A Model Counter For Constraints Over Unbounded Strings

Loi Luu, Shweta Shinde, Prateek Saxena
School of Computing, National University of Singapore
{loiluu, shweta24, prateeks}@comp.nus.edu.sg

Brian Demsky
University of California, Irvine
bdemsky@uci.edu

Abstract

Model counting is the problem of determining the number of solutions that satisfy a given set of constraints. Model counting has numerous applications in the quantitative analyses of program execution time, information flow, combinatorial circuit designs as well as probabilistic reasoning. We present a new approach to model counting for structured data types, specifically strings in this work. The key ingredient is a new technique that leverages generating functions as a basic primitive for combinatorial counting. Our tool SMC which embodies this approach can model count for constraints specified in an expressive string language efficiently and precisely, thereby outperforming previous finite-size analysis tools. SMC is expressive enough to model constraints arising in real-world JavaScript applications and UNIX C utilities. We demonstrate the practical feasibility of performing quantitative analyses arising in security applications, such as determining the comparative strengths of password strength meters and determining the information leakage via side channels.

1. Introduction

Model counting is the classical problem of computing the number of solutions which satisfy a set of constraints. This problem arises in many fields of computer science including artificial intelligence, program optimizations and information flow analysis [30, 39]. For example, probabilistic inference problems in Bayesian networks can be solved by first representing the network as a set of propositional clauses, and then model counting the clause set to compute all the marginal probabilities [14, 17, 39]. Similarly, model counting has applications to various program transformation and optimization problems such as memory size minimization [47], worst case execution time estimation [33], increasing parallelism [47], and improving cache effectiveness [19]. More recently, model counting is used in a variety of security applications. For example, quantitative information flow (or QIF) is the problem of determining "how much" information flows from the inputs to the observable outputs of a program [44]. QIF analyses can be cast as model counting queries, where the constraints represent the relation between the inputs and outputs implied by the program. Efficient model counting, therefore, directly benefits QIF analysis.

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594331

To understand how model counting is useful in quantitative analyses and probabilistic reasoning, consider the following contrived example of a password checker program.

```
if (password == guess) accept=1; else accept=0;
```

The variable `password` is a secret input while users can control the value of input variable `guess`. By observing the output value (`accept`), an attacker can learn some information about the secret input. Let's say that the analyst is interested in computing the probability of the attacker's success at guessing the secret password in a single try, assuming that the secret password is randomly chosen. It is easy to see that the probability of success depends on the size of the password, and in fact, drops as the length increases. For example, suppose both `password` and `guess` are two booleans, the value of `password` is completely disclosed based on the observed value of `accept`. When both `password` and `guess` are 2-bit in size, the attacker's guess has probability $1/4$ of succeeding. To compute this probability, the analysis can solve a model counting query satisfying the relation (or constraints) imposed by the program between the inputs and outputs. Specifically, the attacker succeeds if and only if the constraint `password == guess` holds true — this is a simple constraint which is satisfied by only one out of the 4 possible values for the attacker-controlled input `guess`. In analysis of real programs, these constraints could be much more complex. For instance, the targeted `password` could have additional strength requirements imposed [24] or the `guess` could be computed using automated rules by a tool [49]. Encoding such constraints may yield a complex set of constraints in the analysis. In this work, we initiate the study of the problem of model counting on structured data types, specifically strings, using a new approach.

Model Counting for Strings. Previous work shows that model-counting is feasible for constraints represented over integers and booleans [8, 16]. But these techniques cannot be directly applied to complex data representations such as strings. Reasoning about string-manipulating programs and code is of growing importance for various analyses, as evidenced by the large body of recent work on solvers for satisfiability-checking of string constraints [28, 29, 40]. However, model counting for strings, which is a step beyond satisfiability checking, has not been addressed so far.

Consider the constraint `strstr(s1, "xxxxy")` $= 1000$. This constraint is satisfied when the first occurrence of the pattern `"xxxxy"` in the string `s1` begins at offset 1000. A naïve solution is to directly enumerate all possible strings and check their validity against the constraints. However, this approach is intractable for large solution sets. Alternatively, techniques that attempt to trace the program execution and exhaustively explore all paths through the `strstr` function at the instruction level will result in path exploration space that is exponential in the size of `s1`. Another approach is to apply the existing model counting techniques directly to strings. We can represent a string as a bitvector and the string constraints as operations over bitvectors. Then we employ the current model counting for bitvectors constraints to calculate

the number of solutions. However, this approach may not scale to complex string constraints. For example, the constraints representing the regular expression `S.match("(a|b)*")` as bitvectors can grow exponentially in the size of input [29, 40]. In fact, we evaluate the inefficacy of these approaches (Section 4) for simple string programs, and none of our evaluated tools scale for strings of size beyond 20 bytes.

Our solution. To address the above challenges, this paper initiates the study of the string model-counting problem and presents a new tool called SMC. Given a set of string constraints, SMC can compute the bounds on the cardinality of the valid string set with high precision and efficiency. Our observation is that *generating functions (GFs)*, an important mathematical tool for reasoning about infinite series, provide a mechanism for reasoning about the cardinality of string sets [41]. The basic idea behind GFs is to encode the number of strings of length k as the *k-th* coefficient of a polynomial. These polynomials can be represented as finite expressions. These finite expressions (GFs) are powerful because they represent potentially infinite sets of strings. As we show in this work, operations or constraints over string variables directly translate into operations over the corresponding GFs of the operands. Therefore, model counting for string constraints that appear in the real-world code can be achieved via operations over GFs.

We present an implementation of our solution in SMC tool. Our SMC infrastructure is open source and is available online [13]. We evaluate SMC on 18,901 benchmarks generated from real-world JavaScript web applications [40]. We find that SMC can compute the total number of solutions for 18,901 benchmarks in 3 hours 7 minutes with an average of 0.595 seconds per benchmark. We demonstrate the use of SMC to quantify the sensitive information leaked in several UNIX utilities when they operate on encrypted data as proposed in a recent work [46]. As a final case study, we use SMC to quantitatively compare the strength of three password meters in real-world websites — Ebay, Drupal and Microsoft, and measure their efficacy in preventing passwords that are known dictionary words [24].

Contributions. This paper makes the following contributions:

- **String Analysis Approach and the SMC Tool:** We present a new approach for model counting problem on string data type by using *generating functions*. Our string model counting tool (SMC) can analyze string operators in real world C and JavaScript programs for unbounded strings.
- **Evaluation:** SMC outperforms publicly available string model counting tools in precision and efficiency. Further, our evaluation illustrates the applications of SMC to QIF in general and specifically to compare password strength meters.

2. Problem & Approach

Model counting provides a tool that is useful in a variety of analyses. In this work, we focus on the problem of model counting specifically for strings and present a solution that is scalable for practical applications.

2.1 Motivating Example

Consider the following problem which is useful in security analysis of an attacker's effort in a password guessing attack. Suppose a remote attacker learns the UNIX login password for a system administrator, but the administrator learns of the breach and proactively changes the password before the attacker can use it. On UNIX systems, typically users use the `passwd` utility to change their password which performs "strength checks" on the new password before updating it. Specifically, Figure 1 shows the code for the C utility `obscure` from BUSYBOX [1] which is used by the `passwd`

```
1 static int string_checker_helper
2 (const char *p1, const char *p2) {
3    /* as sub-string */
4    if strcasestr(p2, p1) != NULL
5    /* invert */
6    || strcasestr(p1, p2) != NULL)
7       return 1;
8    return 0;
9 }
10
11 static int string_checker
12 (const char *p1, const char *p2) { ...
13    int ret = string_checker_helper(p1, p2); ...
14    char *p = reverse_of(p1); ...
15    ret |= string_checker_helper(p, p2);   ...
16    return ret;
17 }
18
19 static const char *obscure_msg(const char *old_p,
20    const char *new_p, const struct passwd *pw) {
21    ...
22    if (old_p && old_p[0] != '\0') {
23       /* check vs. old password */
24       if (string_checker(new_p, old_p)) {
25          return "similar to old password";
26       }
27    }   ...
28    return NULL;
29 }
```

Figure 1: Code snippet from a simplified version of `obscure.c` in BUSYBOX to reject weak passwords. The function *obscure_msg* returns a message to indicate why the new password *new_p* is bad.

utility to check the password strength. The analysis question is — how many possible new password values are there for the attacker to try? Note that the attacker knows the old password and the constraints imposed on `new_p` by `obscure` function.

Model counting techniques can be used to answer this question. Concretely, let's say the administrator's old password (`old_p`) is `ab@123`, and the attacker is trying to estimate the size of set of possible new passwords (`new_p`). The `obscure` function updates the new password only if the old and new password are not too *similar*, that is, one does not contain the other or its reverse case-insensitively. Therefore, the relation between the old and new password can be expressed as a set of constraints, and the attacker can use it to compute the size of solutions for `new_p`. For example, consider the code snippet in Figure 1. Under the input `old_p` = `ab@123` and a new valid password, Lines 22, 24, 13, 4, 6, 8, 14, 15, 4, 6, 8, and 28 are executed. This path imposes the following constraints on `new_p`:

```
strcasestr(new_p, "ab@123") == NULL ∧
strcasestr("ab@123", new_p) == NULL ∧
strcasestr(new_p, "321@ba") == NULL ∧
strcasestr("321@ba", new_p) == NULL
```

Extracting such constraints is an independent challenge and can be done by a variety of techniques. For example, symbolic analysis tools [5, 22, 26, 27, 40] could compute these constraints by analyzing the body of the `obscure` function, given the `old_p` value as `ab@123`. In this paper, we assume that these pre-computed constraints are given as input. Once these constraints are extracted, our focus is on solving the model counting query efficiently.

