Constraint Normalization and Parameterized Caching for Quantitative Program Analysis

Anonymous Author(s)

ABSTRACT

Symbolic program analysis techniques rely on satisfiability-checking constraint solvers, while quantitative program analysis techniques rely on model-counting constraint solvers. Hence, the efficiency of satisfiability checking and model counting is crucial for efficiency of modern program analysis techniques. In this paper, we present an extensible group-theoretic constraint normalization framework that reduces constraints to a normal form to support constraint caching. Our normalization framework includes reductions that preserve the cardinality of the solution set of a constraint, but not its solution set, for model counting queries. We present constraint normalization techniques for string constraints in order to support analysis of string manipulating code. We also present a parameterized caching approach where, in addition to storing the result of a model-counting query, we also store a counting object in the constraint store that allows us to efficiently recount the number of satisfying models for different maximum bounds. We implement our caching framework in our tool Cashew and integrate it with the symbolic execution tool Symbolic PathFinder (SPF) and the model-counting constraint solver ABC. Our experiments show that constraint caching can significantly improve the performance of symbolic and quantitative program analyses. For instance, Cashew can normalize the 10,104 unique constraints in the SMC/Kaluza benchmark down to 394 normal forms, achieve a 10x speedup on the SMC/Kaluza-Big dataset, and an average 3x speedup in our SPF-based side-channel analysis experiments.

ACM Reference format:
Anonymous Author(s). 2017. Constraint Normalization and Parameterized Caching for Quantitative Program Analysis. In *Proceedings of 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, 4–8 September, 2017 (ESEC/FSE 2017),* 12 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The developments in the area of Satisfiability-Modulo-Theories (SMT) [8, 10] and the implementation of powerful SMT solvers [9, 17, 18] have been the key technological developments that enabled the rise of effective symbolic program analysis and testing techniques in the last decade [12, 23, 28, 45]. However, performing symbolic analysis via satisfiability checking is not sufficient for *quantitative program analysis*, which is an important problem that arises in many contexts such as probabilistic analysis [11, 21, 35], reliability analysis [19] and quantitative information flow [5, 14, 24, 37, 38, 40–42, 46, 49].

The enabling technology for quantitative program analysis is model-counting constraint solvers. A model-counting constraint solver returns the number of solutions for a given constraint within a given bound [4, 6, 36].

Since constraint solving and model counting are heavily used in program analysis, improving performance of these tasks is of critical importance. In this paper, we present a new approach for constraint normalization and constraint caching with the goal of improving the performance of quantitative program analyses.

The key step in constraint caching is normalization of constraints, i.e., reducing constraints to a normal form, where if two constraints are equivalent (w.r.t. satisfiability or model counting), then they are reduced to the same normal form. Using the normal form of a constraint as a key, we can recover results of previous satisfiability or model-counting queries without recomputing them.

Earlier techniques for constraint caching [3, 27, 51] 1) focus only on numeric constraints and do not handle string constraints, 2) use normalization techniques that preserve the exact solution set of a constraint, which reduces cache hits for model-counting queries, and 3) always produce cache misses for model-counting queries if a different bound is used, even if the queried constraint remains the same. In this paper, we extend earlier results in multiple directions:

- We present constraint normalization techniques for model counting where the normalized form of a constraint may not preserve the solution set for the constraint but preserves the cardinality of the solution set.
- We extend constraint caching to string constraints which is crucial for analyzing string manipulating code.
- We present a parameterized caching approach where, in addition to the result of a model-counting query, we also cache a counting object in the constraint cache that allows us to efficiently recount the models for different bounds.
- We present an extensible group-theoretic framework for constraint normalization that generalizes earlier results on constraint normalization for caching.

We implemented our techniques in our tool Cashew, which we integrated with Symbolic PathFinder (SPF) [39] and a model-counting constraint solver ABC [4]. Our experiments demonstrate that constraint caching can improve the performance of quantitative program analysis significantly.

The rest of the paper is organized as follows: In Section 2 we provide some motivating examples for constraint caching. In Section 3 we give an overview of our constraint caching framework. In Section 4 we discuss our group-theoretic normalization scheme. In Section 5 we describe the constraint language we support. In Sections 6 and 7, we present the constraint normalization procedure.

ESEC/FSE 2017, Paderborn, Germany

DOI: 10.1145/nnnnnnn.nnnnnnn

In Section 8, we present our experiments. In Section 9, we discuss related work. In Section 10, we conclude the paper.

2 MOTIVATION

The amount of string-manipulating code in modern software applications has been increasing. Common uses of string manipulation include: 1) Input sanitization and validation in web applications; 2) Query generation for back-end databases; 3) Generation of data formats such as XML and HTML; 4) Dynamic code generation; 5) Dynamic class loading and method invocation. In order to analyze programs that use string manipulation, it is necessary to develop techniques for efficient manipulation of string constraints. Recently, there has been significant amount of work in string constraint solving to address this problem [2, 20, 25, 26, 29, 31–33, 44, 48, 53]. One of our contributions in this paper is a constraint normalization and caching framework that can handle string constraints.

Consider the following string constraint F:

$$b = \text{“}https\text{”} \wedge \mathsf{prefix\_of}(b, url) \wedge c = \text{“?”} \wedge \mathsf{contains}(c, url) \\ \wedge\, w \in (0|1)^+ \wedge \mathsf{index\_of}(w, url) = 5$$

F is a conjunction of constraints on string variables b, c, w, and url. In addition to string equality and regular expression membership, string constraints such as prefix_of and contains and string functions such as index_of are commonly used in string manipulating code. The solution set of F is the set of string values that can be assigned to variables b, c, w, and url for which F evaluates to true.

Constraints such as F commonly arise in symbolic program analysis. For example, F can correspond to a path constraint generated during symbolic execution of a string-manipulating program. A fundamental question about a constraint F generated during program analysis is its satisfiability. Symbolic program analysis techniques generate numerous satisfiability queries while analyzing programs. Given that satisfiability checking is computationally expensive, it is crucial to answer satisfiability queries efficiently in order to build scalable symbolic program analysis tools.

On the other hand, quantitative program analysis techniques ask another type of question while analyzing programs. Assume that we bound the length of the string variables b, c, w, and url in constraint F to 5. How many different string values are there for the variable b such that the constraint F is satisfiable within the given bound? This type of queries can be answered by model-counting constraint solvers. Again, due to the high complexity of model counting, answering model counting queries efficiently is crucial for quantitative program analysis.

Now, consider the following string constraint G:

$$k = \text{“\#”} \wedge w = \text{“}http:\text{”} \wedge \mathsf{contains}(k, var0) \wedge z \in (1|0)^+ \\ \wedge\, \mathsf{index\_of}(z, var0) = 5 \wedge \mathsf{prefix\_of}(w, var0)$$

Constraint G is a constraint on string variables k, z, w, and $var0$. Assume that constraints F and G are generated during program analysis and it is necessary to check the number of satisfying solutions and satisfiability of each. Can we avoid making redundant calls to the constraint solver? Note that the solution sets of F and G are different since different string constants appear in these two constraints. However, the satisfiability and the cardinality of the solution sets for these two constraints are identical. Hence, if we were able to detect the relationship between the number of satisfying models of F and G and had stored the result of a model-counting query on F, then when we see G we do not have to call the model-counting constraint solver again. The same holds for satisfiability queries.

The question now becomes: Can we find a fast way to determine that F and G are equivalent with respect to satisfiability and model counting so that we can prevent redundant calls to the constraint solver? In this paper, we present a constraint normalization scheme to determine this type of equivalence. Based on our normalization scheme, the normalized form of F and G are identical:

$$v0 = \text{“}a\text{”} \wedge v1 = \text{“}bccde\text{”} \wedge \mathsf{contains}(v0, v2) \\ \wedge\, \mathsf{prefix\_of}(v1, v2) \wedge v3 \in (f|g)^+ \wedge \mathsf{index\_of}(v3, v2 = 5)$$

Hence, given two constraints, in order to determine their equivalence, we first normalize the constraints and check if their normalized forms are the same. Using a constraint store to cache the results of prior queries to the solver, we avoid redundant queries for constraints that have the same normalized form.

