{S}}(k)$, the number of string solutions of length *exactly* k. It is of interest to compute $\#F_{A_\mathbb{S}}(k)$, the number of solutions *within* a given bound. This is accomplished easily using a known “trick” often used to simplify graph algorithms. We add an artificial accepting state s_{n+1} to $A_\mathbb{S}$, resulting in a new DFA A'_S, with λ-transitions from each accepting state to s_{n+1}, and a λ-cycle on s_{n+1}. Then one can see that $\#F_{A_\mathbb{S}}(k) = \#f_{A'_S}(k+1)$, and so we apply the transfer matrix method on A'_S.

The method for counting strings of a given length allows us to perform model counting for linear constraints as well. However, we must interpret the bound k in a slightly different manner. A solution DFA $A_\mathbb{Z}$ for a set of integer tuples encodes the solutions as bit-strings. Thus, paths of length k in an integer automaton correspond to bit string of length k. Since we are using a 2’s complement representation with leading sign bits, bit strings of exactly length k correspond to integers in the range $[-2^{k-1}, 2^{k-1})$. Thus, the transfer matrix method allows us to perform model counting over integer domains parameterized by intervals of this form by computing $\#f_{A_\mathbb{Z}}(k)$. To count models for arbitrary intervals (a, b), we intersect $A_\mathbb{Z}$ with the DFA representing $a \leq x_i \leq b$ for any variable x_i, and then count paths in the resulting DFA.

The methods described above allow us to compute $\#F_{A_\mathbb{S}}(k)$ and $\#f_{A_\mathbb{Z}}(k)$ independently. Now, we can compute $\#\varphi(b_S, b_\mathbb{Z}) = \#F_{A_\mathbb{S}}(b_S) \cdot \#f_{A_\mathbb{Z}}(b_\mathbb{Z})$.

5 CONSTRAINT SIMPLIFICATION

We use several heuristics to simplify the constraints before automata-construction and model counting steps.

Term Re-Write Rules: All terms are first reduced with respect to a re-write system based on a set of rules (Fig. 5). These rules include both term normalization rules and tautological simplifications of atomic constraints. Here, i, j are distinct integer constants, t, v are distinct string constants and $\gamma_1, \gamma_2, \gamma_3$ are (not necessarily distinct) string terms.

Dependency Analysis: To reduce the amount of work required to solve a constraint, we note that not all variables of a constraint

$$\begin{array}{rr}
\gamma_1 . \mathbf{begins}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| & \gamma_1 . \mathbf{contains}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| \\
\gamma_1 . \mathbf{ends}(\gamma_2) \rightarrow |\gamma_1| \geq |\gamma_2| & \neg\gamma . \mathbf{ends}(t) \rightarrow \gamma \neq t \\
\neg\gamma . \mathbf{contains}(t) \rightarrow \neg\gamma . \mathbf{begins}(t) & \\
\gamma_1 . \gamma_2 = \gamma_3 . \gamma_4 \rightarrow \quad |\gamma_1| + |\gamma_2| = |\gamma_3| + |\gamma_4| & \\
\gamma_1 . \gamma_2 = \gamma_3 \rightarrow \quad |\gamma_1| + |\gamma_2| = |\gamma_3| \wedge \gamma_3 . \mathbf{begins}(\gamma_1) &
\end{array}$$

Figure 6: Implication rules

need be counted together. We define the *constraint graph* of a formula φ to be the graph defined on the set of variables of φ where an edge exists between any two variables if they appear in the same clause of φ. This constraint graph can be decomposed into a finite set of connected components. A connected component C is a maximal subgraph such that if $u, v \in C$ then there exists a path between u and v in C. Constraints on any given variable depend only on variables within its connected component. This allows us to decompose a formula based on connected components, solve and count each component individually, and then take the product of the results to obtain accurate counts for tuples of variables. This results in smaller automata and faster computation.