2.2 Problem Definition

In the string model counting problem, we are given a set of constraints C over the set of free variables V. Let S_q be the set of feasible solutions for a query of an n-bit variable $q \in V$ that satisfies C. The model counting problem is then to estimate $|S_q|$ as a function of n, denoted by the quantity $S_q(n)$. Our solution is represented as upper and lower bounds $(u(n), l(n)$ respectively) on the cardinality of the set of solutions to a set of constraints.

There are two important properties of model counters, their *precision* and *soundness*. We say that the model counter is:

- *Sound* iff it produces bounds $[l(n), u(n)]$, such that $\forall i \in [0, n]$, $l(i) \leq |S_q(i)| \leq u(i)$.
- *ϵ-precise* iff ϵ is the distance of $l(n)$ and $u(n)$ in *log*-scale, specifically $\epsilon = \frac{log_2(u(n)+1) - log_2(l(n)+1)}{log_2(2^n+1)}$. Note that $0 \leq \epsilon \leq 1$ and the smaller the ϵ, the better the bounds. If $\epsilon = 0$, the model counter computes the precise number of solutions (i.e. $l(n) = u(n)$). On the other hand, if $\epsilon = 1$, it returns the most imprecise (naïve) bounds as [0, 2^n].

We seek to determine the set cardinality as a function of the bit-width of the query variable q. Most prior systems only reason about counting queries for user-specified, finite bit-width inputs and are limited to unstructured data types (such as integers or bit-vectors) [5, 34]. In contrast, we develop a model counter for the theory of strings which is expressive enough to model string constraints from real-world applications, such as JavaScript programs [40]. Our tool SMC takes as input a set of string constraints C, and reports the size of the feasible set $S_q(n)$ satisfying C, for a query variable q of length n. SMC computes sound upper and lower bound estimates of $S_q(n)$.

2.3 Challenges & Approach Overview

To see why model counting string constraints is challenging, let's consider the limitations of applying the existing approaches to string model counting in our running experiment (Figure 1). First, note that the size of the solutions is exponential [1] in the size of `new_p`. Therefore, solutions that enumerate to count the elements in solution space will take exponential time to produce the set size.

Another approach is to use dynamic symbolic execution techniques that trace the execution of the `strcasestr` implementation. Given a symbolic n-byte `new_p`, we can trace the execution of the `strcasestr` implementation to extract the branch conditions involving checks on each byte in the symbolic input. Each executed path yields a set of path constraints over bit-vectors (or arrays of bytes). The number of solutions to these constraints can be efficiently calculated using off the shelf tools [5]. Then, we can sum up the set of solutions over all paths executed in the `strcasestr` function. This idea works but its running time is proportional to the number of executed paths. Again, readers can verify that the number of paths explored in a naïve implementation of `strcasestr` can be exponential in the input size [2]. These techniques have been used in previous counting tools, however, they do not scale up for constraints over unbounded strings.

In our approach, we avoid counting by enumerating or summing over paths in the constraints altogether. Our high-level insight is that programs use specific, well-defined operations to construct and compute on structured data types like strings. If we utilize this structure in our counting mechanism, we can compute the solution set cardinalities precisely and more efficiently. More precisely, the key idea that enables reasoning about unbounded, structured data types like strings is the novel use of *generating functions* — a succinct closed form expression that represents the set size. This expression captures the size of the solution set not only for a single input size, but also for the set cardinality of all (possibly infinite) solutions. Each given constraint structurally restricts or expands the solution space based on the size of the solution sets of its input operands. Therefore, constraints can be modeled as operations over the corresponding closed form expressions of their input operands, yielding the resulting set cardinality also as a closed form expression. Furthermore, this approach enables us to reuse existing symbolic manipulation tools such as Mathematica for computing on GFs [50].

[1] In total there are 256^n possible strings of length n, and the constraints reduce this set by a small amount as only some strings have the expected patterns in them.

[2] Consider a naïve implementation in which there is nested loop. The outer loop picks a position i in the input at which the match might occur. The inner loop performs a character-by-character comparison against the searched pattern. If the inner loop fails, the outer loop increments the position and retries. Here there are exponentially many paths possible. For $i = 0$, there could be 6 places where the inner loop fails to complete the match with `ab@123`. For $i = 1$, there are again 6 places where the inner loop could fail to match. Each failed position leads to a new path. For a 1,000 byte input, there could be roughly 6^{1000} paths when `ab@123` is not a substring.

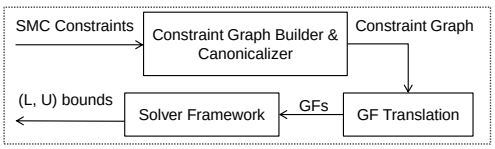

Figure 2: Architecture of SMC Constraint Solver

SMC first translates the given constraint into its constraint language, and then applies inference rules to produce a generating function which represents the cardinality of the set of `old_p` values satisfying these constraints (See Figure 2). For example, the constraint is first translated into a disjunction of simpler SMC clauses that capture the semantics of the `strcasestr` operator, as shown in Table 1. Subsequently, it computes bounds for each constraint independently and combines the closed form expressions for the lower and upper bounds using the composition rules defined in Section 3.2. The closed form expressions capture the cardinality of the solution set for *all* possible lengths, as we explain later, and can be evaluated at specific lengths. For `obscure` example, SMC reports that there are 8 unsatisfiable values of `new_p` of length 6 which do not satisfy the constraints within 1 second. Thus, the attacker cannot reduce his guessing space much although he knows `old_p` and `obscure`'s source code. For `new_p` of length 7, SMC reports 512 to 4,084 unsatisfiable values. In fact, the exact number of unsatisfiable `new_p` of length 7 is 4,084, hence our computed bounds are not only sound but precise ($\epsilon = 10^{-15}$). Subsequent queries for a different input length with SMC return immediately, as the closed form expression is already computed. In contrast, the previous QIF tool we had access to, reasons about these constraints at the level of bits, and it could not produce an answer in 2 hours [5] (See Section 4.3).

3. SMC Design

The key idea we utilize in this work is an encoding of the solution set using generating functions. We provide a quick introduction to generating functions in Section 3.1. The SMC language and design is presented in Section 3.2 and Section 3.3. Figure 2 shows the high level architecture of our SMC tool.

3.1 String Sets Cardinality and Generating Functions

Generating functions (GFs) have been utilized in analysis of combinatoric structures and algorithmic complexity of programs extensively [31, 41]. However, GFs have sparingly been used in automated program analyses of practical data structures; here we provide a quick background on generating functions for encoding cardinality of string sets.

We define the set Ω^P to be the set of strings that satisfy the set of properties P. Next, we define the set of strings S_i^P as the set of strings of length i that satisfy the set of properties P. It then directly follows that we can represent Ω^P as a union of all sets S_i^P:

$$\Omega^P = \bigcup_{i \geq 0} S_i^P$$

Because the collection of sets $\{S_i^P \mid i \geq 0\}$ is pairwise disjoint, i.e., one string cannot have two different length i and j that $j \neq i$,

Constraint	SMC's constraint	GF for lower bound	GF for upper bound
`strcasestr(new_p, "ab@123") == NULL`	C_1 = ¬ `new_p.contains("ab@123")` ∧ ¬ `new_p.contains("aB@123")` ∧ ¬ `new_p.contains("Ab@123")` ∧ ¬ `new_p.contains("AB@123")`	$\frac{1}{4X^6+(1-MX)}$	$\frac{1}{X^6+(1-MX)}$
`strcasestr("ab@123", new_p) == NULL`	C_2 = ¬ `new_p = "ab@123"` ∧ ¬ `new_p = "aB@123"` ∧ ... ∧ ¬ `new_p = "2"` ∧ ¬ `new_p = "1"`	$\frac{1}{1-MX} - (8X + 9X^2 + 8X^3 + 7X^4 + 6X^5 + 4X^6)$	$\frac{1}{1-MX} - (8X + 9X^2 + 8X^3 + 7X^4 + 6X^5 + 4X^6)$
`strcasestr(new_p, "321@ba") == NULL`	C_3 = ¬ `new_p.contains("321@ba")` ∧ ¬ `new_p.contains("321@bA")` ∧ ¬ `new_p.contains("321@Ba")` ∧ ¬ `new_p.contains("321@BA")`	$\frac{1}{4X^6+(1-MX)}$	$\frac{1}{X^6+(1-MX)}$
`strcasestr("321@ba", new_p) == NULL`	C_4 = ¬`new_p = "321@ba"` ∧ ¬`new_p = "321@Ba"` ∧ ... ∧ ¬`new_p = "a"` ∧ ¬`new_p = "A"`	$\frac{1}{1-MX}$ - $(8X + 9X^2 + 8X^3 + 7X^4 + 6X^5 + 4X^6)$	$\frac{1}{1-MX} - (8X + 9X^2 + 8X^3 + 7X^4 + 6X^5 + 4X^6)$
Full Path Constraints	$C_1 \wedge C_2 \wedge C_3 \wedge C_4$	$\frac{1}{8X^6+(1-MX)} - (8X + 18X^2 16X^3 + 14X^4 + 12X^5 + 8X^6)$	$\frac{1}{X^6+(1-MX)} - (8X + 18X^2 16X^3 + 14X^4 + 12X^5 + 8X^6)$

Table 1: Translation of string constraints into SMC language then into GF for the running example. M is the alphabet size.

the cardinality of Ω^P can be represented as the sum of a series $a_0, a_1, ..., a_i, ...$ in which each a_i is the cardinality of the corresponding set S_i^P, i.e., $a_i = \left|S_i^P\right|$.

Ordinary Generating Functions. Given the representation of the cardinality of the string set Ω^P as the sum of the series $a_0, a_1, a_2, ..., a_n$, we can encode the cardinalities as an ordinary generating function, i.e., as the coefficients of a polynomial.

Definition 3.1. (Ordinary Generating Function) The *ordinary generating function (GF)* of the sequence $a_0, a_1, a_2, ..., a_i, ...$, is the summation

$$A(z) = \sum_{i \geq 0} a_i z^i. \tag{1}$$

We use the notation $[z^i]A(z)$ to refer to the coefficient a_i.