For both satisfiability and model-counting queries, we can cache the result of the query in a constraint store, use normalization to determine equivalence of constraints, and then reuse the query results from the store when we get a cache hit. However, since model-counting queries come with a bound parameter, in order for the query to match, the bound also has to match. If the bound does not match, can we still reuse a model counting query result? Parameterized model-counting techniques [4, 36] do not only count the number of solutions for a constraint within a given bound, but they also generate a model counter that can count the number of solutions for any given bound. Note that counting the number of solutions with different bounds may be necessary during program analysis. For example, consider the following constraint:

$$\mathsf{contains}(x, \text{“}abcde\text{”}) \wedge |y| > |x| \wedge y \in (ab)^*$$

This constraint has no solutions for bounds less than 5 but has satisfying solutions for higher bounds.

In this paper, we present a parameterized caching approach that utilizes parameterized model-counting constraint solvers. We assume that, in response to a model-counting query, parameterized model-counting constraint solvers return a model-counting object that can be used to count the number of models for any given bound. By storing the model-counting object in the constraint cache store, we are able to reuse model counting query results even for queries with different bounds.

3 CONSTRAINT CACHING

Our tool Cashew, depicted in Figure 1, is designed to work with a wide range of model-counting solvers to support quantitative program analyses. Algorithm 1 outlines how Cashew handles model-counting queries. Cashew expects a query of the form (F, V, b), where F is a well-formed formula, V is a set of variables in F, and b is a bound. The answer to the query, denoted as $\#(F, V, b)$, is the number of satisfying solutions for F for the variables in V within the bound b. We normalize the formula, variable(s) and bound using our normalization procedure, Normalize-Query, which is described in the following sections. The resulting normalized query is denoted as $[\![F, V, b]\!] = \text{Normalize-Query}(F, V, b)$.

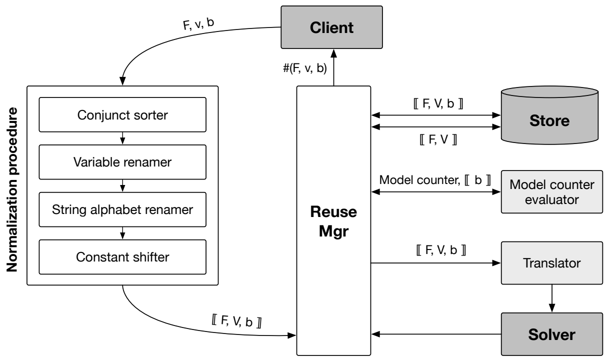

Figure 1: Architecture of Cashew

Algorithm 1 Constraint-Caching(F, V, b):

Input: A query (F, V, b).
Output: The number of satisfying solutions of V in F under the length bound b.

```
⟦F, V, b⟧ = Normalize-Query(F, V, b)
if Hit on ⟦F, V, b⟧ then
    return #(F,V,b)
end if
if Hit on ⟦F, V⟧ then
    Evaluate the model counter for bound ⟦b⟧ using the model-counter evaluator;
    Store the result under ⟦F, V, b⟧
    return #(F,v,b)
end if
Translate ⟦F, V, b⟧ and send it to the selected model-counting solver
Store the returned model counter under ⟦F, V⟧
Store #(F,V,b) under ⟦F, V, b⟧
return #(F,V,b)
```

Depending on the capabilities of the selected model-counting constraint solver, $[\![F, V, b]\!]$ is queried differently. Algorithm 1 outlines the normalization and query process. Typical model-counting constraint solvers [13, 34, 47], return a single count value ($\#(F, v, b)$) after receiving a query of the form (F, V, b). For such constraint solvers, our caching algorithm first sends the query $[\![F, V, b]\!]$ to the cache store. If there is a cache hit, the result is returned to the client. If not, the normalized query is sent to the model-counting solver, and the result is stored under $[\![F, V, b]\!]$ and returned to the client.

We call a model-counting constraint solver *paramaterized* if it returns a model-counter object that can be used to compute the number of satisfying solutions for an arbitrary bound. ABC [4] is a parameterized model-counting constraint solver where the model-counter object is the transfer matrix of an automaton that accepts all satisfying models of the given constraint. SMC [36] and barvinok [50] are also parameterized model-counting constraint solvers where the model-counter object is a generating function.

For parameterized solvers, the store is queried as follows: First, $[\![F, V, b]\!]$ is queried. In the case of a hit, the result ($\#(F, V, b)$) is returned to the client. In the case of a miss, an additional query for $[\![F, V]\!]$ is made. If this results in a hit, the model counter for $[\![F, V]\!]$ is recovered from the store. This model counter is sent to the model-counter evaluator which evaluates $\#(F, V, b)$ based on $[\![b]\!]$. The result returned by the model-counter evaluator is stored under $[\![F, V, b]\!]$ and is returned to the client. If both queries are misses, a call to the selected solver is made, the model counter is computed and cached under the key $[\![F, V]\!]$, and $\#(F, V, b)$ is evaluated based on $[\![b]\!]$, stored under $[\![F, V, b]\!]$ and returned to the client.

In order to use Cashew's parameterized caching functionality and reuse cached model counters, a service that is able to take a model counter (such as a transfer matrix or a generating function) and evaluate it for a particular bound is required. This service is referred to as the model-counter evaluator.

4 GROUP-THEORETIC FRAMEWORK

The goal of normalization is to reduce constraints equivalent under some property to the same form. This objective is shared by work in constaint programming, where detecting symmetries in constraints leads to a more efficient search [15, 22]. Symmetry-breaking for constraint programming is based on concepts from group theory [16] and we adopt this formulation to describe our normalization scheme for caching.

Our framework provides a means for constructing normal forms of constraints based on groups of property-preserving transformations. For different analysis problems, it might be necessary to preserve the entire solution set, the cardinality of the solution set, or only the satisfiability of constraints, each corresponding to a different level of normalization. Our framework is equally applicable, regardless of the desired level of normalization. Our framework is also not restricted to a constraint language, but is equally applicable to any background theory on which a group of property-preserving transformations can be defined.

Symmetry Groups. A group $(\mathcal{G}, \mathrm{op})$ is a set of elements together with a binary operator that satisfies the four group axioms: closure, associativity, identity, and invertibility. For example, the set of all transpostions on the natural numbers, $\mathbb{N}$, under the binary operator function composition form a group. The transposition from $\mathbb{N}$ to itself defined by the relation $\{(1, 2), (2, 1)\}$ is an example of an element of this group which maps 1 to 2, 2 to 1 and all other elements of $\mathbb{N}$ to themselves.

A subset of a group is called a *subgroup* if it also forms a group under the same binary operator. We construct the group of cardinality-preserving transformations under composition $(\mathcal{G}_{\text{card}}, \circ)$ by introducing its generating subgroups. As composition is the only binary operator we consider, we simply refer to this group as $\mathcal{G}_{\text{card}}$ throughout the remainder of the paper.

Solution-Set-Preserving Subgroups of $\mathcal{G}_{card}$. A solution-set-preserving transformation is one under which the solution set is mapped to itself. Any solution-set-preserving transformation is trivially cardinality-preserving. Our first solution-set-preserving subgroup of $\mathcal{G}_{\text{card}}$ is the group mentioned above—that of all transpositions on $\mathbb{N}$. This group is identical to the permutation group on $\mathbb{N}$ whose elements fix all but a finite number of numbers. Because the indices of conjuncts within a constraint are natural numbers, the above subgroup can be said to act on the domain of indices $\mathcal{I}$.

Intuitively, this subgroup captures our understanding that the solution set of a constraint is independent of the order of the conjuncts in it. Under the transposition $\{(1, 2), (2, 1)\}$ for example, the formula $x > 0 \land y < 0$ is mapped to $y < 0 \land x > 0$, making the two orderings equivalent modulo the action of this group.

Our second solution-set-preserving subgroup is given by the transposition group on the infinite domain of possible variable names, $\mathbb{V}$. Since the solution set of a constraint is independent of the choice of variable names, two constraints that are equivalent modulo the action of this group have the same solution set. As a simple example, realize that the number of solutions for $x < 7 \wedge x > 2$ is the same as that for $w < 7 \wedge w > 2$. This follows from the transposition given by the relation $\{(x, w), (w, x)\}$ being an element of this transposition group.

