Model Counting for Recursively-Defined Strings

Minh-Thai Trinh[1], Duc-Hiep Chu[2], and Joxan Jaffar[1]

[1] National University of Singapore, Singapore
[2] Institute of Science and Technology, Austria

Abstract. We present a new algorithm for model counting of a class of string constraints. In addition to the classic operation of concatenation, our class includes some *recursively defined* operations such as Kleene closure, and replacement of substrings. Additionally, our class also includes *length constraints* on the string expressions, which means, by requiring reasoning about numbers, that we face a *multi-sorted* logic. In the end, our string constraints are motivated by their use in programming for web applications.

Our algorithm comprises two novel features: the ability to use a technique of (1) *partial derivatives* for constraints that are already in a solved form, i.e. a form where its (string) satisfiability is clearly displayed, and (2) *non-progression*, where cyclic reasoning in the reduction process may be terminated (thus allowing for the algorithm to look elsewhere). Finally, we experimentally compare our model counter with two recent works on model counting of similar constraints, SMC [18] and ABC [5], to demonstrate its superior performance.

1 Introduction

In modern software, strings are not only ubiquitous, they also play a critical part: their improper use may cause serious security problems. For example, according to the Open Web Application Security Project [20], the most serious web application vulnerabilities include: (#1) Injection flaws (such as SQL injection) and (#3) Cross Site Scripting (XSS) flaws. Both vulnerabilities involve string-manipulating operations and occur due to inadequate sanitisation and inappropriate use of input *strings* provided by users.

The *model counting* problem, to count the number of *satisfiable assignments* for a constraint formula, continues to draw a lot of attention from security researchers. Specifically, model counters can be used directly by quantitative analyses of information flow (in order to determine how much secret information is leaked), combinatorial circuit designs, and probabilistic reasoning. For example, the constraints can be used to represent the relation between the inputs and outputs implied by the program in quantitative theories of information flow. This, in turn, has numerous applications such as quantitative information flow analysis [11, 24, 6, 21], differential privacy [3], secure information flow [22], anonymity protocols [10], and side-channel analysis [16]. Recently, model counting has also been used by probabilistic symbolic execution where the goal is to compute the probability of the success and failure program paths [13, 9].

Given the rise of web applications and their complicated manipulations of string inputs, model counting for string constraints naturally becomes a very important research problem [18, 5]. There have been works on model counting for different kinds of domains such as boolean [8], and integer domains [19]. But they are not directly applicable to string constraints. The main difficulties are: (1) string constraints need to be *multi-sorted* because we need to reason about string lengths; and (2) each string length is either *unbounded*, or bounded by a very large number. For example, we can represent a bounded string as a bit vector and then employ the existing model counting for bit vector constraints to calculate the number of solutions. However, as highlighted in [18], the bit-vector representation of the regular expression $\texttt{S}.\mathbf{match}(\text{“}\texttt{(a | b)}^{\star}\text{”})$ could grow exponentially w.r.t. the length of S, and the tools which employ this approach did not scale to strings of length beyond 20.

This work is inspired by two recent string model counters, SMC [18] and ABC [5][3], which have achieved very promising results. However, in contrast to these approaches, this paper *directly* addresses the two challenges of the string domain, (1) which is a multi-sorted theory, and (2) whose variables are generally unbounded. As a result, our model counter not only produces more precise counts, but also is generally more efficient.

We start by employing the infrastructure of the satisfiability solver S3P [26], which in turn builds on top of Z3 [12] to efficiently reason about multiple theories. S3P works by building its reduction tree, reducing the original formula into simpler formulas with the hope that it eventually encounter a *solved form* formula from which a satisfying assignment can be enumerated or proving that all the reduction paths lead to contradictions (i.e. the original formula is unsatisfiable). One key advancement of S3P is the ability to detect *non-progressive* scenarios with respect to a criterion of minimizing the “lexicographical length” of the returned solution, if a solution in fact exists. This helps avoiding infinite chains of reductions when dealing with unbounded strings. In other words, in the search process based on reduction rules, we can soundly prune a subproblem when the answer we seek can be found more efficiently elsewhere. If a subproblem is deemed non-progressive, it means that if the original input formula is satisfiable, then another satisfiable solution of shorter “length” will be found somewhere else. However, because a model counter needs to consider *all* solutions, what offered by S3P is not directly usable for model counting.

Our model counting algorithm proceeds by using the reduction rules of S3P, but to *exhaustively* build the reduction tree T. Each node will be associated with a “generating function” [18] representing its count. We compute the counts for all the leaf nodes, and propagate bottom-up to derive the count of the original input formula. There are four types of leaf nodes, i.e. a path is terminated when one of four scenarios is encountered:

(1) A *contradiction* is derived. The leaf node is assigned a precise count of 0. (This holds for any variable of interest with any length.)
(2) The leaf node is in *solved form*[4]. We delegate to a helper function to precisely count a formula in solved form.
(3) The leaf node is a *non-progressive* formula, detected by S3P’s rules. We can relate the count of that leaf to one of its ancestors via a recurrence relation.

[3] We will discuss them in more detail in the Related Work

[4] We will define “solved form” in Section 5.3.

(4) The path gets stuck or exceeds a predefined budget (often used to enforce termination), we resort to a baseline algorithm. In the implementation we choose SMC as the baseline algorithm.

Note however that counting the solutions of a formula in solved form, i.e., scenario (2), is not a trivial task. This is because the family of satisfiable strings might go beyond a regular language. Constraints on string lengths even further complicate the problem: a formula in solved form does not mean it is satisfiable. For this task, we adapt the notion of *partial derivative* function by Antimirov [4] to construct a tree, called an enumeration tree (for each leaf formula of T that is in solved form). The key distinction of an enumeration tree over the top-level reduction tree is that, because formulas are in solved form, we can perform specialized over/under-approximation techniques for the length constraints, in order to *direct* the enumeration process to repeated formulas, so that recurrence relations between the counts of them can be extracted. In the end, we use Mathematica to evaluate the count for the original formula, given a specific length to the string variable of interest.

Contributions: In summary, this paper proposes a new model counter, called S3#. We make the following theoretical contributions:

- We leverage the infrastructure of an existing string solver, namely S3P, to directly address the two main challenges of model counting for string constraints.
- We convert each non-progression scenario into a recurrence relation between the solution counts of formulas in our reduction tree.
- We propose a novel technique to precisely count the solutions of solved form formulas.

In our empirical evaluation of our implementation, we demonstrate the precision and efficiency of our model counting technique via real-world benchmarks against SMC and ABC, the two state-of-the-art model counting techniques for string constraints. Our first criterion is accuracy, and here we show clearly that our answers are more accurate in all cases. A second criterion is efficiency. We shall argue that we are in fact more efficient. However, there will be some counter-examples. But here we shall demonstrate that the counter-examples are themselves countered by a subsequent lack of accuracy. In the end, we demonstrate that S3# is for now better than the state-of-the-art.

2 Problem & Related Work

We will define the model counting problem for strings, discuss the implications in terms of soundness and precision. We also cover main related work in this Section.

2.1 Problem Definition

Suppose we have a formula F over free variables V. We shall defer defining the grammar for F for now. Let $\texttt{cvar} \in V$ be the string variable of interest and n denote the (symbolic) length of $\texttt{cvar}$. Let $S_{\texttt{cvar}}$ denote the set of solutions for $\texttt{cvar}$ that satisfies

F. We define the model counting problem as finding an estimate of $|S_{\texttt{cvar}}|$ as a function of n, denoted by the quantity $S_{\texttt{cvar}}(n)$. In this paper, we focus on finding a precise *upper bound* $u(n)$ to $S_{\texttt{cvar}}(n)$. For certain applications, a *lower bound* estimate $l(n)$ is of more interest, but it can be defined analogously.