In many practical cases, we can represent these potentially infinitely long polynomials as finite expressions. Then, the Taylor series expansion of the finite expression around zero is equal to the polynomial. What makes this so powerful is that we can then reason about common string manipulations on potentially infinite sets of strings by computing on the (finite) generating function encoding of these coefficients. This is analogous to the common use of finite automaton to represent infinite languages and the manipulation of finite automata in order to manipulate the language represented by the automaton.

To compute a coefficient a_k from $A(z)$, we can use the following formula whose correctness follows directly from a Taylor series expansion [45]:

$$a_k = \frac{A^{(k)}[0]}{k!} \tag{2}$$

in which $A^{(k)}(z)$ is the k-th derivative of $A(z)$.

For example, consider the set of strings consisting of numerical digits. We have one (empty) string of length 0, 10 strings of length 1, 10^2 strings of length 2, and so on. Thus, the cardinality of all such subsets could be represented as the sequence $1, 10, 10^2, 10^3, ..., 10^i, ...$ in which 10^i is the number of strings of size i. If we encode the members of this sequence using the coefficients of a polynomial, we obtain the polynomial $G(z) = \sum_{k \geq 0} 10^k z^k$. We can then represent this infinitely long polynomial using the finite expression $H(z) = \frac{1}{1-10z}$ as H's Taylor series expansion around zero is equal to $G(z)$ [3]. If we apply Equation 2 to $H(z)$, we find that the cardinality of the set of strings of length i is exactly 10^i as expected.

To make the utility of GFs clear, consider the problem of prepending either the character '-' or the character '+' to a numerical string. We can represent the set of initial characters { '-', '+' } with the new GF $F(z) = 2z$. To obtain the number of strings that can be constructed in this fashion, we note that we take an arbitrary numerical string of length i (of which there are 10^i) and prepend either '-' or '+' to give us a 2×10^i strings of length $i + 1$. This corresponds to the polynomial:

$$\sum_{k \geq 0} 2 \cdot 10^k z^{k+1}$$

In the GF domain, operations on the closed form GF and on the normal GF are equivalent, thus we obtain:

$$\sum_{k \geq 0} 2 \cdot 10^k z^{k+1} = 2z \sum_{k \geq 0} 10^k z^k = F(z)G(z)$$
$$= F(z)H(z) = \frac{2z}{1 - 10z}$$

This is an example of a more general rule — to count the number of strings that can be created by concatenating strings from two disjoint sets, we simply multiply the GFs for those sets. Section 3.2 extends this idea beyond concatenations to an expressive set of string operations.

Set Definitions. We next establish a few definitions that we will use throughout the remainder of this paper. For a set X, we use l_X and u_X to refer to a lower bound and an upper bound on its cardinality respectively, such that $l_X \leq |X| \leq u_X$. We use $L_X(z)$, $U_X(z)$ to refer to GFs corresponding to l_X and u_X. We denote Ω as the set of all values, M as the alphabet size and $G(z)$ as the GF that generates Ω.

Given an operation that combines two string sets A and B to create a new set, the following expressions provide a method for computing the cardinality of the new set. These expressions follow directly from set theory.

1. For set intersection, the lower bound is zero (disjoint sets) and the upper bound is cardinality of the smaller set (proper subsets). Formally, $l_{A \cap B} = 0$, $u_{A \cap B} = \texttt{MIN}(u_A, u_B)$.
2. For set union, the lower bound is the cardinality of the larger set (proper subsets) and the upper bound is the sum of cardinalities of both the sets (disjoint sets). Formally, $l_{A \cup B} = \texttt{MAX}(l_A, l_B)$, $u_{A \cup B} = (u_A + u_B)$.
3. $l_{A \circ B} = \texttt{MAX}(l_A \times |B_i|, l_B \times |A_i|)$, $u_{A \circ B} = (u_A \times u_B)$, in which X_i is the set of all strings of length i in set X. The formula for

[3] The reader may notice that series $G(z)$ converges only for $|z| < \frac{1}{10}$, and thus the closed-form expression $H(z)$ gives meaningful results for only those values of z. As our use of the expression $H(z)$ in the GF domain never requires its evaluation for non-zero z, convergence is not a problem of practical importance.

$l_{A \circ B}$ is quite simple but still gives the sound lower bound. We introduce a better formula later in section 3.2.

4. $l_{\neg A} = |\Omega| - u_A, u_{\neg A} = |\Omega| - l_A$

These formulas provide the basis of a procedure for computing the cardinality of sets defined by the constraints in the SMC constraint language.

Operating on Generating Functions. In the remainder of the paper, we will use several functions which operate on the GFs domain. These functions are `MINF`, `MAXF`, `DEDUP`, and `TRUNC` with their definitions as follows.

- An upper bound on the intersection of two string sets is the size of the smaller set. Thus we define

$$F(z) = \texttt{MINF}(F_1(z), F_2(z)) \text{ such as}$$

$$[z^i]F(z) = \texttt{MIN}([z^i]F_1(z), [z^i]F_2(z)) \; \forall i \geq 0$$

  This function returns the GF whose coefficient a_i for any particular z^i is the smaller coefficient of z^i in F_1, F_2. For example,

$$\texttt{MINF}(1 + z + z^2, 1 + 2z^2) = 1 + z^2$$

- A lower bound on the union of two string sets is the size of the larger set. Thus we define

$$F(z) = \texttt{MAXF}(F_1(z), F_2(z)) \text{ such as}$$

$$[z^i]F(z) = \texttt{MAX}([z^i]F_1(z), [z^i]F_2(z)) \; \forall i \geq 0$$

  This function returns the GF whose coefficient a_i for any particular z^i is the greater coefficient of z^i in F_1, F_2. For example:

$$\texttt{MAXF}(1 + z + z^2, 1 + 2z^2) = 1 + z + 2z^2$$

- In some cases, operations may have the potential of double counting strings. Given a GF that may double count strings of given length, we can obtain a conservative lower bound by setting the non-zero coefficients to 1. Thus,

$$\texttt{DEDUP}(F(z)) = \sum_{[z^i]F(z)>0} z^i$$

  The `DEDUP` function simply returns a new GF whose coefficient for z^i is set to 1 if $[z^i]F(z) > 0$. For example:

$$\texttt{DEDUP}(G(z)) = 1 + z + z^2 + z^3 + .. = 1/(1 - z)$$

- Many string operations establish a maximum length for a string. Thus we define the truncation operation $\texttt{TRUNC}(U(z), Number)$ as a polynomial comprising of only the $Number + 1$ first coefficients in the Taylor expansion of $U(z)$. For example, given the Taylor expansion of $G(z)$ is $\sum_{i \geq 0} 256^i z^i$, we have

$$\texttt{TRUNC}(G(z), 3) = 1 + 256z + 256^2 z^2 + 256^3 z^3$$

The `MINF, MAXF` functions can be extended to work with more than 2 arguments.

3.2 SMC Constraint Language

Figure 3 presents the SMC string constraint language. It is expressive enough to handle the real-world string manipulations performed by many applications written in C, C++, and JavaScript [40]. We present an approach that computes sound bounds on the number of strings that satisfy a set of SMC constraints C. Our procedure provides the following soundness guarantee — the number of strings satisfying C is always between the lower and upper bounds.

In our discussion of how we apply GFs to solve the SMC constraint language, we will use the following running example throughout the remainder of this section:

Formula	:=	$\neg$*Formula*
	|	*Formula*$\wedge$*Formula*
	|	*Formula*$\vee$*Formula*
	|	*CoreConstraint*
	|	*FullConstraint*
CoreConstraint	:=	$Var \in RegExp$
	|	$Var = Var$
	|	$Var = Var \circ Var$
RegExp	:=	*Character*
	|	ϵ
	|	*RegExp RegExp*
	|	*RegExp* | *RegExp*
	|	*RegExp**
FullConstraint	:=	`contains`*(Var, ConstString)*
	|	*Number* =`strstr`*(Var, ConstString)*
	|	`length`$(Var) \odot$ *Number*
	|	`length`$(Var) \odot$ `length`*(Var)*
	|	*Var* = *ConstString*
$\odot$	:=	$<\|\leq\|\geq\|>\|=$

Figure 3: Grammar for the SMC language interface. $\circ$ is the concat operation between two strings.

$$R_1\texttt{=(a|ab|ac|abc)} \wedge R_2\texttt{=(c|b|ab|bc|abc)} \wedge$$
$$Var_1 \in R_1 \wedge Var_2 \in R_2 \wedge$$
$$Var = Var_1 \circ Var_2 \wedge Var \in \texttt{(ab|c)}^*$$

Generating Bounds for Regex Matching Constraints. Regular expression (regex) matching is commonly performed by string manipulation code. There is a simple mechanism for transforming regular expressions (a formal description of a set of strings) into a closed form representation of the cardinality of these sets (generating functions). Given an *unambiguous* [4] regex R, the following lemma constructs a generating function for the number of strings matching R. A proof of this lemma can be found in [42].

Lemma 3.1. Let `A` and `B` be two *unambiguous* regular expressions. If `AB`, and `A|B` are *unambiguous* regexs, $A(z)$ is the GF that enumerates `A` and $B(z)$ is the GF that enumerates `B`, then

1. $A(z) + B(z) = (a_0 + b_0) + (a_1 + b_1)z + (a_1 + b_1)z^2 + ...$ is the GF that enumerates `A|B`,
2. $A(z)B(z)$ is the GF that enumerates `AB`, and
3. $\frac{1}{1-A(z)}$ expands to $1 + A(z) + Az^2 + Az^3 + ...$ is the GF that enumerates `A`*.