Cardinality-Preserving Subgroups of $\mathcal{G}_{card}$. Preserving only the *cardinality* of the solution set of a constraint allows for the use of subgroups with more flexible group actions. Under these groups, the solution set of a constraint is bijectively mapped to the solution set of another constraint. The number of solutions remains unchanged under this bijection though the solution set itself is transformed.

Our first family of cardinality-preserving subgroups are given by the Euclidean groups $E(n)$ (symmetry groups on Euclidean space) acting on the solution space of linear integer arithmetic constraints. The elements of these groups are Euclidean motions including rigid motions such as translations and rotations as well as indirect isometries such as reflection. Under these symmetries of Euclidean space, the volume captured by the correspdoning polytope remains unchanged.

Though this volume is preserved under any action of the Euclidean group, the number of lattice points in the polytope can change under certain group actions. Because we are often interested in the number of integer solutions to a constraint, we limit ourselves to considering only those tranformations that preserve the number of lattice points as well as those that can be easily reflected through changes in the syntax of the constraint. In particular, our normalization scheme uses the subgroup of integral translations in Euclidean space as a generating subgroup for $\mathcal{G}_{\text{card}}$. Integral translations can be reflected syntacically in integer constraints through changes in the constant terms of each conjunct. Each constant term must be identically shifted by an integral amount. For example, shifting each constant term of the constraint $y = 2 \wedge x \geq 0 \wedge y \geq 0$ by 2 results in the constraint $x + y = 4 \wedge x \geq 2 \wedge y \geq 2$ which has the same number of integer solutions (6) as the original.

For any arithmetic constraint, F, we call the vector composed of the constant terms of each of its conjuncts its *shift*, denoted $\boldsymbol{Shift}(F)$ The domain that the subgroup of integral translations acts on is that of all possible shifts, which we denote $\mathcal{SH}$. For string or mixed constraints, we do not apply transformations from this subgroup and we say that $\boldsymbol{Shift}(F)$ of such constraints is $\emptyset$.

Our second cardinality-preserving subgroup is given by the group of permutations on the string alphabet, Σ. The solution set of a string constraint can be canonically represented by an automaton that accepts exactly the set of solutions to that constraint. Transitions between states are made based on a set of allowed alphabet symbols. Permuting the set of alphabet symbols thus changes the strings accepted by that automaton but not the cardinality of the accepted set. As a simple example, observe that the number of solutions for the constraints $F := x.\text{contains}(``ac")$ and $F' := x.\text{contains}(``bd")$ are the same.

Orbits under the Symmetry Group $\mathcal{G}_{card}$. These subgroups generate $\mathcal{G}_{\text{card}}$ in the following sense: the domain of any element of $\mathcal{G}_{\text{card}}$ is the union of the domains of the subgroups, making it the Cartesian product $\mathcal{I} \times \mathbb{V} \times \mathcal{SH} \times \Sigma$. Every element of a subgroup acts as an element of $\mathcal{G}_{\text{card}}$ by acting as the identity on every domain element on which it is not defined. Any element of $\mathcal{G}_{\text{card}}$ can be written as a composition of elements from these subgroups.

For a constraint F, the **orbit** of F under $\mathcal{G}_{\text{card}}$ is the set of constraints obtained by applying any element $\sigma \in \mathcal{G}_{\text{card}}$ to F.

The problem of choosing a normalized form for F can now be formulated as choosing a representative constraint from the orbit of F under $\mathcal{G}_{\text{card}}$. We do this by defining a strict ordering on constraints and choosing the well-defined lowest ordered constraint within the orbit as the representative for all constraints within the orbit.

While we have spoken generally about cardinality-preserving group actions, our application of interest is in *parameterized* model-counting which involves finding the number of satisfying solutions to a constraint for any given bound. While most of the group actions defined above preserve the number of solutions for a given bound, the elements of the Euclidean group may not. For example, $y = 2 \wedge x \geq 0 \wedge y \geq 0$ has 6 solutions given a bound of 2 but $x + y = 4 \wedge x \geq 2 \wedge y \geq 2$, which is in the same orbit under $\mathcal{G}_{\text{card}}$, has only one solution for the same bound. In order to preserve the paramaterized model count, the bound is translated according to the same group action as the constraint. In the example above, bound 2 is translated to bound 4 by the same integer translation (2) that translated the shift, resulting in 6 satisfying models.

$$
\begin{aligned}
\mathcal{T}_\mathsf{S} &:= \mathsf{c} \mid \mathsf{v_S} \mid \mathcal{T}_\mathsf{S} \cdot \mathcal{T}_\mathsf{S} \mid \mathsf{char\_at}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{A}) \mid \mathsf{int\_to\_str}(\mathcal{T}_\mathsf{A}) \mid \\
&\qquad \mathsf{replace}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{substr}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{A}, \mathcal{T}_\mathsf{A}) \\
\mathcal{T}_\mathsf{R} &:= \epsilon \mid \mathsf{s} \mid (\mathcal{T}_\mathsf{R}) \mid \mathcal{T}_\mathsf{R} \cdot \mathcal{T}_\mathsf{R} \mid \mathcal{T}_\mathsf{R} | \mathcal{T}_\mathsf{R} \mid \mathcal{T}_\mathsf{R}^* \\
\mathcal{T}_\mathsf{A} &:= \mathsf{n} \mid \mathsf{v_A} \mid -\mathcal{T}_\mathsf{A} \mid (\mathcal{T}_\mathsf{A}) \mid \mathcal{T}_\mathsf{A} + \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} - \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} \times \mathsf{n} \mid \\
&\qquad \mathsf{n} \times \mathcal{T}_\mathsf{A} \mid |\mathcal{T}_\mathsf{S}| \mid \mathsf{index\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{str\_to\_int}(\mathcal{T}_\mathsf{S})
\end{aligned}
$$

Figure 2: Here $\mathsf{c} \in \Sigma$, $\mathsf{n} \in \mathbb{Z}$, $\mathsf{s} \in \Sigma^*$, $\mathsf{v_S}$ and $\mathsf{v_A}$ denote an unbounded string variable and an integer variable, resp.

$$
\begin{aligned}
\mathsf{L} &:= \top \mid \bot \mid \mathsf{S} \mid \mathsf{R} \mid \mathsf{A} \\
\mathsf{S} &:= \mathcal{T}_\mathsf{S} = \mathcal{T}_\mathsf{S} \mid \mathcal{T}_\mathsf{S} \neq \mathcal{T}_\mathsf{S} \mid \mathsf{contains}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{prefix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \\
&\quad \mathsf{suffix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{not\_contains}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \\
&\quad \mathsf{not\_prefix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \mid \mathsf{not\_suffix\_of}(\mathcal{T}_\mathsf{S}, \mathcal{T}_\mathsf{S}) \\
\mathsf{R} &:= \mathcal{T}_\mathsf{S} \in \mathcal{T}_\mathsf{R} \mid \mathcal{T}_\mathsf{S} \notin \mathcal{T}_\mathsf{R} \\
\mathsf{A} &:= \mathcal{T}_\mathsf{A} = \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} < \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} \leq \mathcal{T}_\mathsf{A} \mid \mathcal{T}_\mathsf{A} \neq \mathcal{T}_\mathsf{A}
\end{aligned}
$$

Figure 3: The language L: conjuncts of string (S), regular expression (R) and linear integer arithmetic (A) types.

5 CONSTRAINT LANGUAGE

We focus on constraints over *strings* and *linear integer arithmetic.*

We define three types of terms: string terms $\mathcal{T}_\mathsf{S}$, regular expression terms $\mathcal{T}_\mathsf{R}$, and linear integer arithmetic terms $\mathcal{T}_\mathsf{A}$, as described in Figure 2. We consider three types of constraints over these terms, which we call *conjuncts* throughout this paper: string conjuncts S, regular membership conjuncts R, and LIA conjuncts A. Let L be a language defined over these conjuncts, formalized in Figure 3. Input constraints to our normalization procedure are assumed to be in conjunctive form, where each conjunct is from L.