Equivalence Classes: When no disjunctions are present, the variables of a formula φ can be partitioned into equivalence classes so that any pair of given variables x, y are in the same equivalence class only if they have the same solution set. In our implementation, we construct these equivalence classes based on equality clauses. Every term, variable or otherwise, begins in its own equivalence class and for every equality clause, the equivalence classes of the left and right sides are merged. From each equivalence class a representative is chosen. Each variable in the equivalence class is then replaced by this representative in the formula φ. This optimization can result in the elimination of variables from φ, and hence tracks from its DFA, without any loss of precision in counting.

Term Elimination via Substitution: Constraints generated from symbolic execution in the presence of loops result in the addition of many intermediate variables and terms, usually due by loop unrolling. These intermediate terms form a continuous link between the state of variables before and after the loop body, represented as conjunctions between word equations. If the variables do not appear elsewhere in the constraint formula, we collapse this chain into a single word equality.

Implication Rules: As noted previously, our automata construction for some constraints can be imprecise. Precision can be improved for some of these constraints by augmenting the original formula φ with clauses implied by φ. We present a set of implication rules which define the augmenting clauses added to φ in the presence of certain imprecise constraints in Fig. 6. We only add a clause to φ if we can solve it precisely and if it can potentially improve the precision for another constraint. Implications on string variables appearing in multiple word equations under the same conjunction are combined into a single implication whenever possible.

6 IMPLEMENTATION AND EXPERIMENTS

We implemented the techniques we presented in this paper in a tool called Multi-Track Automata Based model Counter (MT-ABC)[1] by extending an existing tool called Automata Based Model Counter (ABC). We evaluated the precision and performance of MT-ABC using three types of constraints: constraints solely on string

[1]available at https://github.com/vlab-cs-ucsb/ABC

variables, constraints solely on integer variables, and constraints that contain both string and integer variables.

We experimentally compared MT-ABC with five existing model counting constraint solvers: (1) ABC [4], a single-track automata-based model counter for strings, (2) S3# [38], a model counter for strings with some capability of handling relations between strings and integers, (3) SMC [27], a string model counter, (4) LattE [7, 26], a model counter for linear arithmetic constraints, and (5) SMTApproxMC [14], an approximate model counter for the theory of fixed-width words.

Our experiments show: 1) MT-ABC is as or more precise with comparable efficiency than existing string model counters. 2) Multi-track automata enables MT-ABC to capture relations between variables more precisely than single-track automata used in ABC. 3) Parameterized model counting enables MT-ABC to compute multiple length bounds for the same formula efficiently without re-solving. 4) MT-ABC and S3# are the only tools that support mixed constraints with string and integer variables. MT-ABC is as or more precise than S3# for model counting constraints involving relations between string and integer variables, MT-ABC can handle a richer set of constraints than S3#, and S3# produces unsound results.

All experiments, other than those involving S3#, were done on an Ubuntu 16.04 machine with Intel i5 3.5GHz X4 processors and 32GB of memory. We were unable to run S3# on Ubuntu 16.04; all experiments involving S3# were done on the same machine but within an Ubuntu 14.04 virtual machine with 8 GB of memory.

6.1 String Constraints

Security Benchmark: Constraints in this benchmark are taken from various security contexts [27, 38]. For example, two constraints extracted from string manipulation utilities within the BUSYBOXY v.1.21.1 package (wc and grep), and one constraint extracted from a utility in the COREUTILS v.8.21 package (csplit) are used to quantify information leakage for homomorphically encrypted inputs.

Table 1 shows the results of MT-ABC, ABC, S3#, SMC for the security benchmark. Second column shows the string length value used for model counting (i.e., the tools count the number of solution strings with the specified length), last column indicates scale for larger lengths. Both MT-ABC and ABC report an upper bound on the number of solutions, while both SMC and S3# give both lower and upper bounds (S3# reports an exact count when the bounds are the same). Both MT-ABC and S3# generate bounds which are as or more precise than those reported by SMC. In all cases, MT-ABC is as or more precise than ABC. The bounds generated by both MT-ABC and S3# agree for all constraints except ghttpd and ghttpd_ wo_len, where ghttpd_ wo_len is derived from ghttpd by removing the part of the constraint that uses the string length function. For solution strings of length 620, the two solvers give different counts. We could not confirm the model count for these constraints as they are too complex to manually count. However, while experimenting with variations of these constraints, we found out that S3# computes an erroneous count for a simplified version of these constraints. So, we believe that the count that S3# reports is erroneous.