The above lemma gives a GF that precisely enumerates an arbitrary *unambiguous* regex. For example, the GF of the regex `a` is z ($a_1 = 1, a_i = 0$ for $i \neq 1$), since there is exactly one string and its length is 1. Applying the lemma, we obtain the GF corresponding to `a|b` and `ab` are $2z$ (two strings of length 1) and z^2 (one string of length 2) respectively. The GF for all strings created by an alphabet of size M is the GF for the regex .* or `(`c_1`|`c_2`| ... |`c_M`)`*. Applying the first and third rules, we obtain $G(z) = \frac{1}{1-Mz}$ as the GF for all strings from an alphabet of size M.

For an arbitrary regex A, Lemma 3.1 does not give a GF that exactly enumerates A. For example, consider an ambiguous regex `(a|b)`*`|ab`. The corresponding GF derived from Lemma 3.1 is $1/(1 - 2z) + z^2$. For string length, say, 2, the above GF gives 5 strings. However, actually only 4 strings (namely `aa, ab, ba, bb`) are possible. This imprecision arises because the string `ab` is counted twice for given ambiguous regex `(a|b)`*`|ab`. Thus, Lemma 3.1 gives a sound upper bound for A regardless of its ambiguity property. The GF corresponding to the sound lower bound of any regex A is $\texttt{DEDUP}(U(z))$.

[4] A regular expression is considered *ambiguous* if there is a string which can be derived from it in more than one way.

length	$Var_1 \circ Var_2$	$L(z)$	$U(z)$
2	2	2	2
3	5	4	6
4	7	4	7
5	4	2	4
6	1	1	1

Figure 4: Example for the concat ($\circ$) operator. The second column shows the exact number of strings of specific length while the third and forth column show the corresponding bounds computed by the GFs $L(z)$ and $U(z)$.

Bounds for a Single Constraint. We describe how to solve the three types of *CoreConstraint* constraints that appear in SMC (see Figure 3).

- For the constraint $Var \in RegExp$, we use Lemma 3.1 to compute the GFs for the lower bound and upper bound for Var. For example, the GF for the upper bound of $Var \in R_1$ is $U(z) = z + 2z^2 + z^3$ and the GF for the lower bound is $\texttt{DEDUP}(U(z)) = z + z^2 + z^3$.
- For the constraint $Var = Var_1 \circ Var_2$, we rewrite it in the same style as a regular expression to $Var \in (Var_1\, Var_2)$. Applying rule 2 of Lemma 3.1, we compute the $U(z)$ of Var by multiplying the two GFs for the upper bound of Var_1 and Var_2.
  To compute the $L(z)$ of Var, the simple but imprecise approach is to assign $L(z) = \texttt{DEDUP}(L_1(z) \times L_2(z))$. To obtain a more precise bound, we observe that number of possible strings of length k of Var is at least equal to
  $$\texttt{MAX}\{[z^i]L_1(z) \cdot [z^j]L_2(z)\}, \forall i \geq 0, j \geq 0 : k = i + j.$$
  From that observation, we calculate $L(z)$ iteratively using
  $$L(z) = \texttt{MAXF}(([z^i]L_{1/2}(z))z^i \times L_{2/1}(z), L(z))$$
  for a set of specific i. The number and the selection of the i terms can be tuned to trade off the precision of the lower bound with the analysis time. In other words, the more i terms used in the calculation, more precise the bound is, but the longer the execution time. One important note, if there are constraints on the contents of both Var and Var_1 or Var_2, we can no longer reason about the lower bound using generating functions. In this case, SMC assigns a lower bound of 0 to the respective sets to maintain soundness.
  In our aforementioned example, we compute the GFs for the lower bound and upper bound of Var_1, Var_2 to be
  $$L_1(z) = U_1(z) = z + 2z^2 + z^3$$
  $$\text{and } L_2(z) = U_2(z) = 2z + 2z^2 + z^3$$
  respectively (since both R_1 and R_2 are *unambiguous* regexs). The GF for the upper bound of Var is then
  $$U(z) = U_1(z) \cdot U_2(z) = z^6 + 4z^5 + 7z^4 + 6z^3 + 2z^2$$
  To compute the lower bound, if we choose three values of i as 1, 2 and 3, we get
  $$L(z) = \texttt{MAXF}(zL_2(z), 2zL_1(z), 2z^2L_2(z), ..., z^3L_2(z))$$
  $$\text{or } L(z) = 2z^2 + 4z^3 + 4z^4 + 2z^5 + z^6$$
  Figure 4 presents a comparison between the computed bounds and the actual results for string lengths between 2 and 6 of Var.
- The canonical translation step described below eliminates constraints of the form $Var = Var'$ by replacing all uses of Var with Var'.

Bounds for Formulas. We next describe how SMC combines the bounds for individual constraints to infer bounds for formulas over multiple constraints. We analyze formulas over constraints using the following three rules:

- $Formula := Formula_1 \wedge Formula_2$. It follows from set theory that a conservative GF for the lower bound of $Formula$ is 0 and for the upper bound is $U(z) = \texttt{MINF}\ (U_1(z), U_2(z))$. For example, consider the constraint $S \in \{a, ab\} \wedge S \in \{ab, abc\}$. We combine the GFs for the individual constraints, which are $z + z^2$ and $z^2 + z^3$, using this rule to obtain z^2.
- $Formula := Formula_1 \vee Formula_2$. For a disjunction, the GF for the lower bound is $L(z) = \texttt{MINF}(L_1(z), L_2(z))$ and the GF for the upper bound is $U(z) = U_1(z) + U_2(z)$.
- $Formula := \neg Formula_1$. We apply the set theory to derive the GF corresponding to the upper bound of $Formula$ to be $U(z) = G(z) - L_1(z)$. Thus, the GF associated with the lower bound of $Formula$ is $L(z) = G(z) - U_1(z)$.

Canonicalizing Constraint Formulas. SMC next translates the input formulas into a canonical representation by the following steps:

1. **CNF translation:** SMC first translates the formula into conjunctive normal form (CNF), i.e., an AND of ORs.
2. **Constraint Rewriting:** SMC next finds all clauses that consist of a single variable equality constraint and eliminates the those constraints by replacing all appearances of the variable on the left side with the variable on the right hand side. If there is a clause that contains more than one equality constraints, SMC removes the clause and will set the lower bounds of all variables in the clause to 0.
3. **Clause Grouping:** SMC next groups clauses together based on the variables that appear in the constraints that comprise the clause. If a clause contains multiple constraints on different variables, SMC rewrites the clause into several clauses. For example, consider the clause $A \vee B$ in which A and B are constraints on separate variables X and Y respectively. The lower bound of X (Y) is the lower bound computed from A (B) (by assuming A (B) is true), while its upper bound is $G(z)$, the whole space, by assuming nothing (if we do not care about the value of X (Y), then A (B) can have any value).
4. **Concat Graph Construction and Topological Sort:** Constraints in one group can depend on the results of constraints from another group. SMC constructs a constraint group dependence graph and topologically sorts the groups of constraints.
5. **Solving Constraints:** Finally, SMC analyzes the constraints in topological order. Bounds for constraints inside a clause are computed directly from the constraint formula. If there is more than one constraints in a clause, the clauses are combined with the rule for $\vee$. Finally, the rule for $\wedge$ is used to combine the results for all constraints inside a constraint group.

3.3 Avoiding Imprecision

The *CoreConstraint* constraint subset is expressive enough to compute sound cardinality bounds for string manipulations in real programs. However, the calculated bounds unnecessarily lose precision if the constraint solving rules are composed in a naïve order. For example, suppose we wish to count the number of strings in the set S that do not contain the string patterns P_1 or P_2. If we count in a way such that the $\wedge$ inference rule is applied, we lose considerable precision. Specifically, if we translate $\neg S \in (.^*P_1.^*)$, we obtain $U_1(z) = G(z)$ and $L_1(z) = G(z) - P_1(z)$ in which $P_1(z)$ is the upper bound GF corresponding to $(.^*P_1.^*)$. Similarly, we obtain $U_2(z) = G(z)$ and $L_2(z) = G(z) - P_2(z)$ for $\neg S \in (.^*P_2.^*)$ constraint. Using the $\wedge$ inference rule in the end, the GF for the upper bound and lower bound of S is $U(z) = G(z)$ and $L(z) = 0$. While these bounds are sound, they are very imprecise as they are simply the initial bounds for S.

Intuitively, desugaring into the *CoreConstraint* constraint subset loses precision because the transformation to generating functions

				a	b	c	a	b	c_i
				a	b	c	a	b	1
			a	b	c	a	b		0
		a	*b*	c	a	b			0
	a	*b*	*c*	a	b				1
a	*b*	*c*	*a*	b					0

Table 3: Auto-correlation of string `abcab`. As computed in the table, the corresponding polynomial is $c(z) = x^3 + 1$.

loses information about the content of the strings. Preserving information about the content enables a more precise reasoning about the bounds on the cardinality of string sets. Consider the previous example problem of counting strings S that do not contain either P_1 or P_2. To compute this without losing precision, we can leverage correlations between P_1 and P_2 to obtain more precise bounds on S, and not lose the content information of P_1 and P_2 by eagerly transforming to GFs of these sub-operands.

Using this intuition, we introduce a set of higher-level operators, which allow picking the right rule applications — these operators correspond directly to their counterparts in common programming languages such as `C` and JavaScript, such as `contains` and `strstr`. We present these higher-level constraints as the *FullConstraint* language, which is an extension of our simple *CoreConstraint* subset, and explain how to compose them.

Computing Bounds for SMC ***Constraints.*** We next present how SMC analyzes the higher-level *FullConstraint* string constraints.