The set of string operators S_{op} is comprised of the operators listed in the definition of $\mathcal{T}_\mathsf{S}$ and the set of the string comparators S_{comp} by the comparator listed in S; the set of regular expression operators R_{op} by those in $\mathcal{T}_\mathsf{R}$ and the set of regular expression comparators R_{comp} by those listed in R; the set of the linear integer arithmetic operators A_{op} by those in $\mathcal{T}_\mathsf{A}$ and the set of comparators A_{comp} by those listed in A. By $\textsc{Type}(\_)$ we denote a function that takes in a conjunct and returns the comparator of this conjunct. For every term $t \in \mathcal{T}_\mathsf{S} \cup \mathcal{T}_\mathsf{R} \cup \mathcal{T}_\mathsf{A}$, and conjunct $C \in \mathsf{L}$, we define a *length function* denoted as $\parallel t \parallel$, respectively $\parallel C \parallel$, as a total number of variables, constant symbols, and operators in t, respectively in C. The expression $\#Var(t)$ returns the number of variables in t.

6 CONSTRAINT ORDERING

Assume a strict total ordering on constraints, $\prec$. A constraint F is a normal form if for every other constraint F' in its orbit under $\mathcal{G}_{\text{card}}$, $F \prec F'$. There are many ways to impose an ordering on constraints. We present one possible ordering below.

Our ordering produced compositionally, with strict orders defined over various components of our language which are composed to yield an ordering on constraints. To start, we define an ordering on each element of the domain of $\mathcal{G}_{\text{card}}$.

The ordering on both $\mathbb{V}$ and Σ is lexicographical. The ordering on $\mathcal{I}$ is that induced by the natural numbers. We define the ordering on $\mathcal{SH}$, the domain of constant shifts, after we introduce an ordering on vectors. We consider vectors over strict totally ordered sets and denote by $\prec_{vec}$ an order on such vectors.

Let X be a strict totally ordered set, and $\prec_X$ be a strict total order on X. Let $\boldsymbol{v} = (v_0, \ldots, v_n)$ and $\boldsymbol{u} = (u_0, \ldots, u_m)$ be two vectors over X, then $\prec_{vec}$ is defined as:

$$\boldsymbol{v} \prec_{vec} \boldsymbol{u} \iff \begin{cases} m < n, \ or \\ m = n, \ \exists i \ \forall j : \ j < i < n, \ v_j = u_j, \ v_i \prec_X u_i. \end{cases}$$

This defines ordering on $\mathcal{SH}$ since shift vectors are built over integer constants.

Our normalization procedure relies on the following auxiliary functions that given a constraint return, as vectors, various structural and syntactic components characterizing the constraint. These vectors are built over the domains of $\mathbb{V}$, Σ and $\mathbb{Z}$, i.e. over strict totally ordered sets.

- $\boldsymbol{VI}(F)$ — returns *a vector of the indices of variables* as they occur in F relative to other variables, constants and operators. The indices are compared according to the "$<$" operator over $\mathbb{Z}$.
- $\boldsymbol{Int}(F)$ — returns *a vector of integer constants* occuring in F from left to right, ignoring all elements of $\boldsymbol{Shift}(F)$. The vectors are compared according to the "$<$" operator on $\mathbb{Z}$.
- $\mathbb{V}(F)$ — returns *a vector of variable names* occuring in F from left to right. These vectors are compared according to the lexicographical order on $\mathbb{V}$.
- $\boldsymbol{\Sigma}(F)$ — returns *a vector of string characters* occuring in F from left to right. The vectors are compared according to the lexicographical order on Σ.

Next, we define strict total orderings on operators and (separately on) comparators, listing them in the order of the increasing precedence. Both, operators and comperators, are ordered with precedence to S, then R, and A.

- S_{op}: $\cdot$, the rest of the string operators in the lexicographic order according to their names in Figure 2;
- R_{op}: ordered according to the standard precedence order on regular expression operators;
- A_{op}: $+, -, \times, ||, ()$, the rest of the LIA operators in the lexicographic order according to their names in Figure 2.
- S_{comp}: $=, \neq$, the rest of the string comperators in the lexicographic order according to their names in Figure 3;
- R_{comp}: $\in, \notin$;
- A_{comp}: $=, <, \leq, \neq$;

The ordering on comparators allows to define an order $\prec_{type}$ on the types of the conjuncts $\textsc{Type}$, based on the type of the comparator occuring in the conjuncts. The strict total ordering on operators allows to introduce *vectors of operators* of constraints and compare them with $\prec_{vec}$:

- $\boldsymbol{Op}(F)$ — *a vector of string, regular and LIA operators* occuring in F from left to right.

Note, all auxiliary vectors, and their orderings, introduced in this section are defined for constraints and are naturally applicable to conjuncts – as to a special type of constraints with a single conjunct.

In the future, when we compare two elements of the same type we will drop the subscript notation and use $\prec$ to represent comparison between them.

We are now ready to build a strict total order on conjuncts. We define the ordering hierarchically: the structural or syntactic aspects of the conjuncts are compared one at a time in a fixed order, until a tie-breaking aspect is found. This order can be selected in any way. We present one intuitive order below to distinguish conjuncts with more significant differences as early as possible. The conjuncts are first compared based on their type $\textsc{Type}$, then based on their length $\parallel \parallel$, then the total number of variables $\#Var$, then their vectors of operators $\boldsymbol{Op}$, followed by the vectors of indices of variables $\boldsymbol{VI}$, their vectors of integer coefficients $\boldsymbol{Int}$, their vectors of variable names $\mathbb{V}$, then vectors of string constants $\boldsymbol{\Sigma}$, and finally based on their constant shifts $\boldsymbol{Shift}$. This order is described in Algorithm 2.

Algorithm 2 C-LessThan(C_1, C_2): Conjunct Comparison

```
Input: Two conjuncts C1, C2 ∈ L
Output: True if C1 ≺ C2, otherwise False
for each f ∈ [Type, || ||, #Var, Op, VI, Int, V, Σ, Shift] do
    if f(C1) ≺ f(C2) then
        return True
    end if
end for
return False
```

This order is strict and total. Two conjuncts are equal if and only if they are the same conjunct. This allows us to extand the ordering to constraints as follows:

(i) Order constraints based on their total number of conjuncts.
(ii) Then, order constraints by comparing their conjuncts element-wise in order of precedence according to the order imposed on $\mathcal{I}$. This is equivalent to comparing conjuncts pairwise from first to last.
(iii) A constraint F comes before a constraint G if the first differing conjunct of F is lower ordered than that of G under C-LessThan.

This order is described in Algorithm 3.

Algorithm 3 F-LessThan(F, G): Constraint Comparison

Input: Two constraints $F = F_1 \wedge \ldots \wedge F_m$ and $G = G_1 \wedge \ldots \wedge G_n$
Output: True if $F \prec G$, otherwise False

```
1: if m = n then
2:     for i ← 1, n do
3:         if C-LessThan(F_i, G_i) then
4:             return True
5:         end if
6:     end for
7: end if
8: return m < n
```

7 CONSTRAINT NORMALIZATION PROCEDURE

The normal form of a constraint F is the lowest constraint in the orbit of F under $\mathcal{G}_{\text{card}}$. In this section, we present a normalization procedure to find the normal form of a constraint.

Given a transformation $\sigma \in \mathcal{G}_{\text{card}}$, we define $\sigma[F]$, the *action of σ on* F, as a composition of elements of four categories corresponding to each of the components of the domain of $\mathcal{G}_{\text{card}}$:

$\mathcal{I}$: $\sigma_{\mathcal{I}}[F]$ gives the constraint resulting from re-ordering the conjuncts of F according to $\sigma_{\mathcal{I}}$.

$\mathbb{V}$: $\sigma_{\mathbb{V}}[F]$ gives the constraint resulting from renaming the variables of F according to $\sigma_{\mathbb{V}}$;

Σ: $\sigma_{\Sigma}[F]$ gives the constraint resulting from permuting the alphabet constants in F according to σ_{Σ};

$\mathcal{SH}$: $\sigma_{\mathcal{SH}}[F]$ gives the constraint resulting from shifting each element of F's shift according to $\sigma_{\mathcal{SH}}$.