Even though our technique can also produce a precise lower bound, restricting the problem to an upper bound estimate helps in two ways: (1) it is easier to make comparison with ABC [5], which returns only an upper bound estimate; (2) the notions of soundness and precision are more intuitive as follows.

We say that an upper bound $u(n)$ is:

- *sound* iff $\forall i \geq 0, S_{\texttt{cvar}}(i) \leq u(i)$.
- *κ-precise* wrt. some $i \geq 0$ iff κ is the relative distance between $u(i)$ and $S_{\texttt{cvar}}(i)$, i.e. $\kappa = \frac{u(i)-S_{\texttt{cvar}}(i)}{S_{\texttt{cvar}}(i)}$; where $0/0 = 0$ and a positive number divided by zero equals to infinity.

Given a concrete length i of interest for `cvar`, we say an upper bound is the *exact* estimate/count if it is *0-precise* w.r.t. to i. Our definition also implies that it is extremely imprecise to provide a positive count for an unsatisfiable formula (in software testing, this leads to false positives). Furthermore, in the counting process, it is unsound to miss a satisfiable assignment, whereas counting an unsatisfiable assignment or counting one satisfiable assignment for multiple times (also called duplicate counting) are the main reasons that lead to imprecise estimates.

2.2 Related Work

There has been significant progress in building string solvers to support the reasoning of web applications. Recent notable works include [15, 23, 30, 25, 1, 17, 2, 29, 26]. Some of these solvers bound the string length [15, 23], whereas our approach handles strings of arbitrary length (as does ABC). Our solver also supports complicated string operations such as **replace**, which is commonly used in real-world programs (both in JavaScript [23] and Java [14]).

However, to the best of our knowledge, there are only two solvers that support model counting for strings, namely SMC [18] and ABC [5]. ABC has been used to quantify side-channel leakage in a more recent work [7].

The pioneering work [18] proposes to use "generating function" in model counting. Their treatment of string constraints is, however, rather simple. Briefly, a formula is *structurally* broken down into sub-formulas, until each sub-formula is in primitive form so that a generating function can be assigned. The rest of the effort is to appropriately (but routinely) combine the derived generating functions. The rules to combine are slightly different between computing upper bound and computing lower bound estimates. Importantly, these rules are *fixed*. For example, given a formula $F = F_1 \vee F_2$, SMC will count the upper bound for the number of solutions of F_1 and F_2 and then sum them up without taking into account the overlapping solutions between F_1 and F_2. Similarly, the lower bound for $F_1 \wedge F_2$ is simply 0. As highlighted in [5], SMC cannot determine a precise count for a simple regular expression constraint such as $x \in (\text{“}a\text{”}|\text{“}b\text{”})^{\star}|\text{“}ab\text{”}$. It neither can *coordinate* the reasoning across logical connectives to infer precise counts for simple constraints such as $(x \in \text{“}a\text{”}|\text{“}b\text{”}) \vee (x \in$

$“a”|“b”|“c”|“d”)$ nor $(x \in “a”|“b”) \wedge (x \in “a”|“b”|“c”|“d”)$. In short, the sources of the imprecision of SMC may be ambiguous grammars, conjunctions, disjunctions, length constraints, high-level string operations, etc.

ABC [5] enhances the precision by a rigorous method: representing the set of solution strings as an extended form of a deterministic finite automaton (DFA) and then precisely enumerating the count when a bound on string length is given. However, there are two issues with this approach. First, it might suffer from an up-front exponential blow-up, in the DFA construction phase. For example, a DFA that represents the concatenation of two DFA could be exponential in size of the input DFAs [28]. (Note that ABC's premise that “the number of paths to accepting states corresponds to the solution count” only holds for a DFA.) Second, to reason about web applications, the constraint language is required to be expressive. This frequently leads to cases that the set of solutions cannot be captured precisely with a regular language, e.g. what is called “relational constraints” in [5]. In such cases, ABC suffers from serious imprecision.

3 Motivating Examples

As stated in Section 1, model counting techniques for bounded domains are not directly applicable to the string domain. We now present some motivating examples where state-of-the-art string model counters are not precise.

First, we discuss the limitation of SMC. As pointed out in [5], it has a severe issue of *duplicate* counting. SMC focuses on the syntax structure of the input formula to recursively break it down into sub-formulas until these are in a primitive form. Then a generating function can be assigned independently to each of them. In other words, SMC does not have a semantics-based analysis on the actual solution set. Below are two simple examples showing imprecise bounds produced by SMC:

$$\begin{array}{ll} X \in ((“a”|“b”)^\star|“ab”) & \Rightarrow 1 \leq S_X(2) \leq 5 \\ X \in (“a”|“b”) \vee X \in (“a”|“b”|“c”|“d”) & \Rightarrow 2 \leq S_X(1) \leq 6 \end{array}$$

The exact counts are 4 and 2 respectively. Both our tool S3# and ABC can produce these exact counts (as upper bounds). Next consider the following examples.

Example 1 (Regular language without length constraints). Count the number of solutions of X in:

$$X = Y \cdot Y \wedge Y \in (“a”)^\star$$

Though the set of solutions for X can be captured by a regular language, the word equation $X = Y \cdot Y$ involves a concatenation operation, making the example non-trivial for existing tools. While ABC crashes, SMC returns an *unsound* estimate $[0; 0]$ – indicating that both the lower bound and the upper bound are 0. (We actually observe this behaviour in our evaluation with small benchmarks in Section 6, Table 1).

Example 2 (Non-regular language with length constraints). Count the number of solutions of X in:

$$X = Y \cdot Z \wedge Y \in (“a”)^\star \wedge Z \in (“b”)^\star \wedge \mathbf{length}(Y) = \mathbf{length}(Z)$$

It can be seen that the set of solutions of X is beyond a regular language. In fact, it is a context-free language: $\{ a^m \cdot b^m \mid m \geq 0 \}$.

For this example, SMC is not applicable because it cannot handle the constraint $\mathbf{length}(Y) = \mathbf{length}(Z)$ — its parser simply fails. Counting the solutions of length 2 for X, ABC gives 3 as an upper bound, while the exact count is 1. For length 500, ABC's answer is 501 though the exact count is still 1. Our tool S3# can produce the exact counts for all these scenarios.

In general, ABC does not handle well the cases where the solution set is not a regular language. The reason is that ABC needs to approximate all the solutions as an automaton before counting the accepting paths up to a given length bound. This limitation is quite serious because in practice, e.g. in web application, length constraints are often used. Therefore, the solution set is usually beyond a regular language. (This is realized frequently in our evaluation with Kaluza benchmarks in Section 6, Table 2 and Table 3).