- `contains` **Constraints:** The `contains` operation checks if the string pattern $p = p_0p_1...p_{n-1}$ appears in the string s. The generating function F_0 that enumerates precisely the size of the set of strings s that do not contain p is given by the following equation:

$$F_0(z) = \frac{c(z)}{z^n + (1 - Mz)c(z)} \tag{3}$$

  where $c(z)$ is the *auto-correlation polynomial* of p.
  The auto-correlation polynomial $c(z)$ of the string $p = p_0p_1...p_{n-1}$ is the summation

$$c(z) = \sum_{0 \le i < n} c_i z^i,$$

  where

$$c_i = \begin{cases} 1 & \text{if } 0 \le \forall j \le n - 1 - i, p_j = p_{j+i} \\ 0 & \text{otherwise} \end{cases}$$

  Sedgewick *et al*. provide a proof for Equation 3 [42]. Subtracting the GF F_0 for string set not containing p from the GF $G(z)$ for Ω yields the GF $F_1(z)$ for the set of strings that contain p:

$$F_1(z) = G(z) - F_0(z) = \frac{z^n}{(1 - Mz)[z^n + (1 - Mz)c(z)]} \tag{4}$$

  The example of $c(z)$ for string $p =$ `abcab` is shown in Table 3. As in Equation 4, the GF that enumerates all ASCII strings containing p is

$$F_s(z) = \frac{z^5}{(1 - 256z)[z^5 + (1 - 256z)(1 + z^3)]}$$

  From $F_s(z)$, by applying Lemma 2, we get $a_1 = a_2 = a_3 = a_4 = 0$, as there is no string of length from 0 to 4 that contains `abcab`. We also get $a_5 = 1, a_6 = 512, ...$ as the number of strings containing S of length 5, 6 respectively.
- `strstr` **Constraints:** The `strstr` operator returns the position t of the first occurrence of a constant pattern $P = p_0p_1...p_{n-1}$ in the string S. If there is no such position t, `strstr` returns a negative number. A constraint on a `strstr` operation gives both the constant pattern and the position returned by the operation.

  If t is negative, we translate the constraint into an equivalent negated `contains` constraint. For $t \ge 0$, we use the following equation to compute an exact GF for the size of the set S:

$$F[P, t, n](z) = E\frac{z^{n+t}}{1 - Mz} \tag{5}$$

  in which

$$E = [z^t]F_0(z) - \sum_{0 < i \le t, C[i] = 1} [z^{n+t-i}]F[P, t - i, n](z)$$

  The GF $F_0(z)$ enumerates the set of strings that do not contain P and C is the auto-correlation table for P. We represent S as $A \circ P \circ B$ with conditions that $\texttt{length}(A) = t$ and there is no occurrence of P in $A \circ P$ other than the postfix P. Note that the latter condition is not equivalent to $\neg A.\texttt{contains}(P)$ because there are cases when a postfix of A combined with a prefix of P produce a string that contains P. In other words, if there exist some $C[i] = 1 (0 < i \le t)$ in the auto-correlation table C for P, we have to eliminate all A ending with $P[0..i - 1]$ that have no occurrence of P. The number of such A is equal to the number of $S' = A \circ P[i..n - 1]$ that satisfy the $\texttt{strstr}(S', P) = t - i$.
- `length` **Constraints:** The `length` operation returns the length of a string. Given the set of strings $S = \bigcup_{i \ge 0} S_i$, whose cardinality is represented by GF $F(z)$, the `length` operators of S decide which S_i is eliminated from S or the *i-th* coefficient in Taylor expansion of $F(z)$ is constrained to 0. For example, GF for S with no constraints is $G(z) = \frac{1}{1-Mz}$. The constraint $\texttt{length}(s) > 2$ restricts S to $\bigcup_{i>2} S_i$ and eliminates S_0, S_1, and S_2. The new GF for S is $G'(z) = \frac{1}{1-Mz} - 1 - Mz - M^2z^2$ which gives $a_0 = a_1 = a_2 = 0$. This is where the `TRUNC` function is used. Table 2 presents inference rules for `length` constraints as well as the SMC constraint language.

Analyzing Combinations of Constraints. We next discuss how the SMC tool combines certain types of constraints together and analyzes them as a group in order to improve precision. SMC combines constraints in the following cases:

- **Conjunctions of** `contains` **Constraints:** SMC groups together `contains` constraints and analyzes the combination at once to improve the precision of the lower bound. The generating function for the lower bound of the number of strings not containing any of the string patterns $P_1, P_2, ..., P_N$ is given by:

$$B(z) = \frac{\prod_{i \le N} C_i(z)}{D + (1 - Mz)\prod_{i \le N} C_i(z)}, \tag{6}$$

  where

$$D = \sum_{i=1}^{N} Z^{|P_i|} \times \frac{\prod_{j \le N} C_j(z)}{C_i(z)}$$

  $|P_i|$ is the length of P_i

  $C_i(z)$ is the auto-correlation polynomial of P_i

  Equation 6 can be proven correct under the assumption that the patterns have no overlap by a straightforward extension of the proof given by Sedgewick *et al*. [42]. If the patterns have overlap, the GF $B(z)$ undercounts the number of strings and remains a conservative lower bound. The GF $B(z)$ is a tighter lower bound than the one computed by separately transforming each constraint by using the $\wedge$ inference rule. To make this more concrete, let's revisit our previous example of the set S that does not contain P_1 or P_2. By separately and eagerly translating the constraints into generating functions, we represent S as $\neg S.\texttt{contains}(P_1) \wedge \neg S.\texttt{contains}(P_2)$. Then GFs for the

Expression	L(z)	U(z)
$Var \in RegExp$	$\texttt{DEDUP}(U_{RegExp}(z))$	$U_{RegExp}(z)$
`contains`*(Var, ConstString)*	$\frac{z^n}{(1-Mz)[z^n+(1-Mz)c(z)]}$	
$\neg$ `contains`*(Var, ConstString)*	$\frac{c(z)}{z^n+(1-Mz)c(z)}$	
$Number =$ `strstr`*(Var, ConstString)*	As in Equation 5	
$Var = Var_1 \circ Var_2$	As discussed in Section 3.2	$U_1(z) \times U_2(z)$
$\texttt{length}(Var) > Number$	$L(z) - \texttt{TRUNC}(L(z), Number)$	$U(z) - \texttt{TRUNC}(U(z), Number)$
$\texttt{length}(Var) \leq Number$	$\texttt{TRUNC}(L(z), Number)$	$\texttt{TRUNC}(U(z), Number)$
$\texttt{length}(Var) = Number$	$z^{Number}([z^{Number}]L(z))$	$z^{Number}([z^{Number}]U(z))$
$Formula = \neg Formula$	$G(z) - U_1(z)$	$G(z) - L_1(z)$
$Formula = Formula_1 \wedge Formula_2$	0	$\texttt{MINF}\ (U_1(z), U_2(z))$
$Formula = Formula_1 \vee Formula_2$	$L(z) = \texttt{MAXF}(L_1(z), L_2(z))$	$U(z) = \texttt{MINF}\ ((U_1(z) + U_2(z), G(z))$

Table 2: Inference Rules for the SMC Constraint Language.

bounds of S computed by Equation 4 and the inference rule $\wedge$ are $L(z) = 0$ and $U(z) = \texttt{MINF}(U_1(z), U_2(z))$.
With Equation 6, we improve the lower bound of S to $L(z) = B(z)$, which is more precise than 0.

- **Conjunctions of $\notin$ Constraints:** Given a conjunction of $\notin$ constraints $\neg S \in R_1 \wedge \neg S \in R_2 ... \wedge \neg S \in R_n$, SMC translates the formula into the representation $\neg S \in (R_1 \mid R_2 \mid ... \mid R_n)$. Since $(R_1 \mid R_2 \mid ... \mid R_n)$ is a regular expression, SMC can then apply the inference rule for $S \in Regex$ and $Formula = \neg Formula$ to produce a better lower bound.
- **Conjunctions of $\in$ Constraints:** To handle a conjunction of $\in$ constraints $S \in R_1 \wedge S \in R_2 ... \wedge S \in R_n$, SMC first translates all regular expression R_i to their corresponding finite state automaton (FSA). After that, it computes the product automaton (or the intersection automaton) of all n generated FSAs and re-translates the product automaton into a final regular expression R_p. The number of S satisfying the given constraints equals the number of S in $S \in R_p$. SMC finally applies the $\in$ inference rule to $S \in R_p$ and returns a more precise upper bound.

3.4 Sound Approximations

The `MINF`, `MAXF`, and `DEDUP` operators are formalized in terms of the individual coefficients and SMC cannot directly express these operators in closed form. SMC instead leverages Mathematica to compute the first N (configured by users and should be the maximum length of the input) coefficients and generates a polynomial approximation of the generating function. We note that the `MAXF` and `DEDUP` operators are only used in the context of lower bounds, and thus it is conservative to truncate the remaining terms. The `MINF` function is used in the context of computing upper bounds. In this case, we conservatively replace the truncated portion of the generating function with terms from one of the input generating functions. Another improvement which requires losing closed form is the $\wedge$ rule. We notice that $A \wedge B \geq \texttt{MAX}(A + B - \Omega, 0)$, hence we set $L_{A \wedge B} = \texttt{MAXF}(L_A + L_B - \Omega, 0)$ to obtain a better lower bound for the $\wedge$ inference rule than zero.

4. Implementation and Evaluation

We implement the SMC model counter design as a tool in C and Python with 2,680 and 1,160 lines of code respectively. SMC queries Mathematica 9.0.1 through the Mathlink interface for symbolic manipulation [9, 50]. The tool takes as input a set of string constraints and translates them into GFs. After this, SMC solves for the coefficients in response to client's analysis queries. In our experience, the framework can efficiently solve for the coefficients of 1,000 character strings in less than a second. Our SMC implementation is open source and available online [13].

First, we show how SMC is useful in a number of quantitative and qualitative analyses through case studies. Second, we directly compare the precision and performance of our technique to publicly available quantitative analysis tools in Section 4.3. Finally, we demonstrate the expressiveness of the SMC constraint language in handling counting queries of path constraints extracted from real-world JavaScript applications in Section 4.4. All our experiments were conducted on 64-bit Ubuntu version 12.04 with a 2.9GHz i7-3520M processor and 8 GiB memory. All reported times are averaged over 10 runs.