We first present an expensive but complete procedure for normalization in Algorithm 4 and give guarantees for its termination and correctness. Given a constraint F, this procedure probes each permutation F' of conjuncts in F, building and applying a composite σ from transformations specific to the domains $\mathbb{V}$, Σ, and $\mathcal{SH}$ which reduces F' until the only transformations that can reduce it further involve an action on $\mathcal{I}$. The results among all permutations of F are compared and the lowest-ordered result is chosen as the normal form of F.

The procedure uses auxiliary functions to build the minimizing domain-specific transformations:

Min-σ-$\mathbb{V}(F')$ constructs $\sigma_{\mathbb{V}}$ compositionally — it proceeds through the conjuncts of F' from left to right renaming the variables of F' in order of appearance. Each time a new variable is encountered a transposition is added to the composition that permutes the name of the encountered variable and the lowest-ordered variable name that no other variable of F' has been renamed to yet. At the start of the procedure, $\sigma_{\mathbb{V}}$ is initialized to the identity transposition on $\mathbb{V}$.

Min-σ-$\Sigma(F')$ similarly constructs σ_{Σ} — it proceeds through the conjuncts of F' from left to right, this time permuting string characters. Each time a new string character is encountered, a transposition is added to the composition that permutes the encountered string character with the lowest-ordered character that no other character in F' has been mapped to yet. σ_{Σ} is initialized as the identity transposition on Σ.

Min-σ-$\mathcal{SH}(F')$ returns $\sigma_{\mathcal{SH}}$ — the tranformation on ***Shift***(F) that translates the constant coefficient of the first appearing (from left to right) linear integer arithmetic conjunct in F' to 0. If F contains variables that are shared between string and LIA constraints $\sigma_{\mathcal{SH}}$ is the identity transformation.

Algorithm 4 Complete-Normalization(F)

Input: A constraint F
Output: The normalized form of F

```
1: F_min := F
2: for each permutation F' of conjuncts in F do
3:     σ_V := Min-σ-V(F')
4:     σ_Σ := Min-σ-Σ(F')
5:     σ_SH := Min-σ-SH(F')
6:     F' := σ_V ∘ σ_Σ ∘ σ_SH[F']
7:     if F-LessThan(F', F_min) then
8:         F_min := F'
9:     end if
10: end for
11: return F_min
```

Theorem 7.1. *Algorithm 4 terminates.*

Proof. Given a constraint F, there are finitely many permutations of conjuncts F'. Consequently, there are finitely many executions of the "for each" loop. Construction of each permutation F' is linear in the length of F. Construction of each of the domain-specific transformations within a single "for each" call is performed in a single pass through the conjuncts of F', thus, is linear in the length of F, too. Computation of the action of the final transformation on F' is also linear in the length of F. Thus, Complete-Normalization terminates. □

Theorem 7.2. *Algorithm 4 returns the normal form of F.*

Proof. Assume G = Complete-Normalization(F) is not the normal form of F. Then either G is not in the orbit of F under $\mathcal{G}_{\text{card}}$ or there is some constraint H in the orbit of F such that $H \neq F$ and $F \not\prec H$. We show that both result in a contradiction.

Assume G is not in the orbit of F. G is the result of permuting the conjucts of F, the action of some $\sigma_{\mathcal{I}}$, composed with domain specific transformations. Each domain-specific transformation has an inverse in $\mathcal{G}_{\text{card}}$ as does any permutation of the conjuncts of F. Therefore, there exists some σ in $\mathcal{G}_{\text{card}}$ such that $\sigma[G] = F$, meaning that G and F are in the same orbit.

Now assume that there is some H in the orbit of F such that $H \neq G$ and $G \not\prec H$. The order of conjuncts in H is given by some transposition of the indices of F. This means that there is some iteration of the for loop of Algorithm 4 in which the conjuncts of the considered permutation of F are ordered identically to those of H. By construction, our choices of $\sigma_{\mathbb{V}}$, σ_{Σ} and $\sigma_{\mathcal{SH}}$ reduce this constraint to the lowest-ordered constraint that maintains the same ordering of conjuncts. Therefore either $G = H$ or $G \prec H$. □

Algorithm 4 gives a normalization procedure which is *sound* (each orbit has at least one fixed point) and *complete* (there is exactly one fixed point for each orbit). In practice, however, such a brute force exploration is very expensive. For our implementation, we use a sound but not complete normalization procedure given in Algorithm 5. Given F, Normalization(F) returns the semi-normal form on F — a constraint within the orbit of F which, though not necessarily the lowest in the orbit, is not higher ordered than F.

Algorithm 5 simplifies Complete-Normalization procedure in that instead of brute-forcing all permutations of conjuncts in F, it inexpensively chooses a permutation by ordering the conjuncts of F according to C-LessThan up to the point when further refinement involves comparison over the domains $\mathbb{V}$, Σ, or $\mathcal{SH}$. In other words, the conjuncts are not compared according to their variable names, string constants or shifts. It is possible that two conjuncts in F are equal by this comparison, in which case their initial order in F is preserved. The resulting permutation of conjuncts defines a transposition on $\mathcal{I}$. We apply this transposition to F, resulting in a constraint F'. $\sigma_{\mathbb{V}}$, σ_{Σ}, and $\sigma_{\mathcal{SH}}$ are generated by the same auxiliary functions as in Algorithm 4, composed, and applied to F'. The result is the semi-normal form of F.

Algorithm 5 Normalization(F)

Input: A constraint F
Output: A semi-normal form of F

$F' :=$ **Permute** conjuncts of F according to Algorithm 2 up until $\mathbb{V}$
$\sigma_{\mathbb{V}} :=$ Min-σ-$\mathbb{V}(F')$
$\sigma_{\Sigma} :=$ Min-σ-$\Sigma(F')$
$\sigma_{\mathcal{SH}} :=$ Min-σ-$\mathcal{SH}(F')$
$[\![F]\!] := \sigma_{\mathbb{V}} \circ \sigma_{\Sigma} \circ \sigma_{\mathcal{SH}}[F']$
return $[\![F]\!]$

Theorem 7.3. *Algorithm 5 is sound.*

Proof. Each action on F is the action of an element of $\mathcal{G}_{\text{card}}$. By definition, the resulting formula is in the orbit of F under $\mathcal{G}_{\text{card}}$. □

The procedure given in Algorithm 5 is not complete. There are orbits for which not every constraint is reduced to the same form. Though this potentially increases the number of misses to the cache, our experimental results demonstrate the large number of formulas mapped to the same semi-normal form by Algorithm 5.

Queries to Cashew are of the form (F, V, b) where V is the set of variables on which to count, and b is the maximum length of a satisfying solution. To ensure that the cardinality of the solution set is preserved after normalizing F, both V and b must be normalized according to the same transformations applied to F during Algorithm 6. Procedure Normalize-Query(F, V, b) implements the query normalization.

Algorithm 6 Normalize-Query(F, V, b)

Input: A query (F, V, b)
Output: A normalized query $[\![F, V, b]\!]$

$[\![F]\!] :=$ Normalization(F)
$\sigma :=$ the transformation used to normalize F
$[\![V]\!] := \sigma[V]$
$[\![b]\!] := \sigma[b]$
return $([\![F]\!], [\![V]\!], [\![b]\!])$

8 EXPERIMENTAL EVALUATION

We implemented our tool, Cashew, as an extension of the Green [51] caching framework. This allows Cashew to use any of the existing Green services, and it allows Green users to benefit from our normalization procedure. We experiment with Cashew-enabled satisfiability and model-counting services, which support string constraints and linear integer arithmetic. They also support *mixed* constraints, i.e., those involving both string and arithmetic operations. In this evaluation, we used ABC [4] as our constraint solver. As we explained in Section 3, other model-counting constraint solvers can be integrated instead of ABC by providing an appropriate translator (and, optionally, a model-counting-object evaluator).

All the experiments were run on an Intel Core i7-6850 3.5 GHz computer running Linux 4.4.0. The machine has 128 GB RAM, of which 4 GB were allocated for the Java VM.