4 The Core Language

We present the core constraint language in Figure 1.

$$
\begin{array}{llll}
\mathit{Fml} & ::= & \mathit{Literal} \mid \neg\, \mathit{Literal} \mid \mathit{Fml} \wedge \mathit{Fml} & \\
\mathit{Literal} & ::= & A_s \mid A_l & \\
A_s & ::= & T_{str} = T_{str} & \\
A_l & ::= & T_{len} \leq m & (m \in C_{int}) \\
T_{str} & ::= & a & (a \in C_{str}) \\
 & \mid & X & (X \in V_{str}) \\
 & \mid & \mathbf{concat}(T_{str}, T_{str}) & \\
 & \mid & \mathbf{replace}(T_{str}, T_{regexpr}, T_{str}) & \\
 & \mid & \mathbf{star}(T_{regexpr}, M) & (M \in V_{int}, M \geq 0) \\
T_{regexpr} & ::= & a & (a \in C_{str}) \\
 & \mid & (T_{regexpr})^{\star} \mid T_{regexpr} \cdot T_{regexpr} & \\
 & \mid & T_{regexpr} + T_{regexpr} & \\
T_{len} & ::= & m & (m \in C_{int}) \\
 & \mid & M & (M \in V_{int}) \\
 & \mid & \mathbf{length}(T_{str}) \mid \Sigma_{i=1}^{n}(m_i * T_{len}) &
\end{array}
$$

Fig. 1: The Syntax of Our Core Constraint Language

Variables: We deal with two types of variables: V_{str} consists of string variables (X, Y, Z, T, and possibly with subscripts); and V_{int} consists of integer variables (M, N, P, and possibly with subscripts).

Constants: Correspondingly, we have two types of constants: string and integer constants. Let C_{str} be a subset of $\xi^{\star}$ for some finite alphabet ξ. To make it easier to compare with other model counters, we choose the same alphabet size, that is 256. Elements of C_{str} are referred to as string constants or constant strings. They are denoted by a, b, and possibly with subscripts. The empty string is denoted ϵ. Elements of C_{int} are integers and denoted by m, n, possibly with subscripts.

Terms: Terms may be string terms or length terms. A string T_{str} term (denoted D, E, and possibly with subscripts) is either an element of V_{str}, an element of C_{str}, or a function on terms. More specifically, we classify those functions into two groups: recursive and non-recursive functions. An example of recursive function is **replace** (which is used to replace *all* matches of a pattern in a string by a replacement), while an example of non-recursive function is **concat**. The concatenation of string terms is denoted by **concat** or interchangeably by $\cdot$ operator. For simplicity, we do not discuss string operations such as **match**, **split**, **exec** which return an array of strings. We note, however, these operations are fully supported in our implementation.

A length term (T_{len}) is an element of V_{int}, or an element of C_{int}, or a **length** function applied to a string term, or a constant integer multiple of a length term, or their sum. Furthermore, $T_{regexpr}$ represents regular expression terms. They are constructed from string constants by using operators such as concatenation ($\cdot$), union ($+$), and Kleene star ($\star$). Regular expression terms are only used as parameters of functions such as **replace** and **star**.

Following [25], we use the **star** function in order to reduce a membership predicate involving Kleene star to a word equation. The **star** function takes two input parameters. The first is a regular expression term, while the second is a non-negative integer variable. For example, $X \in (r)^\star$ is modeled as $X = \mathtt{star}(r, N)$, where N is a *fresh* variable denoting the number of times that r is repeated.

Literals: They are either string equations (A_s) or length constraints (A_l).

Formulas: Formulas (denoted F, G, H, K, I, and possibly with subscripts) are defined inductively over literals by using operators such as conjunction ($\wedge$), and negation ($\neg$). Note that, each theory solver of Z3 considers only a conjunction of literals at a time. The disjunction will be handled by the Z3 core. We use $\mathtt{Var}(F)$ to denote the set of all variables of F, including bound variables. Finally we can define the quantifier-free first-order two-sorted logic for our formulas as simply string equations involving some recursive and non-recursive functions, conjoined with some length constraints.

As shown in [25], to sufficiently reason about web applications, string solvers need to support formulas of quantifier-free first-order logic over string equations, membership predicates, string operations and length constraints. Given a formula of that logic, similarly to other approaches such as [25], our top level algorithm will reduce membership predicates into string equations where Kleene star operations are represented as recursive **star** functions. Other high level string operations can also be reduced to the above core constraint language. After such reductions, the new formula can be represented in our core constraint language in Figure 1. Note that, our input language subsumes those of other tools. For example, compared with ABC, our **replace** operation can take as input string variables instead of just string constants.

5 Algorithm

We first present the top-level algorithm, and then more details on the helper functions.

5.1 Top-level Algorithm

The top-level algorithm is the recursive function SOLVE presented in Algorithm 1. It takes two input arguments, a current formula F and γ, which is a list of pairs, each containing a formula and a sequence. γ is used to detect non-progressive formulas; we will discuss how γ is constructed and maintained in Section 5.2.

Given an input formula I and a variable of interest `cvar`, treated as global variables, an upper bound estimate $u(n)$ of the count is computed by invoking $\textsc{solve}(I, \emptyset)$. When given a specific length `len` for `cvar`, we can get an integer estimate by evaluating $u(\texttt{len})$ using Mathematica. We discuss how to compute lower bound in our technical report [27].

Our algorithm constructs a reduction tree similar to the satisfiability checking algorithm in [26]. Specifically, the construction of the tree is driven by a set of *rules*.

Definition 1 (Reduction Rule). *Each rule is of the general form*

$$\textit{(RULE-NAME)}\ \frac{F}{\bigvee_{i=1}^{m} G_i}$$

where F, G_i are conjunctions of literals[5], $F \equiv \bigvee_{i=1}^{m} G_i$, *and* $\texttt{Var}(F) \subseteq \texttt{Var}(G_i)$. □

An application of this rule transforms a formula at the top, F, into the formula at the bottom, which comprises a number (m) of *reducts* G_i.

```
function SOLVE($F$: Fml, $\gamma$: a list of pairs of a formula and a sequence)
    if ($F \equiv$ false ) return 0                      /*....................... Case 1 .......................*/
    $\texttt{cnstr}_{\texttt{cvar}} \leftarrow$ EXTRACT($F$, cvar)
    if (ISSOLVEDFORM($\texttt{cnstr}_{\texttt{cvar}}$))
        return COUNT($\texttt{cnstr}_{\texttt{cvar}}$)           /*....................... Case 2 .......................*/
    if ($F$ contains a recursive term or a non-grounded concatenation)
        if ($\exists \langle K, \sigma \rangle \in \gamma, \exists$ progressive substitution $\theta$ w.r.t. $\sigma$ s.t. $F\theta \Rightarrow K$)
            Mark $K$ as an ancestor of a non-progressive formula
            return RECURRENCE($K, F, \theta$)        /*....................... Case 3 .......................*/
    if ($depth = max\_depth$ OR there is no rule to apply)
        return BASESOLVER($F$, cvar)            /*....................... Case 4 .......................*/
    if ($F$ contains a recursive term or a non-grounded concatenation)
        Let $\sigma_F$ be a sequence on Var($F$) s.t. $\tau$ is a prefix of $\sigma_F$
        $\gamma \leftarrow \gamma \cup \langle F, \sigma_F \rangle$
    $\bigvee G_i \leftarrow$ APPLYRULE($F$)                  /*.......... Apply a reduction rule ..........*/
    $sum \leftarrow 0$
    foreach reduct $G_i$ do
        $sum \leftarrow sum$ + SOLVE($G_i, \gamma$)        /*................ Recursive Case ................*/
    return EVALUATE($sum$, $F$)
end function
```

Algorithm 1: Top-level Algorithm

Our algorithm has four base cases that are mutually exclusive as follows:

[5] As per Figure 1.