The running times for all four model counters are comparable for small constraints (obscure, strstr, regex, contains). For large constraints (ghttpd, wc, csplit, nullhttpd), ABC either times out after 20 minutes or runs out of memory, while both MT-ABC and S3# produce results faster than SMC. When the input constraint contains a high concrete value for the string length (ghttpd, wc, grep), MT-ABC generates a large automaton, which leads to a higher running time, whereas without the length constraint (ghttpd_wo_len), both MT-ABC and ABC produce results quickly.

Simplified Kaluza Benchmark: Simplified Kaluza benchmark is a set of satisfiable constraints generated via symbolic execution of JavaScript and originally solved with the Kaluza string solver [33]. The authors of SMC simplified the Kaluza benchmark by replacing integer variables with constants and by removing disjunctions, since SMC cannot handle integer variables and loses precision for disjunctive constraints. Then, they translated these constraints into their input format and separated them into two sets: SMCSmall and SMCBig. We translated them from SMC format to MT-ABC input format. The SMCSmall set contains 17544 constraints and SMCBig contains 1342 constraints. Each constraint contains a query variable to model count on. We compared the performance and upper bounds produced by MT-ABC, ABC, and SMC using this benchmark.

Table 2 compares MT-ABC to ABC and MT-ABC to SMC for solution strings less than or equal to 50. We did not include S3# in this comparison since S3# can only model count solution strings having length exactly equal to the given given length.

For SMCSmall constraints ABC takes 0.0036s per constraint, SMC takes 0.42s per constraint, and MT-ABC takes 0.011s per constraint, on average. For SMCBig constraints ABC takes 6.09s per constraint, SMC takes 4.08s per constraint, and MT-ABC takes 1.35s per constraint, on average. For SMCSmall constraints, MT-ABC generates a more precise count than ABC for 6% of the constraints, and MT-ABC generates a more precise count than SMC for 0.9% of the constraints. For SMCBig constraints, MT-ABC generates a more precise count than ABC for 78% of the constraints, and MT-ABC generates a more precise count than SMC for 75.9% of the constraints. MT-ABC reported a higher count than SMC for one constraint; we manually determined MT-ABC reports the exact count in this case and concluded that the count reported by SMC is erroneous. In summary, for small constraints the performance of all three solvers are comparable, but for big constraints, MT-ABC is more efficient than ABC and SMC and produces more precise counts than ABC and SMC for more than 75% of the big constraints.

6.2 Integer Constraints

Comparison with LattE: We compare the performance of MT-ABC with LattE for model counting linear arithmetic constraints on benchmarks containing constraints generated during reliability [15] and side-channel analyses of Java programs using the symbolic execution tool SPF [6, 8]. We extended the reliability benchmark by adding Merge sort, Quick sort, and Binary search functions. Password, LawDB, and CRIME come from side-channel analysis [6, 8]. Password, LawDB and Binary have 7,8, and 13 constraints respectively; the others range from 600-2000 constraints each.

Some of the constraints (e.g., the constraints coming from the sorting functions) require a data structure with a certain size in order to enable symbolic execution. We fixed the size of such structures to 6. We counted solutions to the path constraints given bit-lengths 4, 8, 16, and 32. MT-ABC and LattE return identical counts

Table 1: Experiments with MT-ABC, ABC, S3#, and SMC on security benchmark. Unsound results are highlighted.