4.1 Case Study: Password Strength Meters

SMC can be used to analyze the comparative efficacy of the password strength meters employed in real systems. We study the `obscure` utility used on UNIX systems and three password strength meters employed in the Microsoft password checker page [10], the Ebay website [4] and the Drupal web application [3].

obscure. The `obscure` utility checks if a new password passes a number of security checks. Our running example describes the part of `obscure` that leaks information about the current password. An attacker who can infer the executed path in `obscure` can reduce his uncertainty about the new password, given the knowledge of the old password. Consider the path executed by the inputs `new_p ="abc@!xy"` and `old_p = "z"` (same execution path as the running example in Figure 1), where the attacker aims to guess the number of possible solutions for `new_p`. The constraints are simple and SMC returns the bounds within 1 second. SMC reports an upper bound of 91,195,154,067,474,198,297,600 and a lower bound of 46,402,149,423,095,961,815,551 for the number of new passwords of length 10 that are not similar to the old password (`"z"`). Through manual calculations, we confirm that the bounds are sound, and in fact, the true value equals the upper bound reported by SMC.

We extend the study of password strength meters to popular web services which encourage users to choose better passwords. In this experiment, we first compute the total number of acceptable passwords for each strength scale (weak, fair, good / medium and strong) by the following three websites: Ebay [4], Drupal [3], and Microsoft [10] (the logic for which was reported in a recent paper [24]). We find that there is considerable variance in the number of passwords acceptable to each site's policy. This can be used to estimate the effort in a password guessing attack against each website, for instance, in computing the attacker's guessing entropy [44]. Next, we measure the number of passwords that intersect with the popular password database dictionary of 3,106 words available from the `John-the-Ripper (JtR)` password cracking tool [7]. The larger the intersection set size, the greater the number of dictionary words permitted to be passwords despite the strength checks. This can be used to directly compare the password strength meters used by our subject sites relative to the `JtR` dictionary set. The result of this experiment is summarized in Table 4.

Building the Strength Meter Model. Given the design choices of three meters, we build a regular expression to represent all the possible passwords acceptable by each policy. Largely, these

	Strength		L = 1	L = 2	L = 3	L = 4	L = 5	L = 6	L = 7	L = 8	L = 9	L = 10	Total
Ebay	Invalid	JtR	7	4	80	295	505	932	467	317	58	13	2678
		Ω	72	1552	37152	$9.34 \cdot 10^5$	$2.4 \cdot 10^7$	$6.2 \cdot 10^8$	$1.61 \cdot 10^{10}$	$4.18 \cdot 10^{11}$	$1.09 \cdot 10^{13}$	$2.82 \cdot 10^{14}$	$2.94 \cdot 10^{14}$
	Weak	JtR	0	0	0	1	43	185	115	64	4	1	413
		Ω	0	3632	$2.24 \cdot 10^5$	$1.14 \cdot 10^7$	$5.53 \cdot 10^8$	$2.67 \cdot 10^{10}$	$1.29 \cdot 10^{12}$	$6.35 \cdot 10^{13}$	$3.15 \cdot 10^{15}$	$1.58 \cdot 10^{17}$	$1.62 \cdot 10^{17}$
	Medium	JtR	0	0	0	0	0	0	2	1	0	0	3
		Ω	0	0	$1.12 \cdot 10^5$	$1.29 \cdot 10^7$	$1.07 \cdot 10^9$	$7.73 \cdot 10^{10}$	$5.28 \cdot 10^{12}$	$3.48 \cdot 10^{14}$	$2.26 \cdot 10^{16}$	$1.45 \cdot 10^{18}$	$1.47 \cdot 10^{18}$
	Strong	Ω	0	0	0	$1.62 \cdot 10^6$	$2.92 \cdot 10^8$	$3.47 \cdot 10^{10}$	$3.44 \cdot 10^{12}$	$3.1 \cdot 10^{14}$	$2.63 \cdot 10^{16}$	$2.14 \cdot 10^{18}$	$2.17 \cdot 10^{18}$
Microsoft	Weak	JtR	7	4	80	296	549	1119	585	0	0	0	2640
		Ω	72	5184	$3.73 \cdot 10^5$	$2.69 \cdot 10^7$	$1.93 \cdot 10^9$	$1.39 \cdot 10^{11}$	$1 \cdot 10^{13}$	0	0	0	$1.02 \cdot 10^{13}$
	Medium	JtR	0	0	0	0	0	0	0	385	62	14	461
		Ω	0	0	0	0	0	0	0	$7.22 \cdot 10^{14}$	$5.2 \cdot 10^{16}$	$3.74 \cdot 10^{18}$	$3.8 \cdot 10^{18}$
Drupal	Weak	JtR	7	4	80	296	549	0	0	0	0	0	936
		Ω	72	5184	$3.73 \cdot 10^5$	$2.69 \cdot 10^7$	$1.93 \cdot 10^9$	0	0	0	0	0	$1.96 \cdot 10^9$
	Fair	JtR	0	0	0	0	0	1119	467	317	58	13	1974
		Ω	0	0	0	0	0	$6.2 \cdot 10^8$	$1.61 \cdot 10^{10}$	$4.18 \cdot 10^{11}$	$1.09 \cdot 10^{13}$	$2.82 \cdot 10^{14}$	$2.94 \cdot 10^{14}$
	Good	JtR	0	0	0	0	0	185	115	64	4	1	369
		Ω	0	0	0	0	0	$2.67 \cdot 10^{10}$	$1.29 \cdot 10^{12}$	$6.35 \cdot 10^{13}$	$3.15 \cdot 10^{15}$	$1.58 \cdot 10^{17}$	$1.62 \cdot 10^{17}$
	Strong	JtR	0	0	0	0	0	0	2	1	0	0	3
		Ω	0	0	0	0	0	$1.12 \cdot 10^{11}$	$8.72 \cdot 10^{12}$	$6.58 \cdot 10^{14}$	$4.88 \cdot 10^{16}$	$3.59 \cdot 10^{18}$	$3.63 \cdot 10^{18}$

Table 4: Summary of results for different password meter strength scales in Ebay, Microsoft and Drupal. For string length L, Ω is the total possible passwords and JtR stands for passwords that are intersected with in John the Ripper database.

meters measure the strength of a password based on its length and the number of character sets (lowercase, uppercase, symbols and digits) it includes. Ebay classifies a password as invalid, weak, medium or strong if it has one, two, three or four different character sets, respectively. Microsoft classifies a given password as weak, medium or strong based on its length (0-8, 8-13, 13 and above) and very strong if it contains all four character sets. Drupal rejects all passwords of length smaller than 6 and considers them as weak. It also checks the number of character sets that the password includes to measure its strength. In addition to these differences, the set of allowed symbol characters are slightly different in each meter. Given these designs, we build the regular expression for each scale for all three websites. For example, the regular expression for invalid passwords in Ebay meter (as described by the policy [4]) is `[a-z]*|[A-Z]*|[0-9]*|[ @#? ^ &*-+]*`. We pass this regular expression to SMC and compute the size of feasible set. Our tool gives precise result for all scales in Ebay, Drupal and Microsoft in 0.314, 0.350 and 0.329 seconds respectively.

Comparing the Password Meter's Strength. We compute how many passwords in the JtR set comprising of 3,106 dictionary words satisfy each policy. We first synthesize a regular expression of JtR set as R_1 by getting a union of all passwords. Thus, to compute the number of passwords satisfying a particular policy represented by a regular expression R_2, we simply give the two regular expressions to SMC. Our tool measures the intersection of R_1, R_2 and returns the GFs for the bounds of feasible set in 228.86, 206.10 and 4.48 seconds for Ebay, Drupal and Microsoft, respectively. Note that a large fraction of this time is spent for computing the complex regex intersection. Our upper bounds are exactly equal to the precise numbers of passwords computed by actually checking each of the passwords against the strength meters. We report that Ebay rejects a majority of the passwords in the set and rates only 3 of them as medium. There is no JtR dictionary word that qualifies as a strong or best password in Microsoft meter, since no password in the database has a length greater than 13. On the other hand, most of the JtR's passwords are considered fair or good passwords as per the Drupal policy. Although Drupal does further check with the registered user's account, it is more likely that JtR dictionary attack will succeed on their site than on the other two. Table 4 presents more detailed results for each strength meter.

4.2 Case Studies: UNIX utilities

One application where SMC could be useful is quantifying leakage in systems that compute functions over encrypted data — for example, to mitigate the impact of data breaches in public cloud-hosted applications [38, 46]. Concretely, the recently proposed AutoCrypt system transforms legacy applications to operate on encrypted inputs by using partially homomorphic encryption techniques [46]. After the transformation, operations on encrypted inputs are transformed to homomorphic computation on encrypted text (e.g. string match using searchable encryption). While each individual homomorphic operation is privacy-preserving, a powerful adversary can learn the order of executed operations, and thus the execution path[5]. This can lead to information leakage about the encrypted file content [6]. SMC can be used to quantify this leakage precisely for any given input.

We quantify leakage in 2 string manipulation utilities from the BUSYBOX v.1.21.1 package (`wc` and `grep`) [1], and one utility from the COREUTILS v.8.21 package (`csplit`) [2]. For these file processing utilities, we use one concrete input file as a sample input and measure how much information would be leaked if they operate on homomorphically encrypted inputs as in AutoCrypt [46]. The first section of Table 5 presents the results. For these calculations, we assume that all the inputs are equally likely in the universal set. That is, they are drawn randomly from an uniform probability distribution. Thus, the minimum number of bits leaked is computed as $log_2 G - log_2 U$ [43], where G is number of all possible files and U is the upper bound of SMC's model count.

grep. We run `grep` with the $-o$ option to find all occurrences of the pattern "information" in an encrypted input file. The input file consists of 11 lines (total 629 bytes) containing 3 occurrences of the pattern — on Lines 4, 8, and 11. We assume that all string operations, such as substring match, are converted to their equivalent privacy-preserving homomorphic operation on encrypted data. By observing the execution path of `grep`, an attacker learns how many and which specific lines in the input file contain the searched pattern. We ran `grep` with SMC and it reports in 48.9 seconds that an adversary can infer at least 268.1 bits of information by learning

[5] This could be possible if the cloud hypervisor is malicious and can monitor the control flow, or via measurements of side channels such as execution timing by malware running in the VM.