- The current formula is unsatisfiable (line 1). We return 0 as the exact count.
- The current formula is in solved form (lines 2-4). We first extract the constraints that are relevant to `cvar`. If the extracted constraints are in solved form (which is defined in Sec 5.3), then we use the helper function COUNT to precisely compute the count of $\texttt{cnstr}_{\texttt{cvar}}$.
- The current formula is non-progressive (line 5-8), or the condition in line 6 holds. Intuitively, it means that there is an ancestor formula K that "subsumes" the current formula F (modulo a renaming θ). We then call the helper function RECURRENCE to express the count of F in terms of the count of K.
- The path is terminated because the maximum depth has been reached or no rule is applicable (lines 9-10). We then simply resort to an existing solver such as SMC.

It is important to note that except for case-1, where a contradiction is detected, a count in some other base case will generally be a "generating function" (e.g., as used in [18]).

Finally, lines 15-17 handle the recursive case, where we first apply a reduction rule to the current formula F, obtaining the reducts G_i. The estimate count for F is the sum of the estimate counts for those G_i. In line 18, if F is not marked as an ancestor of a non-progressive formula, then EVALUATE simply returns the expression *sum*, which is the summation of a number of generating functions. Otherwise, there exists some descendant of F that is deemed non-progressive due to F. For such case, *sum* will be an expression that also involves the count of F, but with some smaller length. In other words, we have a recurrence equation to constrain the count of F. We rely on a function, EVALUATE, to add a recurrence equation into a global variable ϕ that tracks all collected recurrence equations, and prepare its base cases (see Section 5.2) so that concretization can be done when later we provide a concrete value of `len`.

5.2 Non-progressive Formulas

We now discuss the process of detecting non-progression. We first choose any sequence τ from all the variables of the input formula I. Then whenever we encounter a recursive term or a non-grounded concatenation, we add a pair, which consists of the current formula F and a sequence σ_F from all of F's variables, to γ (lines 11-13). The condition for choosing σ_F is that τ must be a prefix of σ_F. This is to help compare solution lengths "lexicographically" [26]. In line 6 of Algorithm 1, if we can find a pair $\langle K, \sigma \rangle \in \gamma$, and a progressive substitution θ w.r.t. σ (informally, θ will increase the solution length), such that $F\theta \Rightarrow K$ then we call F a non-progressive formula. We illustrate with the following example.

Example 3 (Non-progression). Count the number of solutions of X in:

$$\text{“}a\text{”} \cdot X = X \cdot \text{“}a\text{”}$$

See Fig. 2 where K is the formula of interest. By applying (SPLIT) rule to K, we obtain two reducts K_1 and K_2. In K_1, X is an empty string, whereas in K_2 we deduce that "a" must be a prefix of X. Next, by substituting X with "a"$\cdot X_1$ in K_2, we obtain F. If we keep on applying (SPLIT) and (SUB) rules, we will go into an infinite loop. As such, non-progression detection [26] is crucial to avoid non-termination. The technique will

$$\text{(SPLIT)}\ \dfrac{K \equiv \text{“}a\text{”} \cdot X = X \cdot \text{“}a\text{”}}{K_1 \equiv X = \epsilon \wedge \text{“}a\text{”} = \text{“}a\text{”} \qquad \text{(SUB)}\ \dfrac{K_2 \equiv X = \text{“}a\text{”}\cdot X_1 \wedge \text{“}a\text{”} \cdot X = X\cdot\text{“}a\text{”}}{F \equiv X = \text{“}a\text{”}\cdot X_1 \wedge \text{“}a\text{”} \cdot X_1 = X_1\cdot\text{“}a\text{”}}}$$

Fig. 2: Solving Steps for Example 3

find $\theta = [X_1/X]$ s.t. $F\theta \Rightarrow K$ and conclude that F is non-progressive. For satisfiability checking, it is sound to prune F and continue the search for a solution in K_1.

However, for model counting, we have to consider all solutions, including those contributed by F, if any. Thus we propose, instead of pruning F, we extract a relationship between the counts of F and of K, with RECURRENCE as a helper.

```
function RECURRENCE($K$: Fml, $F$: Fml, $\theta$)
    $d \leftarrow$ DIFF($K, F, \theta$, cvar)
    Let $f_K$ be a function over $l_K$, respresenting the estimate count of $K$
    return $f_K(l_K - d)$
end function
```

Algorithm 2: RECURRENCE Function

RECURRENCE is presented in Algorithm 2. It is important to note that based on θ, we can compute the length difference between `cvar` in K and the corresponding variable (for the substitution) in F. For the example above, it is the length difference between X and X_1, which is 1. We then can extract a relationship between the count of F and the count of K, thus further constraining the count of K with a recurrence equation.

Let f_K be the counting function for K; it takes as input the symbolic length l_K of `cvar` and returns the number of solutions of `cvar` for that length. In short, because $F\theta \implies K$, the count for F is (upper) bounded by $f_K(l_K - 1)$.

Now assume we compute the count of K (the variable of interest is still X) with `len` $= 3$. Following Algorithm 1, when we backtrack to node K, its *sum* is the expression $f_{K_1}(l_K) + f_K(l_K - 1)$; where $f_{K_1}(l_K)$ is a function that returns 1 when l_K is 0, and returns 0 otherwise. By calling EVALUATE($f_{K_1}(l_K) + f_K(l_K - 1)$, K) in line 18, we will add a recurrence equation $f_K(l_K) = f_{K_1}(l_K) + f_K(l_K - 1)$ into ϕ. We also compute its base case $f_K(0)$, which is $f_{K_1}(0) + f_K(-1) = 1 + 0 = 1$. (Based on the distance d, a number of base cases might be required.) Finally, since K is the input formula of interest, when given query length `len` = 3, we can compute the value of $f_K(3) = 1$.

5.3 Solved Form Formulas

We now discuss how to compute an estimate count for a formula in solved form, i.e., the COUNT function.

As presented in Figure 3, a formula is in solved form if it is a conjunction of atomic constraints and their negation. An atomic constraint is either an equality string constraint which is in solved form or a length constraint. To be in solved form, an equality string constraint can only be between a variable and a concatenation of *other* variables, between a variable and a constant, or between a variable and a **star** function. *Each variable can only appear once in the LHS of all equality constraints.*

$$
\begin{array}{lll}
\textit{SFml} & ::= \textit{Atom} \;\mid \neg\, \textit{Atom} \;\mid \textit{SFml} \wedge \textit{SFml} & \\
\textit{Atom} & ::= A_{eq} \;\mid A_{l} & \\
A_{eq} & ::= T_{var} = T_{concat} \;\mid T_{var} = T_{ground} & \\
T_{var} & ::= X & (X \in V_{str}) \\
T_{concat} & ::= X & (X \in V_{str}) \\
 & \mid\; \mathbf{concat}(T_{concat}, T_{concat}) & \\
T_{ground} & ::= a & (a \in C_{str}) \\
 & \mid\; \mathbf{star}(T_{regexpr}, M) & (M \in V_{int}, M{\geq}0)
\end{array}
$$

Fig. 3: Solved Form

In fact, one purpose of applying reduction rules is to obtain solved form formulas. For most cases, when no rule is applicable, the current formula is already in solved form. In this basic form, we can easily enumerate all the solutions for the string constraints. However, these solutions are also required to satisfy additional length constraints. As a result, a solved form formula system still might not have any solution.