Program	Len	SMC		ABC		MT-ABC		S3#		
		Lower/Upper Bound	Time	Upper-Bound	Time	Upper-Bound	Time	Exact Count	Time	Count Scale
ghttpd	620	[10626.2;1031904473.2]	26.07	–	–	1031904473	21.69	1031904472.8	0.54	$\times 10^{1465}$
	11	[256;767]	0.49	767	0.56	767	0.029	767	0.49	
ghttpd wo_len	620	[10626.2;1031904473.2]	25.99	1031904473	0.55	1031904473	0.14	1031904472.8	0.52	$\times 10^{1465}$
	11	[256;767]	0.49	767	0.57	767	0.069	767	0.49	
nullhttpd	500	[2.9;1369.8]	9.78	–	–	0	0.032	0	0.47	$\times 10^{1129}$
csplit	629	$[5.9 * 10^{1460};3.1 * 10^{1481}]$	98.01	–	–	0	0.024	0	0.54	
grep	629	$[0.7 * 10^{1408};0.1 * 10^{1435}]$	150.97	$2.0 * 10^{1473}$	5.1	0	3.94	0	0.56	
wc	629	[0.979;8.0]	153.93	–	–	0.979	9.05	0.979	3.35	$\times 10^{1289}$
obscure1	10	[11.2;11.6]	0.45	11.2	0.013	11.2	0.023	11.2	0.46	$\times 10^{23}$
obscure2	6	[2.8;2.8]	0.47	2.8	0.075	2.8	0.077	2.8	0.46	$\times 10^{14}$
strstr1	5	[196608;196608]	0.45	1099511431168	0.017	1099511431168	0.002	1099511431168	0.45	
strstr2	5	[16776960;16776960]	0.45	16776960	0.026	16776960	0.004	16776960	0.46	
regex	4	[0;0]	0.52	16	0.004	16	0.002	16	0.45	
contains	5	[67108096;67108096]	0.45	67108096	0.007	67108096	0.002	67108096	0.46	

Table 2: ABC (u_{ABC}), MT-ABC ($u_{\text{MT-ABC}}$) and SMC (u_{SMC}) upper bounds comparison.

Benchmark	#Constraints	$u_{\text{MT-ABC}} < u_{\text{SMC}}$	$u_{\text{MT-ABC}} = u_{\text{SMC}}$	$u_{\text{MT-ABC}} > u_{\text{SMC}}$
SMCSmall	17544	166 (0.9%)	17388 (99.1%)	1 (0.0%)
SMCBig	1342	1019 (75.9%)	323 (24.1%)	0 (0.0%)
		$u_{\text{MT-ABC}} < u_{\text{ABC}}$	$u_{\text{MT-ABC}} = u_{\text{ABC}}$	$u_{\text{MT-ABC}} > u_{\text{ABC}}$
SMCSmall	17544	1025 (6%)	16529 (94%)	0 (0.0%)
SMCBig	1342	1046 (78%)	296 (22%)	0 (0.0%)

Table 3: MT-ABC and LattE average time (seconds) per numeric constraint for different bit-lengths. The last two columns denote the combination of all lengths (columns for lengths 4,16 omitted for space). For each bit-length, the execution time of the faster tool is in bold.