8.1 Model counting over the SMC/Kaluza string constraint dataset

The Kaluza dataset is a well-known benchmark of string constraints that are generated by dynamic symbolic execution of real-world JavaScript applications [44]. The authors of the SMC solver [36] translated the satisfiable constraints to their input format: one contains 1,342 big, while the other contains 17,554 small where big and small classification is done based on the constraint sizes in the Kaluza dataset. We shall refer to the former as the original SMC-Big and to the latter as the original SMC-Small.

Duplicate constraints. While inspecting the results of our normalization, we found out that many of the files within each dataset are identical (indistinguishable by `diff`). Due to the presence of duplicates, even trivial caching (without any normalization) will yield some benefit on the original datasets. After removing all duplicate files, only 359 of the 1,342 constraints in SMC-Big and 9,745 of the 17,554 constraints in SMC-Small were found to be unique. As we discuss below, our normalization procedure allows further reductions in this dataset, increasing the benefits of caching well beyond what can be achieved with trivial caching.

Model counting. Since these constraints correspond to path conditions from symbolic execution, counting the number of satisfying models of each one could be necessary for quantitative analysis. We model-counted all constraints in each set as a simple way to emulate the behavioral pattern (w.r.t. caching) of one or more users performing quantitative analyses on the original programs.

When counting the models of a constraint over unbounded strings, in order to avoid infinite counts, one needs to set a bound on the length of strings to be considered. In this experiment, we set the bound to 50 characters for both sets. We ran the whole dataset (model-counting each constraint) first without normalization or caching, and then again with Cashew normalization and caching enabled. In non-caching mode, each constraint was sent unmodified to the model-counting solver. In caching mode, the cache was cleared before running SMC-Big, and again before running SMC-Small. Since these path constraints were produced by an external symbolic executor, in this experiment we did not use SPF. Instead, Cashew was run in standalone mode. Note that since all constraints were model-counted, the order in which we traverse the datasets does not matter, as each normalized constraint will fall within some orbit, and for that orbit, the full cost of model counting will be paid exactly once (on the first cache miss).

Results. Table 1 shows the total, maximum and average model-counting time, as well as the speedups obtained by Cashew on each of these metrics, for the two datasets with and without duplicates. On the SMC-Big set, Cashew achieved a speedup over 10x. On the SMC-Small set, which is a rather bad case for the caching trade-off because it contains a large number of very small constraints, Cashew still achieved a 2.19x speedup.

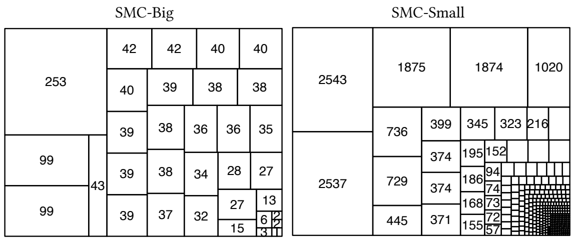

Figure 4: Orbit sizes for the original SMC datasets.

		Without caching	With caching	Speedup
Big (no dups)	Average	8.94 s	0.82 s	10.90x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	3,208.65 s	293.21 s	10.94x
Small (no dups)	Average	0.12 s	0.05 s	2.40x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	1,211.09 s	552.56 s	2.19x
Big (original)	Average	23.32 s	0.26 s	89.70x
	Maximum	121.92 s	40.13 s	3.03x
	Total time	31,297.90 s	358.17 s	87.38x
Small (original)	Average	0.13 s	0.05 s	2.60x
	Maximum	1.09 s	1.12 s	0.97x
	Total time	2,221.91 s	971.50 s	2.29x

Table 1: Model counting SMC-Big and SMC-Small.

For the original datasets, these numbers (e.g., a 87x speedup) are largely due to the presence of duplicates, which makes even trivial caching very effective. We report the results because the original datasets are widely used, and because the duplicates might indeed have been genuinely generated by symbolic execution of various different (yet similar) JavaScript programs.

Figure 4 depicts the effect of our normalization procedure on the original benchmarks. The area of each orbit is proportional to its size. Labels indicating orbit size are shown only when they fit in the available space. For the original SMC-Small set, the 17,554 original constraints are reduced to 360 orbits. For the SMC-Big set, the 1,342 original constraints are reduced to just 34 orbits.

We do not compare Cashew with Green because the original Green (without Cashew) cannot handle string constraints.

Note that the largest constraint in SMC-Small takes slightly more time after normalization. We cannot infer much from this, because the largest constraint in SMC-Small barely takes one second; the small difference (about 30 msec) could be due to noise. However, the maximum time for SMC-Big decreased by 3x with caching enabled, from 122 to 40 seconds. This is due to normalization. The constraint that (without normalization) requires maximum time to be model-counted falls within some orbit. It does not matter which constraint in that orbit will be the one to cause a cache miss once caching is enabled — only one of them will, and as they are all normalized to the same normal form, any of them would take the same model-counting time. What is interesting is that said time can be significantly smaller than the maximum pre-normalization time.

Table 2 shows the number of orbits that are achieved by different subsets of the transformations in our normalization procedure. Since some transformations can benefit from others, instead of considering them in isolation, we measured the effect of disabling each one. We did not include $\sigma_{\mathcal{SH}}$ as it doesn't apply to the string

Transformations enabled	#Orbits (SMC-Big)	#Orbits (SMC-Small)
None	359	9754
All transformations	34	360
All except $\sigma_{\mathcal{I}}$	72	376
All except $\sigma_{\mathbb{V}}$	344	9645
All except σ_{Σ}	35	841
All except removeVar	34	361
All except removeConj	40	386

Table 2: Effect of transformations on orbit refinement.

domain. The removeVar and removeConj transformations are preprocessing steps that remove redundant variables and conjuncts, respectively. These results indicate that all transformations yield some benefit, and that $\sigma_{\mathbb{V}}$ is the most beneficial transformation overall. For SMC-Small, removing σ_{Σ} more than doubles the number of orbits. The same is true of $\sigma_{\mathcal{I}}$ for SMC-Small. This shows that different transformations can be more effective for different datasets.

8.2 SPF analysis of string-handling code

In this second part of the experimental evaluation we use Symbolic PathFinder [39] with Cashew, to symbolically execute Java programs that operate on strings. In order to support model-counting-based quantitative analyses, we are interested in obtaining a model count for each leaf path constraint.

As an example of quantitative information flow analysis, we study some possible applications of Cashew to side-channel analysis. We consider four Java programs in which a side channel can allow an attacker to gain information about a hidden secret. `PasswordCheck1` contains a method that checks whether or not a user-given string matches a secret password. Due to the way the program is written, the attacker can deduce that the longer the program executes, the longer a prefix of the hidden password was matched. `PasswordCheck2` is another variant that attempts to mitigate that vulnerability by requiring a certain number of characters to be compared before returning, even if a mismatch has already been found. This yields a more interesting side channel, which can still be exploited but is much noisier and less predictable. `Obscure` is a Java translation of the `obscure.c` program used in [36], which is a password change authorizer. Given an old password and a new one, `Obscure` performs a series of tests to determine whether the new password is different enough from the old one. `CRIME` is a Java version of a well-known attack, *Compression Ratio Info-leak Made Easy* [7, 43]. This is a side channel in space — the secret is concatenated with a string that can be controlled by the attacker, and both are compressed together before encryption. Thus, the attacker can try various strings and observe the changes in the size of the compressed payload to infer, from the compression rate, the level of similarity between the secret and the injected content.

In symbolic execution, it is not always desirable to make *all* arguments of a method symbolic. This is often the case due to scalability issues. It can also be due to the need to explore a nonstandard distribution of some parameter. Consider, for instance, a situation where a list of passwords from a website is unwillingly disclosed to the public. As a consequence, users are strongly encouraged to change their passwords, and an algorithm similar to the `Obscure` program is employed to ensure that they are sufficiently different from the stolen ones. We might be interested in measuring the amount of

Program	Caching	Total time	Speedup	#Hits	#Misses	H/M
Password1	None	297 s	-	-	-	-
	Trivial	258 s	1.15x	17,547	56,173	0.31
	Cashew	106 s	2.80x	62,797	10,923	5.75
Password2	None	3,364 s	-	-	-	-
	Trivial	3,379 s	0.99x	30,448	824,832	0.04
	Cashew	1,243 s	2.71x	659,804	195,476	3.38
Obscure	None	2,158 s	-	-	-	-
	Trivial	1,965 s	1.10x	2,000	59,000	0.03
	Cashew	609 s	3.54x	44,893	16,107	2.79
CRIME	None	3,005 s	-	-	-	-
	Trivial	2,941 s	1.02x	31,884	84,127	0.38
	Cashew	1,067 s	2.82x	78,289	37,722	2.08

Table 3: SPF-based quantitative analyses of string programs.