Given a list of solved form formulas, we define its count as the count for the conjunction of all the formulas (note that the conjunction might not be in solved form). Now, given a solved form formula H, function COUNT will generate an enumeration tree rooted at $\{H\}$ (i.e. a singleton list with a formula H). Each node in the tree will be a list of solved form formulas, though as before, it is associated with a counting function, or *count* for short. Let β be a map between a formula list (i.e. a node in the tree) and its count. E.g., the count of $\{H\}$ is $f_H = \beta(\{H\})$. We then use function RECUR_EQ to collect a set of recurrence equations (added into ϕ) between the counts for different nodes in the tree. These equations are parameterized by an integer variable l_H. In the end, COUNT(H) will return the count for H, denoted by $f_H(l_H)$.

In RECUR_EQ function, given a list of formulas α, we compute the count $f_\alpha(l_\alpha)$. Lines 6-8 handle the case when there exists an unsatisfiable formula in the list α. Lines 9-11 handle the case when we can reuse the result of an ancestor node. Lines 12-18 are to derive the child nodes by applying partial derivative functions, which are defined below. The count for a parent node is the sum of those for child nodes, which do not have the same starting character c_i. Those which share the same starting character c_i are put into λ_i, which is a list of *SFml* list. For each λ_i, we use MOIVRE function to obtain the precise definition for the sum of the counts of all $\lambda_{ij}(1{\leq}j{\leq}n)$ (to avoid overlapping solutions). MOIVRE function will then call RECUR_EQ with the first parameter is a list of formulas, which is the flattened combination of elements from λ_i.

In Algorithm 3, the TAIL function (line 16) is implemented via the variants of the partial derivative function of regular expressions by Antimirov [4]. The Antimirov's function can be denoted as δ_c which compute the partial derivative of the input regular expression w.r.t. character c. Concretely, $\delta_c(r)$ is a regular expression whose language is the set of all words w (including the empty one) such that $c{\cdot}w \in L(r)$. We now extend it by defining the partial derivative function for negation-free *formulas* in solved form. (We explain the handling of negation in our technical report.)

Definition 2 (Partial Derivative). *Given a string variable X, and a character $c \in \xi$, a partial derivative function $\delta_{X,c}$ of a solved form formula is defined as follows:*

```
function COUNT($H$: SFml)
    Let $l_H$ be an integer variable and $\phi$ be a recurrence equation list
    Let $\beta$ be a mapping from SFml list to a counting function name
    RECUR_EQ($\{H\}$, $l_H$, $\phi$, $\beta$)
    return $\beta(\{H\})(l_H)$
end function

function RECUR_EQ($\alpha$: SFml list, $l_\alpha$: int, $\phi$: recurrence equation list, $\beta$: a mapping)
    Let $f_\alpha$ be a function from integer to integer
    if ($\exists H \in \alpha : H \equiv$ false )
        $\phi \leftarrow \phi \cup \{f_\alpha(l_\alpha) = 0\}$
        return
    if ($\exists$ variable renaming $\theta : \beta[\theta(\alpha)] = f_{parent}$)
        $\phi \leftarrow \phi \cup \{f_\alpha(l_\alpha) = f_{parent}(l_\alpha)\}$
        return
    $\beta[\alpha] \leftarrow f_\alpha$
    Let $\zeta$ be the set of all the possible starting characters of cvar
    $Sum \leftarrow 0$
    foreach character $c_i \in \zeta$ do
        $\lambda_i \leftarrow$ TAIL($\alpha$, cvar, $c_i$)                /* Each $\lambda_i$ is a SFml list list */
        $Sum \leftarrow Sum +$ MOIVRE$(\lambda_i, l_\alpha - 1, \phi, \beta)$
    $\phi \leftarrow \phi \cup \{f_\alpha(l_\alpha) = Sum\}$
end function

function MOIVRE($\lambda_i$: SFml list list, $N$: int, $\phi$: recurrence equation list, $\beta$: a mapping)
    Let $n$ be the size of $\lambda_i$
    for $k = 1$ to $n$ do
        Let $Comb$ be all the combination $\binom{n}{k}$ of $\lambda_i$ and $m$ be its size
        foreach combination $C \in Comb$ do
            $\alpha_i \leftarrow$ FLATTEN$(C)$
            RECUR_EQ($\alpha_i$, $N$, $\phi$, $\beta$)
        $a_k \leftarrow \Sigma_{i=1}^{m} f_{\alpha_i}$
    return $\Sigma_{k=1}^{n} (-1)^{k-1} * a_k$
end function
```

Algorithm 3: COUNT Function and Its Auxiliary Functions

$$\delta_{X,c}(Y = T_1) \overset{def}{=} \{Y = T_1\} \qquad \delta_{X,c}(A_l) \overset{def}{=} \{A_l\} \qquad \delta_{X,c}(X = \epsilon) \overset{def}{=} \{\texttt{false}\}$$

$$\delta_{X,c}(X = c{\cdot}s) \overset{def}{=} \{X = s\} \qquad \delta_{X,c}(X = d{\cdot}s) \overset{def}{=} \{\texttt{false}\} \;\; \textit{if} \;\; d \in \xi \textit{ and } d{\neq}c$$

$$\delta_{X,c}(X = \mathbf{star}(r, N)) \overset{def}{=} \{X = w \cdot \mathbf{star}(r, N{-}1)\}, \; \textit{where } w \in \delta_c(r)$$

$$\delta_{X,c}(X = Y \wedge H_2) \overset{def}{=} \{X = Y\} \overset{*}{\wedge} \delta_{Y,c}(H_2)$$

$$\delta_{X,c}(X = Y{\cdot}Z \wedge H_2) \overset{def}{=} \{X = Y \cdot Z\} \overset{*}{\wedge} \delta_{Y,c}(H_2) \;\; \textit{if} \;\; \neg e(Y)$$

$$\delta_{X,c}(X = Y{\cdot}Z \wedge H_2) \overset{def}{=} \{X = Y \cdot Z\} \overset{*}{\wedge} \delta_{Y,c}(H_2)$$
$$\cup \; \{X = Z \wedge Y = \epsilon\} \overset{*}{\wedge} \delta_{Z,c}(H_2) \;\; \textit{if} \;\; e(Y)$$

$$\delta_{X,c}(H_1 \wedge H_2) \overset{def}{=} \delta_{X,c}(H_1) \overset{*}{\wedge} \delta_{X,c}(H_2) \qquad \square$$

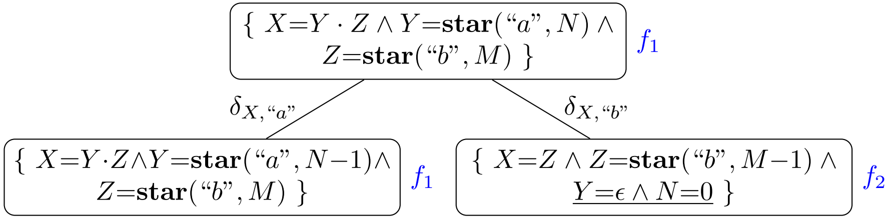
The function $e(Y)$ checks if a variable Y can be an empty string or not. For example, if we have $Y = \mathbf{star}(\text{“}a\text{”}, N) \wedge N{\geq}0$ then $e(Y) = \texttt{true}$, but if $Y = \text{“}a\text{”}$ then

$e(Y) = \texttt{false}$. Meanwhile the operator $\overset{*}{\wedge}$ for two sets is the Cartesian product version of $\wedge$. We now explain Definition 2 via a simple example.