Benchmark	Bit-length = 8		Bit-length = 32		Bit-length = 4,8,16,32	
	MT-ABC	LattE	MT-ABC	LattE	MT-ABC	LattE
LawDB	0.0218	**0.0118**	0.0223	**0.0144**	**0.0227**	0.0408
Heap	**0.0165**	0.0214	**0.0209**	0.0217	**0.0212**	0.0868
Booking	**0.0104**	0.0133	**0.0106**	0.0133	**0.0107**	0.0534
Bubble	**0.0184**	0.0218	0.0264	**0.0221**	**0.0268**	0.0879
Binary	**0.0246**	0.0250	0.0409	**0.0256**	**0.0410**	0.1036
DaisyChain	**0.0128**	0.0359	**0.0138**	0.0361	**0.0140**	0.3571
Selection	**0.0171**	0.0217	0.0224	**0.0219**	**0.0228**	0.0878
Crime	**0.0143**	0.2628	**0.0151**	0.2604	**0.0153**	0.9873
Merge	**0.0183**	0.0215	0.0262	**0.0216**	**0.0266**	0.0868
Flap	**0.0094**	0.0308	**0.0094**	0.0308	**0.0096**	0.1234
Quick	**0.0173**	0.0219	0.0236	**0.0224**	**0.0239**	0.0891
Insertion	**0.0190**	0.0218	0.0270	**0.0220**	**0.0273**	0.0880
RobotGame	**0.0113**	0.1408	**0.0113**	0.1397	**0.0114**	0.5717
AlarmClock	**0.0095**	0.0121	**0.0096**	0.0121	**0.0097**	0.0487
Password	**0.0102**	0.0542	**0.0102**	0.0542	**0.0102**	0.2185

for all constraints as both model counters are precise in counting linear arithmetic constraints. We focus on the timing comparison between MT-ABC and LattE. As a side note, the LattE input format does not support disequalities and thus needs a preprocessing step when such constraints arise. The LattE integration with SPF uses Omega [20]; we refer the reader to [6, 8, 15] for integration details.

Figure 3 shows that in general MT-ABC performs better than LattE, though there are exceptions (such as LawDB, Binary). Note that MT-ABC always outperforms LattEwhen counting multiple bit-lengths. Since MT-ABC is a parameterized model counter, it first solves a constraint without constraints on bit length, and then reuses the generated automaton to count for multiple bit-lengths. In contrast, LattE needs to be called separately for each bit-length.

Table 4: MT-ABC and SMTApproxMC average time (seconds) per numeric constraint for different bit-lengths. For each bit-length, the execution time of the faster tool is in bold.

Benchmark	Bit-length = 2		Bit-length = 3	
	MT-ABC	SMTApproxMC	MT-ABC	SMTApproxMC
Bubble	**0.011**	0.502	**0.011**	1.046
Booking	**0.019**	0.530	**0.018**	24.09
Selection	**0.017**	0.518	**0.017**	14.29
Password	**0.011**	47.28	**0.011**	1680.67
Merge	**0.018**	0.528	**0.018**	24.08
FlapController	**0.793**	1.487	**0.791**	158.81
Binary	**0.009**	0.656	**0.009**	4.525
Insertion	**0.019**	0.531	**0.019**	24.05
Heap	**0.017**	0.513	**0.017**	10.33
Quick	**0.017**	0.521	**0.017**	16.97
Alarm	**0.009**	0.472	**0.010**	0.99

Comparison with SMTApproxMC: We compare the performance of MT-ABC with SMTApproxMC using the same program analysis benchmarks we used in comparison of MT-ABC with LattE. Since SMTApproxMC targets the theory of fixed-width words, we translated each benchmark into the SMT2-BitVector format that SMTApproxMC is able to handle. We ran both MT-ABC and SMTApproxMC using bit-lengths of 2 and 3 since SMTApproxMC does not scale to larger bit-lengths. As some of the benchmarks contains constants which require more than 2 or 3 bits to be represented in bitvector format, we omit them from our comparison. Table 4 shows the execution time of both tools. For both bit-lengths and all benchmarks, MT-ABC is significantly faster than SMTApproxMC. MT-ABC produces an exact count in every case, while SMTApproxMC reports an approximate count which varies in precision. The average difference in model count as percentage of the domain size between the two tools is 3.7% and 4.2%, for bit-lengths of 2 and 3, respectively, with a maximum difference of 23.4% and 25.7% for bit-lengths of 2 and 3 for the FlapController. For every constraint in these benchmarks, MT-ABC significantly outperforms SMTApproxMC while producing as or more precise counts.