	Total time (no cache)	Total time (Cashew)	Total norm. time	Total norm. calls	Average norm. time	Total cache size
SMC-Big	3,209 s	293 s	<3 s	359	8 ms	54 KB
SMC-Small	1,211 s	553 s	31 s	9,745	3 ms	104 KB
Password1	297 s	106 s	20 s	73,720	275 μs	2.9 MB
Password2	3,364 s	1,243 s	94 s	855,280	110 μs	58 MB
Obscure	2,158 s	609 s	19 s	61,000	300 μs	5.1 MB
CRIME	3,005 s	1,067 s	47 s	116,011	400 μs	8.8 MB

Table 4: Cashew normalization and caching costs.

leakage of the algorithm over that particular list of passwords. By running SPF on `Obscure` with the new password as a symbolic string, and the old password as a concrete string, we can measure the leakage for that particular stolen password. By repeating this for various passwords from the list, we can quantify the algorithm's leakage for that list's particular distribution. Doing so requires running SPF repeatedly on the same code, but with different secret strings. This will affect many path conditions in fundamental ways, but others might be unaffected, or changed in such a way that Cashew can still normalize them down to a previously seen one.

RockYou. The RockYou1K dataset is a sample of 1,000 real-world passwords taken from the RockYou leak [52] without duplicates. The sample consists of 1,000 unique passwords that cover all lengths between 6 and 16 characters, and can include any ASCII symbols.

Results. For each of the four programs under analysis, we ran 1,000 symbolic-execution-based side-channel analyses, using as the secret each of the 1,000 passwords in the RockYou1K dataset. For `PasswordCheck1` and `PasswordCheck2`, the secret is the password, which is concrete, and the user's guess is a symbolic string. For `Obscure`, the roles are reversed: what we made concrete is the old password, which is no longer secret, whereas the user-chosen new password (which is secret) is symbolic. For `CRIME`, we used a concrete secret (session ID) and a symbolic user-injected payload.

Table 3 shows execution time, hits and misses for three execution modes. The first mode uses neither normalization nor caching. In the second mode, only *trivial caching* is performed — normalization is disabled, which measures the extent to which syntactically identical path conditions (akin to the duplicates mentioned in Section 8.1) are generated. In the third mode, Cashew's normalization is enabled. Note that each symbolic execution generates many path conditions. The tables show the aggregated results over all path conditions of each execution and the 1,000 executions of each mode. As in the previous section, we do not compare Cashew with Green in these experiments because the original Green (without Cashew) cannot handle string constraints.

The results show that, for these experimental subjects, Cashew achieved an average speedup of nearly 3x, while trivial caching only achieved 1.06x on average (and, for `PasswordCheck2`, was in fact slower than no caching). Additionally, the hit/miss ratios improve dramatically when switching from trivial caching to Cashew.

Costs of caching and normalization. A caching scheme involves overheads and space/time trade-offs. Normalization overhead must be kept low, since its cost must be paid not only for each hit, but also for each miss. Cache size must also be kept within reasonable limits. Cashew is implemented on top of Green and, like Green, uses the in-memory Redis [1] database by default. This allows extremely fast queries, but competes with the client application for available RAM. As Table 4 shows, the average time to normalize a constraint in our SPF symbolic execution experiments was only a few hundred microseconds. It was about 8 milliseconds for the largest formulas (the SMC-Big set, with an average size before normalization of about 10 KB of text per constraint). Finally, as shown in Table 4, the total cache memory usage was very reasonable for these experiments.

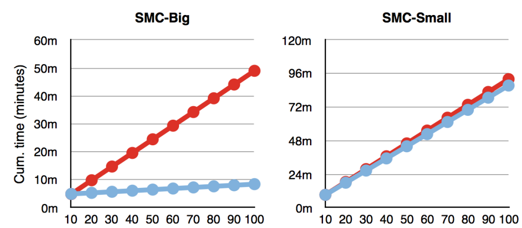

Figure 5: SMC datasets for increasing bounds. Cashew: non-parameterized (above) and parameterized (below) caching.

8.3 Parameterized caching

In this last part of the experimental evaluation we present some experiments for evaluating parameterized caching—that is, caching that leverages parameterized model-counting solvers.

The motivation behind these experiments is that users of Cashew who are targeting quantitative information analysis techniques often perform their analyses with various different bound values. For example, in side-channel analysis, one may want to compute the amount of information leakage for different lengths of a symbolic secret or input. This requires using different bounds when counting models. In scenarios where we have reason to believe that there is potential for reusing already-created model-counting objects for multiple values of b, we can try to amortize the time required to construct them by caching them.

String constraints: SMC/Kaluza. Recall the SMC-Big and SMC-Small datasets from Section 8.1. We ran these two datasets several more times, starting with the string length bound b at 10 characters and raising it up to 100 characters. Since our goal was to evaluate the usefulness of caching model-counting objects, we did not clear the cache between successive values of b. Again, we did not compare with Green in these experiments because Green (without Cashew) does not support constraints over strings.

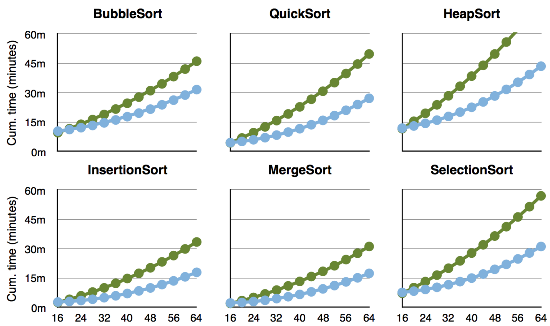

Figure 6: Symbolic execution of sorting/searching programs for increasing bounds. Green (above) vs. Cashew (below).

Figure 5 shows the cumulative time spent running SMC-Big and SMC-Small, respectively, for $b \in \{10, 20, 30, \ldots, 100\}$ characters. We did this twice for each dataset. The upper lines (red) correspond to Cashew with parameterized caching disabled (model-counting objects are not cached). The lower lines (blue) correspond to Cashew with parameterized caching enabled. In this mode, an extra cost is paid to cache the model-counting objects, but doing so enables the possibility of reusing them later on. The left chart shows that caching model-counting objects is indeed beneficial for SMC-Big. This is an idealized amortization scenario, since all stored model-counting objects are reused on each successive bound value. Nevertheless, it is useful to confirm that for this dataset, running even one additional bound is profitable, and that this profit becomes larger each time we run the dataset for an additional value of b. The right chart shows a similar phenomenon for SMC-Small, but although the gap does increase, the lines are so close together that caching model-counting objects would probably not be worth its cost. This is consistent with a large number of small problems.

Arithmetic constraints. The goal of these experiments is to evaluate the usefulness of Cashew's parameterized caching when symbolically executing Java code whose branch conditions involve linear integer arithmetic operations. Green can handle arithmetic constraints, so we can use it as the baseline for these experiments.

One well-known class of algorithms that involve integer arithmetic constraints and give rise to nontrivial path conditions are classical sorting algorithms. For these experiments we ran an SPF-based quantitative analysis (symbolic execution and model counting on complete path conditions) on the following algorithms: BubbleSort, InsertionSort, SelectionSort, QuickSort, HeapSort, and MergeSort.

Figure 6 shows the cumulative time spent in the analysis of each of the seven Java programs, for $b \in \{16, 20, 24, \ldots 64\}$. Since we are counting over the integers, the bound b now denotes the maximum number of bits that may be used to represent an integer. We ran each series twice. The upper curve (green) corresponds to Green, with caching enabled, using its normalization procedure for integer arithmetic constraints. The lower curve (blue) corresponds to Cashew with parameterized caching enabled.