Example 4 (String-only constraints). Count the number of solutions of X in:

$$X = Y \cdot Z \wedge Y = \mathbf{star}(\text{“}a\text{”}, N) \wedge Z = \mathbf{star}(\text{“}b\text{”}, M)$$

Below is the counting tree for the input solved form formula. Suppose the count for the root node is $f_1(l_1)$. By applying $\delta_{X,\text{“}a\text{”}}$ for the formula in the root node, we obtain the left node where $Y{=}\mathbf{star}(\text{“}a\text{”}, N{-}1)$. If we substitute $N{-}1$ with N, the formula in the left node becomes the formula in the root node. Therefore, the count for the left node is $f_1(l_1{-}1)$, since we have just removed a character “a” from X.



$\{\ X{=}Y \cdot Z \wedge Y{=}\mathbf{star}(\text{“}a\text{”}, N) \wedge$
$Z{=}\mathbf{star}(\text{“}b\text{”}, M)\ \}$
f_1
$\delta_{X,\text{“}a\text{”}}$
$\delta_{X,\text{“}b\text{”}}$
$\{\ X{=}Y{\cdot}Z{\wedge}Y{=}\mathbf{star}(\text{“}a\text{”}, N{-}1){\wedge}$
$Z{=}\mathbf{star}(\text{“}b\text{”}, M)\ \}$
f_1
$\{\ X{=}Z \wedge Z{=}\mathbf{star}(\text{“}b\text{”}, M{-}1) \wedge$
$\underline{Y{=}\epsilon \wedge N{=}0}\ \}$
f_2

In short, we have a set of recurrence equations as below:

$$f_1(l_1) = f_1(l_1{-}1) + f_2(l_1{-}1)$$

Note that, we will remove redundant constraints which do not affect the final count (e.g. $Y{=}\epsilon \wedge N{=}0$ in the right node). Similarly, we can have a counting tree for $X{=}Z \wedge Z{=}\mathbf{star}(\text{“}b\text{”}, M{-}1)$ and a recurrence equation for f_2. In addition, we also need to compute the base case for the definition of f_1, that is $f_1(0) = 1$.

The main technical issue that we have to overcome is non-termination of the counting tree construction (which leads to non-termination of REC_EQ function). Fortunately, because of the recursive structure of strings, in the case of string-only constraints, we can guarantee to terminate and to generate recurrence equations for every counting function (see Theorem 1). The difficulty here is of course when the constraints also include string lengths. To handle length constraints, we propose *over/under-approximation* techniques in order to give precise upper/lower bounds for counting functions. But first we need to propose another variant of the derivative function.

Definition 3 (Multi-Head Partial Derivative). *Let s = “?...?”$\cdot c$ be a concatenation between i copies of “?” and the character c. A multi-head partial derivative function $\Delta_{X,s}$ for the string variable X and the string s is defined as follows:*

$$\Delta_{X,c}(H) \overset{def}{=} \delta_{X,c}(H) \qquad \Delta_{X,s}(Y{=}T_1) \overset{def}{=} \{Y{=}T_1\} \qquad \Delta_{X,s}(A_l) \overset{def}{=} \{A_l\}$$

$$\Delta_{X,s}(X{=}Y \wedge H_2) \overset{def}{=} \{X{=}Y\} \overset{*}{\wedge} \Delta_{Y,s}(H_2)$$

$$\Delta_{X,s}(X{=}Y_0...Y_n \wedge H_2) \overset{def}{=} \{X{=}Y_0...Y_n\} \overset{*}{\wedge} \delta_{Y_i,c}(H_2) \quad \textit{if} \ \underset{0 \leq j \leq n}{\neg\mathrm{CONCAT}(Y_j)}$$

$$\Delta_{X,s}(H_1 \wedge H_2) \overset{def}{=} \Delta_{X,s}(H_1) \overset{*}{\wedge} \Delta_{X,s}(H_2)$$

□

The function CONCAT(Y) checks if a variable Y is bound with any concatenation. For example, if we have $Y = Z_1{\cdot}Z_2$ then CONCAT(Y)= $\texttt{true}$. Note that, given a negation-free formula in solved form, we can always transform it to the form $X{=}Y_0...Y_n \wedge Y_0 = T_0 \wedge ... \wedge Y_n = T_n \wedge A_l$, where $\neg$CONCAT(Y_j) $(0{\leq}j{\leq}n)$.

With the use of multi-head partial derivative function as the new implementation for the TAIL function (line 16), we now have to update Algorithm 3 correspondingly. Specifically, in line 13, instead of finding the starting characters c_i of `cvar`, we now need to construct the set of string s_i, which is composed by i copies of "?" and the character c_i. This construction is guided by the length constraints.

Suppose we have a set of constraints on string lengths. By using inference rules, we can always transform the above set into a disjunction of conjunctive formulas on the second parameters of **star** functions. For example,

$$X = Y{\cdot}Z \wedge Y{=}\mathbf{star}(\text{“}a\text{”}, N) \wedge Z{=}\mathbf{star}(\text{“}b\text{”}, M) \wedge 2{*}\mathbf{length}(Y){+}\mathbf{length}(Z){=}4{*}P$$

can be transformed into

$$X{=}Y{\cdot}Z \wedge Y{=}\mathbf{star}(\text{“}a\text{”}, N) \wedge Z = \mathbf{star}(\text{“}b\text{”}, M) \wedge 2N{+}M{=}4P.$$

Thus, w.l.o.g., let us assume that the length constraints exist in the form of a conjunctive formula on the second parameters of **star** functions. Suppose we have a formula H composed by a conjunction of equality constraints A_k (in which the variable of interest X is constructed by concatenating constant strings and $Y_i)$ and $Y_i{=}\mathbf{star}(s_i, N_i)$ $(0{\le}i{\le}p)$, along with linear arithmetic constraints on N_i $(0{\le}i{\le}n)$, where $N_0, ..., N_p$ are the second parameters of **star** functions, and $N_{p+1}, ...N_n$ are integer variables.

$$H \equiv \bigwedge A_k \wedge \bigwedge Y_i{=}\mathbf{star}(s_i, N_i) \wedge \bigwedge \Sigma_{i=0}^{i\le n} a_{ij}{*}N_i \le b_j \quad \text{where} \quad 0{\le}j{\le}m$$

Then we will try to solve the following set of constraints

$$\bigwedge \Sigma_{i=0}^{i\le n} a_{ij}{*}N_i \ge 0 \quad \text{where} \quad 0{\le}j{\le}m \qquad (1)$$

If (1) has a solution $(l_0, \ldots l_n)$, then we know that we have to go the node where we have the constraint $\bigwedge_{i=0}^{i\le p} Y_i{=}\mathbf{star}(s_i, N_i{-}l_i)$. Let G be the formula labelling that node. With the substitution $\theta = [N_0{-}l_0/N_0, ..., N_n{-}l_n/N_n]$, we will have $G\theta \Rightarrow H$. Therefore $f_G(l_G) = f_H(l_H{-}|s_0|{*}l_0{-}|s_1|{*}l_1 ...{-}|s_p|{*}l_p)$. This ensures the termination of the construction of the counting tree for H since other nodes are of less complexity than G.