6.3 Mixed String and Integer Constraints

For our final tool comparison we use the unmodified SMT2 Kaluza benchmark, used in [36], which requires constraint solvers to reason about constraints over mixed string and integer variables. In [38] this benchmark was used by the authors to demonstrate that S3# can handle mixed string and integer constraints. However, for these constraints, no model counting was performed, only a satisfiability check was done in [38]. When we used S3# to model count (by giving a string length) we found out that S3# reported erroneous results for many constraints.

We focused on a subset of the SMT2 Kaluza benchmark. We compared MT-ABC and S3# on 28059 of the smaller constraints within the benchmark, given a query variable and a string length bound of 50 (solutions for the query variable must have an exact length of 50 characters). MT-ABC and S3# agree on 24317 (87%) of the constraints. In each of these cases, S3# was able to give an exact count, matching the upper bound given by MT-ABC. In the other 3742 (13%) constraints, S3# reported both a lower and upper bound, neither of which matched the upper bound reported by MT-ABC.

For the constraints where MT-ABC and S3# produce different counts, the lower bound reported by S3# was between 1-3 models, while the upper bound seemed entirely random, fluctuating either below or above the count reported by MT-ABC. In the SMT2 Kaluza benchmark, there are many sets of constraints which are essentially equivalent to each other, some differing only in variable naming. We manually confirmed the upper bound returned by MT-ABC for many of the constraints was the exact count, while the upper bound reported by S3# between identical constraints varied wildly, with many of them being unsound. Additionally, we found that S3# gives different results for identical files with different names. Consider a constraint from the SMT2 Kaluza benchmark, $\mathbf{length}(s) = i$, where s is an string variable, i an integer variable. We created three files each containing this single constraint, differing only in name. For query variable s and query length 5, the number of models is $256^5 = 1099511627776$, or 2^{40}. While MT-ABC gives the exact count for all three files, S3# reports three *different* upper bounds, all unsound (1.8401^{33}, 1.8567^{30}, 1.8554^{26}). We observed similar behavior from S3# given different constraints from the Kaluza dataset.

We reached out to the developer of S3# ([38]) for a possible explanation. One issue is that they assumed that constraints from the Kaluza data set could be transformed into their solved form, but they did not verify this, nor the soundness of their results for this dataset in [38]. Thus, it is possible that either the Kaluza constraints cannot all be transformed into solved form, or S3# has a faulty implementation. Additionally, the authors of S3# were unable to explain why their tool was producing non-deterministic unsound upper bounds when the input constraint cannot be transformed into their solved form. Our experiments suggest that the techniques presented in [38] and their implementation in S3# are not able to handle mixed numeric and string constraints with both string and integer variables. Hence, to the best of our knowledge, MT-ABC is the only model counting constraint solver that can handle this class of constraints.

7 RELATED WORK

There has been significant amount of work on string constraint solving in recent years [1, 16, 18, 19, 22, 25, 33, 36, 37, 43]; however none of these solvers provide model-counting functionality. Meanwhile, due to the importance of model counting for quantitative program analyses, model counting constraint solvers are gaining increasing attention. SMC and S3# are model-counting constraints solvers for string constraints [27, 38]. Our model counting approach is more precise and more expressive than SMC since SMC cannot propagate string values across logical connectives and cannot handle complex string operations such as *replace*. S3# handles string constraints involving length constraints, but suffers a severe loss in precision when length constraints include symbolic integers. Although the expressiveness of S3# is comparable to that of MT-ABC for string constraints, unlike MT-ABC S3# cannot handle pure numeric constraints, and it produces unsound results for mixed constraints.

LattE [7] is a model counting constraint solver for linear integer arithmetic. LattE uses the polynomial-time Barvinok algorithm [10] for integer lattice point enumeration. LattE cannot handle string constraints, so our approach is more expressive than LattE.

Automata-based constraint solving and model counting techniques we use in this paper are not domain-specific like the approaches used in LattE, SMC, and S3# but general in the sense that, they can handle any set of constraints that can be mapped to automata. As we present in this paper, it is possible to map both numeric and string constraints and their combinations to automata.