[6] For a more detailed explanation, we refer readers to the reduced indistinguishability property [46]

Program	Input size (bytes)	Output leakage (bits)	No. of constr-aints	Path leakage								
				Min leakage (bits)	Normal version			Precise version			Other tools	
					Max leakage (bits)	Time (s)	ϵ-precision	Max leakage (bits)	Time (s)	ϵ-precision	Leakage (bits)	Time (s)
Test with 4 utilities												
`obscure`	10	0.113	5	0.057	0.113	0.3	0.001	0.113	0.3	0.001	N/A	N/A
`grep`	629	231.9	32	268.1	≈ 5032	48.9	0.95	355.3	227.8	0.017	N/A	N/A
`wc`	629	299.3	534	747.0	≈ 5032	185.0	0.85	750.1	214.7	0.001	N/A	N/A
`csplit`	629	96.0	44	110.6	≈ 5032	39.1	0.98	179.4	147.3	0.014	N/A	N/A
Comparison with `FuzzBALL`												
`obscure`*	6	≈ 0	93	≈ 0	≈ 0	0.5	0.06	≈ 0	0.5	0.06	#	2 hr
`strstr(input, "abc")!=NULL`	5	22.4	1	22.4	22.4	0.4	0	22.4	0.4	0	#	2 hr
`strstr(input, "abc")!=NULL`	4	23.0	1	23.0	23.0	0.4	0	23.0	0.5	0	13.5	150
`match_regex(input, "`$(a\|b)^*$`")`	4	28.0	1	28.0	≈ 32	0.4	0.13	≈ 32	0.4	0.13	#	2 hr
Comparison with Castro *et al*. [23]												
`Ghttpd`	620	N/A	8	63.4	4960	9.2	0.99	80.2	45.6	0.003	≈ 248	≈ 1
`Null HTTPd`	500	N/A	6	239.1	4000	4.6	0.95	248.0	17.4	0.002	≈ 500	≈ 8
Comparison with `QUAIL`												
`strstr(input, "ab")=2`	5	16.00	1	16.00	16.00	0.2	0	16.00	0.2	0	16.00	6.1
`strstr(input, "ab")=2`	7	16.00	1	16.00	16.00	0.2	0	16.00	0.2	0	16.00	648
`input.contains("ab")`	5	14.00	1	14.00	14.00	0.3	0	14.00	0.3	0	14.00	5.1
`input.contains("ab")`	7	13.42	1	13.42	13.42	0.3	0	13.42	0.3	0	13.42	606

N/A : Not available # : The tool did not terminate in 2 hours. *: Executed with inputs discussed in Section 2.3

Table 5: Summary of evaluation results of SMC for UNIX utilities and comparison to previous works. Normal, precise - SMC are running without and with concatenation optimization respectively (see Section 3.2).

the executed path; i.e. over all possible $(2^8)^{629}$ possible files, the knowledge of the executed path reduced the uncertainty by $2^{268.1}$. If the attacker learns just the number of occurrences, not the specific lines, the leakage is 231.9 bits.

wc. The `wc` utility counts the number of words, new line characters as well as total number of characters in the input file. Under the same encrypted file used in the `grep` case study, we ran the `wc` utility. The input file has 11 newline characters and 77 spaces. By observing the executed path, the attacker can learn the positions of the newlines and spaces, thereby learning some information about the encrypted input. The executed path consists of 534 constraints, and SMC computes the cardinality of the solution set in roughly 215 seconds (about 3.5 minutes). The leakage (or reduction in uncertainty) is reported to be between 747.0 to 750.1 bits. If the attacker only knows the number of line-breaks and spaces, not their precise positions as gleaned from the program execution path, the leakage is roughly 3 times lower. Thus, the execution path reveals significantly more information than one might naïvely expect.

csplit. The `csplit` utility splits the input file into two parts: one has no occurrence of the input pattern and other starts with the pattern. `csplit` returns two numbers a, b as the size of each file. If an attacker obtains a and b, he can infer the file content as a string S of length $a + b$ and the first appearance of the pattern in S is at position a. We choose the split pattern as the word "information" and `csplit` returns a and b as 206 and 423 respectively. In terms of SMC's constraints, it is equivalent to $\texttt{strstr}(S, \texttt{"information"}) = 206$ and $\texttt{strlen}(S) = 629$. SMC reports a precise value of 96.0 bits of leakage for input file content. If the attacker knows the exact execution path, he learns the specific line number and the location of pattern used to split the file. In such a case, the leakage is higher — SMC reports it to be 110.6 to 179.4 bits. The total number of constraints in this calculation is 44, which takes 147.3 seconds to compute.

4.3 Comparison to Existing Techniques

Comparison with `FuzzBALL`**.** In their previous work, Newsome *et al*. presented a technique to measure the quantitative influence of an input on the program output based on the size of the feasible output value set [36]. This technique is employed in `FuzzBALL` and enables us to compute the number of possible outputs in a program. By considering the input as if it were an output, i.e. assigning the output as input at the end of execution, we use `FuzzBALL` to quantify the number of inputs that drive the execution to a specific path. We use `FuzzBALL` and SMC to calculate the cardinality of several string sets. One limitation of `FuzzBALL` when measuring influence is that it currently does not work with output values that are larger than a single Vine "register" (64 bits) [5]. SMC can work with arbitrary size inputs and report results for strings that contain thousands of characters. To compare the precision and performance of the two tools, we selected benchmarks with input lengths not larger than 8. Section 2 in Table 5 presents results for all test cases. These results show that SMC is faster and returns more precise results than `FuzzBALL`. Out of 4 small test cases, `FuzzBALL` did not return results for 3 of them after 2 hours of running, while SMC ran within 0.5 seconds for every test. SMC computed the sound bounds of leakage in all test cases and returned precise leakage for 2 of them (precise of $\epsilon = 0$). `FuzzBALL`, on the other hand, did not return a precise result for any test case.

Comparison to Privacy-preserving Bug Reporting. Castro *et al*. examined the `Ghttpd` and `Null HTTPd` servers for better bug reporting [6, 11, 23]. They provided a solution to enhance users' privacy in the bug-reporting process by obfuscating the bug report. One part of their work informs the users of how much information is leaked in the obfuscated content. They also compute an upper bound of privacy loss as SMC does, but their technique differs. First, they calculate a bound for each input byte and then combine the results to obtain an overall bound. We ran both `Ghttpd` and `NULL HTTPd` with the path condition corresponding to the input used by Castro *et al*. [23]. The bounds computed by SMC are consistent with their upper bounds. When computed by the precise version, our upper bounds are smaller and give a better range.

Comparison to QUAIL. QUAIL, a recently introduced tool with a customized input language, allows analysts to supply probability distributions for the inputs and computes the output probability distribution [20]. QUAIL's input language supports binary operators, integer arithmetic as expressions, and array datatypes, thereby supporting imperative code constructs. Internally, it builds a Markov chain model for input programs. SMC, in contrast, axiomatizes

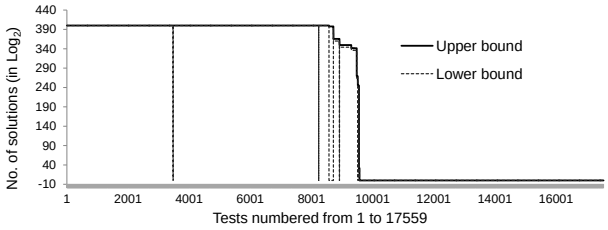

Figure 5: Bounds on no. of solutions for `Kaluza`'s small test cases

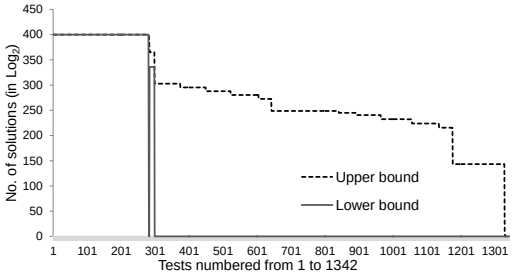

Figure 6: Bounds on no. of solutions for `Kaluza`'s big test cases

the complex string functions and employs GFs as its mechanism. To evaluate the QUAIL tool (available online), we hand-coded a simple implementation of `strstr` and `contains`. Under the input `strstr (input, "ab")=2` or `input.contains("ab")`, QUAIL produces a result after 10 mins for an input size of 7 characters. But for input size 20, it did not finish even after 2 hours. SMC, however, finished and returned the computed leakage within one second. Both tools are equally precise in this case study.

4.4 Expressiveness: JavaScript Benchmarks

To evaluate the expressiveness of SMC's constraint language, we tested it with a large set of publicly available benchmarks which are path conditions from real-world JavaScript application traces. The path conditions were originally collected by `Kudzu` and translated into the constraint language of the `Kaluza` string solver [40]. To work with these benchmarks, we wrote a wrapper to translate the `Kaluza` constraint language to the SMC language. Of the 18,901 satisfiable cases in the benchmark, SMC handles and reports the number of satisfiable strings for 17,559 test cases marked "small" and 1,342 "big" cases in the original benchmarks.

4.4.1 Results

We run the model counting analysis for the both small and big benchmarks with the maximum string length limited to 50.

Small test cases. The results for the small test cases are relatively precise since the constraints are quite simple and less in number. On an average, there are 2.05 constraints per benchmark. The model count for all 17,559 small test cases are plotted in Figure 5. The total execution time is 1 hour 9 minutes with an average of 0.235 seconds per benchmark (median 0.327 seconds).