Otherwise, we will try to remove as least as possible the integer constraints from (1) in order to make it become satisfiable. This is where the over-approximation applies. Suppose we have to remove the constraints where $j \in \mu$ to obtain a satisfiable formula

$$\bigwedge \Sigma_{i=0}^{i\le n} a_{ij}{*}N_i \ge 0 \quad \text{where} \quad 0{\le}j{\le}m \wedge j \notin \mu$$

then the upper bound for the number of solutions of H is the number of solutions of

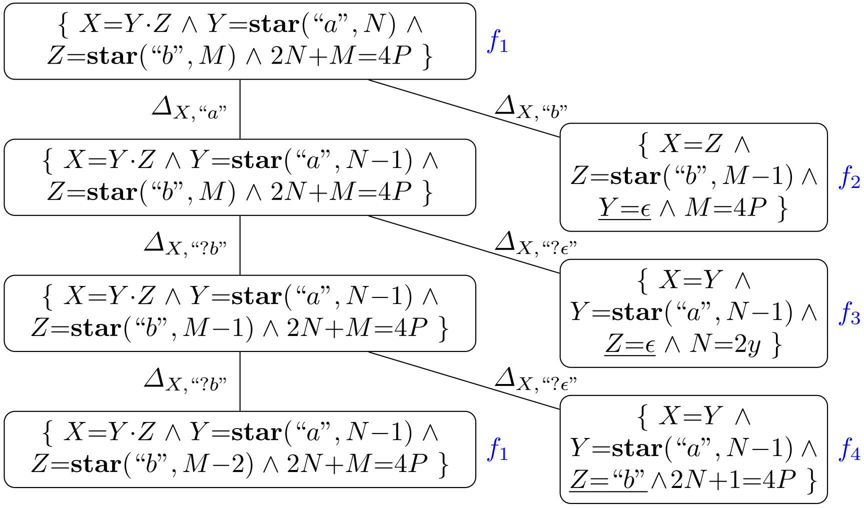
$$H' \equiv \bigwedge A_k \wedge \bigwedge Y_i{=}\mathbf{star}(s_i, N_i) \wedge \bigwedge \Sigma_{i=0}^{i\le n} a_{ij}{*}N_i \le b_j \quad \text{where} \quad 0{\le}j{\le}m \wedge j \notin \mu$$

It is obviously seen that the largest upper bound is the number of solutions of the string-only formula $H'' \equiv \bigwedge A_k \wedge \bigwedge Y_i{=}\mathbf{star}(s_i, N_i)$. (The lower bound for the number of solutions of H is the number of solutions of explored nodes in the counting tree for H. So the deeper we explore, the more precise lower bound we have. The smallest lower bound of course is 0.) To illustrate more, let us look at the following example.

Example 5 (String and length constraints). Count the number of solutions of X in:

$$X = Y \cdot Z \wedge Y = \textbf{star}(``a", N) \wedge Z = \textbf{star}(``b", M) \wedge 2N{+}M{=}4P$$

First, we need to solve the equation $2N{+}M{-}4P{=}0$ in order to find the solution $N{=}1$, $M{=}2$, $P{=}1$. Then we know that we need to drive the counting tree to the node that contains the constraint $Y = \textbf{star}(``a", N-1) \wedge Z = \textbf{star}(``b", M-2)$ as follows.


{ $X{=}Y{\cdot}Z \wedge Y{=}\textbf{star}(``a", N) \wedge$ $Z{=}\textbf{star}(``b", M) \wedge 2N{+}M{=}4P$ }
f_1
$\Delta_{X,``a"}$
$\Delta_{X,``b"}$
{ $X{=}Z \wedge$ $Z{=}\textbf{star}(``b", M{-}1) \wedge$ $\underline{Y{=}\epsilon} \wedge M{=}4P$ }
f_2
{ $X{=}Y{\cdot}Z \wedge Y{=}\textbf{star}(``a", N{-}1) \wedge$ $Z{=}\textbf{star}(``b", M) \wedge 2N{+}M{=}4P$ }
$\Delta_{X,``?b"}$
$\Delta_{X,``?\epsilon"}$
{ $X{=}Y \wedge$ $Y{=}\textbf{star}(``a", N{-}1) \wedge$ $\underline{Z{=}\epsilon} \wedge N{=}2y$ }
f_3
{ $X{=}Y{\cdot}Z \wedge Y{=}\textbf{star}(``a", N{-}1) \wedge$ $Z{=}\textbf{star}(``b", M{-}1) \wedge 2N{+}M{=}4P$ }
$\Delta_{X,``?b"}$
$\Delta_{X,``?\epsilon"}$
{ $X{=}Y \wedge$ $Y{=}\textbf{star}(``a", N{-}1) \wedge$ $\underline{Z{=}``b"} \wedge 2N{+}1{=}4P$ }
f_4
{ $X{=}Y{\cdot}Z \wedge Y{=}\textbf{star}(``a", N{-}1) \wedge$ $Z{=}\textbf{star}(``b", M{-}2) \wedge 2N{+}M{=}4P$ }
f_1

In short, we have a set of recurrence equations as below:

$$f_1(l_1) = f_1(l_1{-}3) + f_2(l_1{-}1) + f_3(l_1{-}1)$$
$$f_1(0) = 1; f_1(1) = 0; f_1(2) = 1; \forall n : f_4(n) = 0$$

Similarly, we can construct recurrence equations for f_2 and f_3.

Lastly, we make two formal statements about our algorithm. The proof sketch is in our technical report.

Theorem 1 (Soundness). *Given an input formula I, Algorithm 1 returns the sound upper bound (and lower bound) for the number of solutions of I.* □

Theorem 2 (Precision). *Given a solved form formula H which does not contain any constraints of type A_l (i.e. length constraints), Algorithm 3 returns the exact number of solutions of H.* □

6 Evaluation

We test our model counter S3# with two set of benchmarks, which have also been used for evaluating other string model counters. All experiments are run on a 3.2GHz machine with 8GB memory.

In the first case study, we use a small but popular set of benchmarks that are involved in different security contexts. For example, the experiments with 2 string manipulation utilities (wc and grep) from the BUSYBOX v.1.21.1 package, and one utility (csplit)

Table 1. Experiments with Small Benchmarks. The last column is to notify the bound is measured with a scale. The scale for marked rows are 10^{1465}, 10^{1465}, 10^{1129}, 10^{1289}, 10^{23}, 10^{14}, resp.

Program	Len	SMC		ABC		S3#		
		Lower/Upper Bound	Time	Upper-Bound	Time	Exact Count	Time	
ghttpd	620	[10626.2;1031904473.2]	45.1	_	TO	1031904472.8	0.4	*
	11	[256;767]	28.3	**256**	0.3	767	0.3	
ghttpd_ wo_len	620	[10626.2;1031904473.2]	45.1	**1023846357.2**	0.4	1031904472.8	0.4	*
	11	[256;767]	28.3	**765**	0.3	767	0.3	
nullhttpd	500	[**2.9**;1369.8]	24.8	0	0.5	0	0.3	*
csplit	629	[**5.9*10^{1460}**;3.1*10^{1481}]	160.5	Crash	_	0	0.4	
grep	629	[**0.7*10^{1408}**;0.1*10^{1435}]	255.0	Crash	_	0	0.4	
wc	629	[0.97;8.0]	245.9	Crash	_	0.97	5.6	*
obscure1	10	[11.2;11.6]	1.3	11.2	0.1	11.2	0.2	*
obscure2	6	[2.8;2.8]	8.9	2.8	0.5	2.8	0.3	*
strstr1	5	[196608;**196608**]	0.3	1099511431168	0.1	1099511431168	0.3	
strstr2	5	[16776960;16776960]	0.3	Crash	_	16776960	0.3	
regex	4	[0;**0**]	2.4	16	0.1	16	0.2	
contains	5	[67108096;67108096]	0.3	67108096	0.1	67108096	0.3	

from the COREUTILS v.8.21 package, demonstrate the quantification of how much information would be leaked if these utilities operate on homomorphically encrypted inputs as in AutoCrypt [18].