While linear algebraic methods for counting paths in a graph are well established, this paper is the first to implement those methods for the purpose of parameterized model counting for relational string and integer constraints. There has been earlier work on integer constraint model counting by counting paths in numeric DFA [28], but this earlier approach can only count models when there are finitely many models. We built MT-ABC by extending an existing tool called Automata Based model Counter (ABC) [4]. ABC uses a single-track automata representation. ABC cannot model count relational constraints and numeric constraints as precisely as MT-ABC, and it cannot handle constraints with integer variables. ABC has been integrated with Symbolic PathFinder (SPF) and applied to side-channel analysis in [8].

SMTApproxMC [14] is a model counting constraint solver for the theory of fixed-width words, and it uses a different approach for model-counting based on solution sampling [13]. Since SMTApproxMC cannot handle string constraints, we compared SMTApproxMC with MT-ABC on a set of numeric constraints. MT-ABC produces precise counts for linear arithmetic constraints whereas SMTApproxMC can only produce approximations, and our experiments demonstrate that MT-ABC is significantly faster.

8 CONCLUSION

Model counting is a crucial problem in quantitative program analysis. Using automata as a representation for all solutions of a given constraint reduces the model counting problem to path counting. In this paper, we show that, using automata-based constraint solving, one can construct a model counting constraint solver that is able to handle both string and numeric constraints and their combinations. Our experiments on a large set of constraints generated from Java and JavaScript programs indicate that, automata-based model counting approach is as efficient and as precise as domain specific model counting methods, while it is able to handle a richer set of constraints than any other model counting constraint solver.

REFERENCES

[1] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Lukás Holík, Ahmed Rezine, Philipp Rümmer, and Jari Stenman. 2014. String Constraints for Verification. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*. 150–166.

[2] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. [n. d.]. Semantic Differential Repair for Input Validation and Sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. New York, NY, USA, 225–236. https://doi.org/10.1145/2610384.2610401

[3] Abdulbaki Aydin. 2017. *Automata-based Model Counting String Constraint Solver for Vulnerability Analysis.* dissertation. University of California Santa Barbara. https://www.alexandria.ucsb.edu/lib/ark:/48907/f3xp754k

[4] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I.* 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

[5] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA.* 141–153.

[6] Daniel Balasubramanian, Kasper Luckow, Corina Pasareanu, Abdulbaki Aydin, Lucas Bang, Tevfik Bultan, Miroslav Gavrilov, Temesghen Kahsai, Rody Kersten, Dmitriy Kostyuchenko, Quoc-Sang Phan, Zhenkai Zhang, and Gabor Karsai. 2017. ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code. In *submission.*

[7] V. Baldoni, N. Berline, J.A. De Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu. [n. d.]. LattE integrale v1.7.2. http://www.math.ucdavis.edu/ latte/. ([n. d.]).

[8] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016).* ACM, New York, NY, USA.

[9] Constantinos Bartzis and Tevfik Bultan. 2003. Efficient Symbolic Representations for Arithmetic Constraints in Verification. *Int. J. Found. Comput. Sci.* 14, 4 (2003), 605–624.

[10] Alexander I. Barvinok. 1994. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension is Fixed. *Math. Oper. Res.* 19, 4 (1994), 769–779. https://doi.org/10.1287/moor.19.4.769

[11] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Pasareanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015.* 866–877.

[12] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Pasareanu. 2017. Model-Counting Approaches for Nonlinear Numerical Constraints. In *Proceedings of the 9th International NASA Formal Methods Symposium.* 131–138.

[13] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-Aware Sampling and Weighted Model Counting for SAT. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence.* 1722–1730.

[14] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. 2016. Approximate Probabilistic Inference via Word-Level Counting. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.* 3218–3224.

[15] Antonio Filieri, Corina S. Pasareanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013.* 622–631.