Big test cases. We find that the model count in the big test cases (average 187 constraints per benchmark), varies from 0 (unsat cases) to all possible strings (unconstrained cases). Figure 6 presents a plot of minimum and maximum model count for each big test case computed by SMC. Because of the $\wedge$ inference rule, SMC often returns the lower bound of 0 to maintain soundness. The total execution time for 1,342 benchmarks is 1 hour 58 minutes 20 seconds with average of 5.291 seconds per benchmark (median 6.190 seconds).

Precision. SMC computes the exact number of solutions (i.e. $\epsilon = 0$ for precision) for 21.1% and 94% of big and small test cases respectively. As discussed before, a big test case may have hundreds of constraints. Thus they are likely to contain a constraint that SMC handles imprecisely, hence giving a lower bound as 0. The gap between minimum and maximum model count is significant in 70% of big test cases with the average precision of $\epsilon = 0.428$. On the other hand, the precision of small test cases is better (the average precision is $\epsilon = 0.01$) as they are less likely to involve constraints that SMC handles imprecisely.

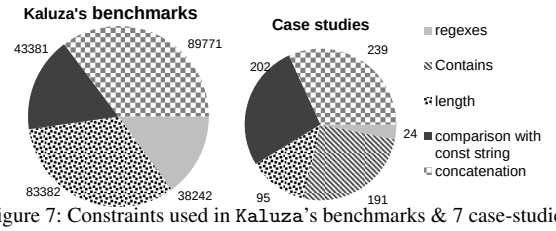

Figure 7: Constraints used in `Kaluza`'s benchmarks & 7 case-studies

4.5 Usage of Constraints

We report the composition of constraint types in Figure 7 for `Kaluza`'s benchmarks, six string programs (4 UNIX utilities and `Null HTTPd`, `Ghttpd`) and the password case study. Since Kaluza's language does not include `strstr` or `contains`, only regex, concat, comparison with const string and length operators are used. The `strstr` operator is not present in the constraints of `obscure` example because we translate `strcasestr` to `contains` constraint directly. The 6 small utilities use all the string operations except regex. For studying password strength meter, we use 24 regular expression constraints in total for three websites. In our experiments, all the supported constraints for strings are thus used multiple times. The most commonly used constraint is concatenation (89,771 times) whereas regexes are used only 24 times.

5. Related Work

Abraham de Moivre introduced the concept of generating functions in 1730 to solve the general linear recurrence problem [25]. Since then, it has been widely used in several fields. In computer science, GFs are use in combinatorial problems [41], analysis of algorithms [42], probabilistic graphical models [32], etc. SMC uses it for addressing the model counting problem.

Today, although several model counting tools are available, none of them supports string operations directly. We briefly discuss the early work on model counting in both boolean and non-boolean (integer) domains [35]. In the boolean domain, Birnbaum *et al*. present an algorithm for counting models of propositional formulas. They extend the Davis-Putnam procedure which checks the validity of a first-order logic formula [21]. In non-Boolean domains, Barvinok's algorithm uses Integer Linear Programming (ILP) to count integer models [16]. LattE [8] implements the enhanced version of Barvinok's algorithm [48]. RelSat solves instances of propositional SAT using constraint satisfaction problem (CSP) look-back techniques [18]. The extended tool determines the exact number of solutions [12]. But it uses approximate model counting (potentially unsound) for propositional formulas. JPF-QIF uses it to compute upper bound on QIF for Java programs [37].

Apart from tools discussed in Section 1, Backes *et al*. use LattE based integer model counting in DisQuant tool to quantify information leakage [15]. Klebanov count the size of the equivalence classes for QIF using LattE [30], and show that model counting is the main bottleneck. Both do not reason about strings and also suffer from the path-explosion problem.

FuzzBALL dynamically collects the symbolic path constraints over bitvectors (which represent strings) and uses strategies such as sampling, enumeration and probabilistic model counting to calculate the output influence [5]. Newsome *et al*. [36] use these techniques for false positive elimination in taint-tracking. However the tool is not always sound, especially when it uses the sampling to es-

timate the model count. QUAIL uses Markovian models to measure information leakage and is designed to support imperative code constructs [20]. As compared to QUAIL, SMC has much better scalability and efficiency for handling string data types.

6. Conclusion

We present SMC, an automatic tool for model counting over an expressive string constraint language. SMC is practical, precise and is able to handle constraints from real-world programs. Key to its success is a combinatorial analysis technique based on generating functions that can be used in several quantitative analysis applications on structured datatypes in the future.

7. Acknowledgements

We thank the anonymous reviewers of this paper for their helpful feedback, and our shepherd Madhusudan Parthasarathy for his insightful comments and suggestions for preparing the final version of the paper. We thank Stephen McCamant, Chin Wei Ngan, Asankhaya Sharma, Ratul Saha and Adi Yoga Sidi Prabawa for their comments on an early presentation of this work. This work is supported by the Ministry of Education, Singapore under Grant No. R-252-000-495-133 and the National Science Foundation under grant CNS-1228995.

References

[1] BusyBox. `http://www.busybox.net/`.

[2] CoreUtils. `http://www.gnu.org/software/coreutils`.

[3] Drupal Password Strength Meter. `https://drupal.org`.

[4] eBay Password Strength Meter. `https://ebay.com`.

[5] Fuzzball. `http://bitblaze.cs.berkeley.edu/fuzzball.html`.

[6] Ghttpd Vulnerability. `http://www.securityfocus.com/bid/5960`.

[7] John the Ripper. `http://www.openwall.com/john/`.

[8] LattE Tool. `http://www.math.ucdavis.edu/~latte/`.

[9] MathLink API. `http://reference.wolfram.com/mathematica/guide/MathLinkAPI.html`.

[10] Microsoft Password Strength Meter. `https://www.microsoft.com/security/pc-security/password-checker.aspx`.

[11] Null HTTPd Vulnerability. `http://www.securityfocus.com/bid/5774`.

[12] RelSat Tool. `http://code.google.com/p/relsat/`.

[13] SMC Tool Online. `https://github.com/loiluu/smc`.

[14] F. Bacchus, S. Dalmao, and T. Pitassi. Solving #SAT and Bayesian Inference with Backtracking Search. *Journal of Artificial Intelligence Research*, 2009.

[15] M. Backes, B. Kopf, and A. Rybalchenko. Automatic Discovery and Quantification of Information Leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[16] A. I. Barvinok. A Polynomial Time Algorithm for Counting Integral Points in Polyhedra When the Dimension Is Fixed. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, 1993.

[17] R. J. Bayardo and J. D. Pehoushek. Counting Models using Connected Components. In *Proceedings of the AAAI National Conference*, 2000.

[18] R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-back Techniques to Solve Real-world SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, 1997.

[19] K. Beyls and E. H. D'Hollander. Generating Cache Hints for Improved Program Efficiency. *Journal of Systems Architure*, 2004.

[20] F. Biondi, A. Legay, L.-M. Traonouez, and A. Wasowski. QUAIL: A quantitative security analyzer for imperative code. In *Proceedings of the 25th International Conference on Computer Aided Verification*, 2013.

[21] E. Birnbaum and E. L. Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 1999.

[22] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic. Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[23] M. Castro, M. Costa, and J.-P. Martin. Better Bug Reporting with Better Privacy. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[24] X. de Carné de Carnavalet and M. Mannan. From Very Weak to Very Strong: Analyzing Password-Strength Meters. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[25] A. de Moivre. *Miscellanea Analytica*. 1730.

[26] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[27] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[28] P. Hooimeijer and W. Weimer. A Decision Procedure for Subset Constraints over Regular Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[29] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for string constraints. In *Proceedings of the Eighteenth International Symposium on Software testing and Analysis*, 2009.

[30] V. Klebanov. Precise Quantitative Information Flow Analysis Using Symbolic Model Counting. In *Proceedings of the International Workshop on Quantitative Aspects in Security Assurance*, 2012.

[31] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[32] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press, 2009.

[33] B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proceedings of Worst-Case Execution Time (WCET) Analysis Workshop*, 2003.

[34] S. McCamant and M. D. Ernst. Quantitative Information Flow as Network Flow Capacity. In *Proceedings of the ACM SIGPLAN Conference on Programming language Design and Implementation*, 2008.

[35] A. Morgado, P. J. Matos, V. M. Manquinho, and J. P. M. Silva. Counting Models in Integer Domains. In *SAT*, 2006.

[36] J. Newsome, S. McCamant, and D. Song. Measuring Channel Capacity to Distinguish Undue Influence. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2009.

[37] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic Quantitative Information Flow. *SIGSOFT Software Engineering Notes*, Nov. 2012.

[38] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[39] D. Roth. On the Hardness of Approximate Reasoning. *Journal Artificial Intelligence*, 1996.

[40] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[41] R. Sedgewick and P. Flajolet. *Analytic Combinatorics*. Cambridge University Press, 2009.

[42] R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Addison-Wesley Professional, 2013.

[43] C. E. Shannon. A Mathematical Theory of Communication. *Bell System Technical Journal*, 1948.

[44] G. Smith. On the Foundations of Quantitative Information Flow. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures*, 2009.

[45] B. Taylor. *Linear Perspective*. 1715.

[46] S. Tople, S. Shinde, Z. Chen, and P. Saxena. AutoCrypt: Enabling homomorphic computation on servers to protect sensitive web content. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[47] A. Turjan, B. Kienhuis, and E. Deprettere. A Compile Time Based Approach for Solving Out-of-Order Communication in Kahn Process Networks. In *Proceedings of The IEEE International Conference onApplication-Specific Systems, Architectures and Processors*, 2002.

[48] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 2007.

[49] M. Weir, S. Aggarwal, B. de Medeiros, and B. Glodek. Password cracking using probabilistic context-free grammars. *2012 IEEE Symposium on Security and Privacy*, 2009.

[50] Wolfram Research. Mathematica. `http://www.wolfram.com/mathematica`, 2013.