Table 1 summarizes the results of running S3# against SMC and ABC[6]. The first and second columns contain the input programs and the query lengths for the query variables. Given those inputs, we then report the bounds produced by each model counter along with its running time. Note that SMC and S3# can give both lower and upper bounds while ABC can only give upper bounds.

For each small benchmark, S3# can give the *exact* count (i.e. lower and upper bounds are equal). All input formulas here can in fact be transformed into solved form. This ultimately demonstrates the precision of our counting technique for solved form formulas. In Table 1, we highlight unsound bounds, generated by SMC and ABC, in bold with grey background.

In addition, the running time of S3# is small. It is much faster than SMC, and comparable to ABC. Among the three model counters, when ABC can produce an answer, it is often the fastest. In such cases, it is because an automaton can be quickly constructed to represent the solution set. However, ABC also crashes a few times with the "BDD is too large" error. For the `ghttpd` and length 620, ABC times out after 20 minutes. In these instances, the solution sets are beyond regular; ABC cannot effectively represent/over-approximate them using an automaton. In contrast, if we remove the length constraints from the `ghttpd` benchmark to obtain `ghttp_wo_len`, ABC can finish it within 0.4 seconds. This indicates that when the solution set is beyond regular, ABC not only loses it precision, but also loses its *robustness*.

[6] We used the latest versions from their websites, as of 20 Dec 2016.

We next consider Kaluza benchmarks, that was also used by SMC and ABC for their evaluations. These benchmarks were generated by Kudzu [23], when testing 18 web applications that include popular AJAX applications. The generated constraints are of boolean, integer and string types. Integer constraints also involve lengths of string variables, while string constraints include string equations, membership predicates.

Importantly, SMC cannot handle many constraints from the original benchmarks; instead SMC used an over-simplified version of Kaluza benchmarks where many important constraints are removed. (ABC [5] had also reported about the discrepancy when comparing with SMC.) As a result, we only compare S3# with ABC in this second case study, using the SMT-format version of Kaluza benchmarks as provided in [17].

Table 2. Kaluza UNSAT benchmarks

# Programs	ABC		S3#	
	Upper Bound	Time	Count	Time
2700	0	1477	0	1130
9314	Crash			

Table 3. Kaluza SAT benchmarks

# Programs	ABC		S3#	
	Upper Bound	Time	L&U	Time
24825	>0	6984	>0	46575
10445	Crash			

Table 2 and 3 summarize the results of running S3# and ABC with two sets of Kaluza benchmarks: satisfiable and unsatisfiable ones. Note that ABC crashes often, nearly half the time[7]. Importantly, for the unsatisfiable benchmark examples, S3# produces the exact count 0. ABC, as in [5], managed to run more benchmarks, but failed to produce the upper bound 0 for $2,459$ benchmark examples; thus they classified them as satisfiable. For the satisfiable examples, S3# is also more informative, always determining that the lower bound is positive.

7 Concluding Remarks and Future Work

We have presented a new algorithm for model counting of a class of string constraints, which are motivated by their use in programming for web applications. Our algorithm comprises two novel features: the ability to use a technique of (1) *partial derivatives* for constraints that are already in a solved form, i.e. a form where its (string) satisfiability is clearly displayed, and (2) *non-progression*, where cyclic reasoning in the reduction process may be terminated (thus allowing for the algorithm to look elsewhere). We have demonstrated the superior performance of our model counter in comparison with two recent works on model counting of similar constraints, SMC and ABC.

Though the algorithm is for model counting of string constraints, we believe it is applicable to other unbounded data structures such as lists, sequences. This is because both the solving and counting methods deal with recursive structures in a somewhat general manner. Specifically, the methods are applied to a general logic fragment of equality and recursive functions.

Acknowledgement. This research was supported by the Singapore MOE under Tier-2 grant R-252-000-591-112. It was also supported in part by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE/SHiNE) and Z211-N23 (Wittgenstein Award).

[7] This differs from the report in [5]. Understandably, ABC has been under active development and there is significant difference in the version of ABC we used and the version had been evaluated in [5].

References

1. P. Abdulla, M. Atig, Y.-F. Chen, L. Holk, A. Rezine, P. Rmmer, and J. Stenman. String constraints for verification. In *CAV*, pages 150–166. Springer, 2014.
2. P. Abdulla, M. Atig, Y.-F. Chen, L. Holk, A. Rezine, P. Rmmer, and J. Stenman. Norn: An smt solver for string constraints. In *CAV 2015*, pages 462–469. Springer, 2015.
3. Mario S. Alvim, Miguel E. Andres, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative information flow and applications to differential privacy. In Alessandro Aldini and Roberto Gorrieri, editors, *Foundations of Security Analysis and Design VI: FOSAD Tutorial Lectures*, pages 211–230, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
4. Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.
5. A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *CAV 2015*, pages 255–272, 2015.
6. M. Backes, B. Kpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, pages 141–153, May 2009.
7. Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *FSE*, pages 193–204, 2016.
8. Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. Quail: A quantitative security analyzer for imperative code. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 702–707, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
9. Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. Compositional solution space quantification for probabilistic software analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 123–132, New York, NY, USA, 2014. ACM.
10. Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Inf. Comput.*, 206(2-4):378–401, February 2008.
11. David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *J. Comput. Secur.*, 15(3):321–371, August 2007.
12. L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, 2008.
13. Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.
14. Scoot Kausler and Elena Sherman. Evaluation of string constraint solvers in the context of symbolic execution. In *ASE*, pages 259–270, 2014.
15. A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116. ACM, 2009.
16. Boris Köpf and David Basin. An information-theoretic model for adaptive side-channel attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 286–296, New York, NY, USA, 2007. ACM.
17. T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A dpll(t) theory solver for a theory of strings and regular expressions. In *CAV*, pages 646–662. Springer, 2014.
18. Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 565–576, New York, NY, USA, 2014. ACM.

19. Antonio Morgado, Paulo Matos, Vasco Manquinho, and Joao Marques-Silva. Counting models in integer domains. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings*, pages 410–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
20. OWASP. Top ten project, May 2013. `http://www.owasp.org/`.
21. Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. Symbolic quantitative information flow. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, November 2012.
22. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.
23. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *SP*, pages 513–528, 2010.
24. Geoffrey Smith. On the foundations of quantitative information flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 288–302, Berlin, Heidelberg, 2009. Springer-Verlag.
25. M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *ACM-CCS*, pages 1232–1243. ACM, 2014.
26. Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Progressive reasoning over recursively-defined strings. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 218–240, Cham, 2016. Springer International Publishing.
27. Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Technical report, 2017 (see the web-site). `http://www.comp.nus.edu.sg/~trinhmt/`.
28. Sheng Yu, Qingyu Zhuang, and Kai Salomaa. The State Complexities of Some Basic Operations on Regular Languages. *Theor. Comput. Sci.*, pages 315–328, 1994.
29. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV*. Springer, 2015.
30. Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *ESEC/FSE*, pages 114–124, 2013.