[16] Vijay Ganesh, Mia Minnes, Armando Solar-Lezama, and Martin C. Rinard. 2012. Word Equations with Length Constraints: What's Decidable?. In *Proceedings of the 8th International Haifa Verification Conference (HVC).* 209–226.

[17] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012.* 166–176.

[18] Pieter Hooimeijer and Westley Weimer. 2009. A decision procedure for subset constraints over regular languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 188–198.

[19] Pieter Hooimeijer and Westley Weimer. 2010. Solving String Constraints Lazily. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 377–386.

[20] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. 1996. The Omega calculator and library, version 1.1. 0. *College Park, MD* 20742 (1996), 18.

[21] John Kelsey. 2002. Compression and Information Leakage of Plaintext. In *Fast Software Encryption, 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002, Revised Papers.* 263–276. https://doi.org/10.1007/3-540-45661-9_21

[22] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA).* 105–116.

[23] Louis Latour. 2004. From Automata to Formulas: Convex Integer Polyhedra. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings.* 120–129. https://doi.org/10.1109/LICS.2004.1319606

[24] Jérôme Leroux. 2005. A Polynomial Time Presburger Criterion and Synthesis for Number Decision Diagrams. In *LICS.* 147–156.

[25] Guodong Li and Indradeep Ghosh. 2013. PASS: String Solving with Parameterized Array and Interval Automaton. In *Proceedings of the 9th International Haifa Verification Conference (HVC).* 15–31.

[26] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation* 38, 4 (2004), 1273 – 1302. https://doi.org/10.1016/j.jsc.2003.04.003

[27] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. 2014. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 57.

[28] Erin Parker and Siddhartha Chatterjee. 2004. An Automata-Theoretic Algorithm for Counting Solutions to Presburger Formulas. In *Compiler Construction, 13th International Conference, CC 2004, Barcelona, Spain.* 104–119. https://doi.org/10.1007/978-3-540-24723-4_8

[29] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo d'Amorim. 2014. Quantifying information leaks using reliability analysis. In *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA.* 105–108.

[30] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. 2012. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[31] Bala Ravikumar and Gerry Eisman. 2004. Weak minimization of DFA - an algorithm and applications. *Theor. Comput. Sci.* 328, 1-2 (2004), 113–133. https://doi.org/10.1016/j.tcs.2004.07.009

[32] Juliano Rizzo and Thai Duong. 2012. The CRIME attack *(Ekoparty Security Conference).*

[33] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy.*

[34] Geoffrey Smith. 2009. On the Foundations of Quantitative Information Flow. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, York, UK, March 22-29, 2009. Proceedings.* 288–302.

[35] Richard P. Stanley. 2011. *Enumerative Combinatorics: Volume 1* (2nd ed.). Cambridge University Press, New York, NY, USA.

[36] Cesare Tinelli Clark Barrett Morgan Deters Tianyi Liang, Andrew Reynolds. 2014. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, Proceedings.* 646–662.

[37] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS).* 1232–1243.

[38] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2017. Model Counting for Recursively-Defined Strings. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, Proceedings, Part II.* 399–418.

[39] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. 2009. Generating Vulnerability Signatures for String Manipulating Programs Using Automata-Based Forward and Backward Symbolic Analyses. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09).* IEEE Computer Society, Washington, DC, USA, 605–609. https://doi.org/10.1109/ASE.2009.20

[40] Fang Yu, Muath Alkhalaf, Tevfik Bultan, and Oscar H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design* 44, 1 (2014), 44–70. https://doi.org/10.1007/s10703-013-0189-1

[41] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2009. Symbolic String Verification: Combining String Analysis and Size Analysis. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, (TACAS '09).* Springer-Verlag, Berlin, Heidelberg, 322–336. https://doi.org/10.1007/978-3-642-00768-2_28

[42] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. 2011. Relational String Verification Using Multi-Track Automata. *Int. J. Found. Comput. Sci.* 22, 8 (2011), 1909–1924.

[43] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A Z3-based String Solver for Web Application Analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE).* 114–124.