The magnitude of the gap between both curves varies for different programs. In most cases, the initial run on an empty cache (for $b = 16$) is slightly more costly for Cashew due to the overhead of having to store all the model-counting objects in the cache. This is compensated as soon as they are reused at least once, and in all cases we see that the gap between the curves grows as the model-counting objects are reused further. This confirms that parameterized caching is beneficial for these programs if there is a reasonable chance that the model-counting objects may be reused.

9 RELATED WORK

Our work builds on top of Green [51], an external framework for caching the results of calls to satisfiability solvers or model checkers developed by Visser et al. Green has been extended by Jia et. al in their tool GreenTrie [27], which, for a given target formula, efficiently queries the cache for satisfiable formulas that imply it or unsatisfiable formulas implied by it. This allows for additional resuse when GreenTrie is able to detect an implication relation between the target constraint and one in the database. Another caching framework Recal [3], transforms a linear integer arithmetic constraint to a matrix, canonizes it, and uses the result as a normal form with which the query the database. Like GreenTrie, Recal is able to detect some implications between constraints. Kopp et al. [30] also develop a framework for caching, which like ours, uses a group theoretic framework to define a normal form for constraints.

Cashew differs notably from these previous caching frameworks. First, we present a parameterized model counting approach for quantative program analysis which allows us to cache and reuse a model-counter in addition to the results of model counting queries. This allows us to reuse results for model countring queries across different bounds. Cashew also exploits more expressive normalization techniques with reductions that preserve only the *number* of solutions of a constraint instead of their solution set. This allows us to reuse information that the above caching frameworks could not. Cashew is also able to handle string constraints, extending its applicablity to analyses over string manipulating code. In constrant, Green, GreenTrie and Recal only support caching over the domain of quantifier-free linear integer arithmetic. The work of Kopp et al. is built over a domain of first order formulas restricted to predicates, variables, quantifiers and logical connectives but no constants or function symbols and their framework is not implemented.

10 CONCLUSIONS

We provided a general group-theoretic framework for constraint normalization, and presented constraint normalization techniques for string constraits, arithmetic constraints and their combinations. We extended constraint caching to string constraints and combinations of string and arithmetic constraints. We presented constraint normalization techniques for quantitative program analysis that preserve the cardinality of the solution set for a given constraint but not necessarily the solution set itself. We presented parameterized constraint caching techniques that can reuse the result of a previous model counting query even if the bounds of the queries do not match. Our experiments demonstrate that, when combined with our constraint normalization approach, constraint caching can significantly improve the performance of quantitative program analyses.

REFERENCES

[1] Redis. https://redis.io/.
[2] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 150–166, 2014.
[3] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè. Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 305–315. ACM, 2015.
[4] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, Proceedings, Part I*, pages 255–272, 2015. doi: 10.1007/978-3-319-21690-4_15.
[5] M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 141–153, 2009.
[6] V. Baldoni, N. Berline, J. D. Loera, B. Dutra, M. Köppe, S. Moreinis, G. Pinto, M. Vergne, and J. Wu. Latte integrale v1.7.2. http://www.math.ucdavis.edu/ latte/, 2004.
[7] L. Bang, A. Aydin, Q.-S. Phan, C. S. Păsăreanu, and T. Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 193–204. ACM, 2016.
[8] C. Barrett, L. de Moura, S. Ranise, A. Stump, and C. Tinelli. The smt-lib initiative and the rise of smt. In *Haifa Verification Conference*, pages 3–3. Springer, 2010.
[9] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.
[10] C. Barrett, M. Deters, L. De Moura, A. Oliveras, and A. Stump. 6 years of smt-comp. *Journal of Automated Reasoning*, 50(3):243–277, 2013.
[11] M. Borges, A. Filieri, M. d'Amorim, and C. S. Pasareanu. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 866–877, 2015.
[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.
[13] S. Chakraborty, K. S. Meel, R. Mistry, and M. Y. Vardi. Approximate probabilistic inference via word-level counting. *arXiv preprint arXiv:1511.07663*, 2015.
[14] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15(3):321–371, 2007.
[15] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*. Citeseer, 1992.
[16] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. *KR*, 96:148–159, 1996.
[17] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
[18] B. Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.
[19] A. Filieri, C. S. Pasareanu, and W. Visser. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 622–631, 2013.
[20] V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *Proceedings of the 8th International Haifa Verification Conference (HVC)*, pages 209–226, 2012.
[21] J. Geldenhuys, M. B. Dwyer, and W. Visser. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 166–176, 2012.
[22] I. P. Gent and B. Smith. *Symmetry breaking during search in constraint programming*. Citeseer, 1999.
[23] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
[24] J. Heusser and P. Malacaria. Quantifying information leaks in software. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, pages 261–269, 2010.
[25] 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 (PLDI)*, pages 188–198, 2009.
[26] P. Hooimeijer and W. Weimer. Solving string constraints lazily. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 377–386, 2010.
[27] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2015.
[28] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 553–568, 2003.
[29] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, pages 105–116, 2009.
[30] T. Kopp, P. Singla, and H. Kautz. Toward caching symmetrical subtheories for weighted model counting. In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
[31] G. Li and I. Ghosh. PASS: string solving with parameterized array and interval automaton. In *Proceedings of the 9th International Haifa Verification Conference (HVC)*, pages 15–31, 2013.
[32] T. Liang, N. Tsiskaridze, A. Reynolds, C. Tinelli, and C. Barrett. A decision procedure for regular membership and length constraints over unbounded strings. In C. Lutz and S. Ranise, editors, *Proceedings of the 10th International Symposium on Frontiers of Combining Systems*, volume 9322 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2015.
[33] T. Liang, A. Reynolds, N. Tsiskaridze, C. Tinelli, C. Barrett, and M. Deters. An efficient smt solver for string constraints. *Formal Methods in System Design*, 48 (3):206–234, 2016.
[34] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation*, 38(4): 1273 – 1302, 2004. ISSN 0747-7171. doi: http://dx.doi.org/10.1016/j.jsc.2003.04.003.
[35] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 575–586. ACM, 2014.
[36] L. Luu, S. Shinde, P. Saxena, and B. Demsky. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 57, 2014.
[37] B. Mao, W. Hu, A. Althoff, J. Matai, J. Oberg, D. Mu, T. Sherwood, and R. Kastner. Quantifying timing-based information flow in cryptographic hardware. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 552–559. IEEE Press, 2015.
[38] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 193–205, 2008.
[39] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.
[40] Q. Phan, P. Malacaria, O. Tkachuk, and C. S. Pasareanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.
[41] Q. Phan, P. Malacaria, C. S. Pasareanu, and M. d'Amorim. Quantifying information leaks using reliability analysis. In *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA*, pages 105–108, 2014.
[42] Q.-S. Phan and P. Malacaria. Abstract model counting: a novel approach for quantification of information leaks. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 283–292. ACM, 2014.
[43] J. Rizzo and T. Duong. The crime attack. Ekoparty Security Conference, 2012.
[44] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.
[45] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.
[46] G. Smith. On the foundations of quantitative information flow. In *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 288–302, 2009.
[47] M. Thurley. sharpsat–counting models with advanced component caching and implicit bcp. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006.
[48] M. Trinh, D. Chu, and J. Jaffar. S3: A symbolic string solver for vulnerability detection in web applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1232–1243, 2014.
[49] C. G. Val, M. A. Enescu, S. Bayless, W. Aiello, and A. J. Hu. Precisely measuring quantitative information flow: 10k lines of code and beyond. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 31–46. IEEE, 2016.
[50] S. Verdoolaege. barvinok: User guide. *Version 0.23), Electronically available at http://www. kotnet. org/˜ skimo/barvinok*, 2007.
[51] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 58.

ACM, 2012.

[52] M. Weir, S. Aggarwal, M. P. Collins, and H. Stern. Testing metrics for password creation policies by attacking large sets of revealed passwords. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 162–175, 2010. doi: 10.1145/1866307.1866327.

[53] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 114–124, 2013.