Subformula Caching for Model Counting and Quantitative Program Analysis

William Eiers
UC Santa Barbara
Verification Lab
Santa Barbara, CA
weiers@ucsb.edu

Seemanta Saha
UC Santa Barbara
Verification Lab
Santa Barbara, CA
seemantasaha@ucsb.edu

Tegan Brennan
UC Santa Barbara
Verification Lab
Santa Barbara, CA
tegan@ucsb.edu

Tevfik Bultan
UC Santa Barbara
Verification Lab
Santa Barbara, CA
bultan@ucsb.edu

***Abstract*—Quantitative program analysis is an emerging area with applications to software reliability, quantitative information flow, side-channel detection and attack synthesis. Most quantitative program analysis techniques rely on model counting constraint solvers, which are typically the bottleneck for scalability. Although the effectiveness of formula caching in expediting expensive model-counting queries has been demonstrated in prior work, our key insight is that many subformulas are shared across non-identical constraints generated during program analyses. This has not been utilized by prior formula caching approaches. In this paper we present a subformula caching framework and integrate it into a model counting constraint solver. We experimentally evaluate its effectiveness under three quantitative program analysis scenarios: 1) model counting constraints generated by symbolic execution, 2) reliability analysis using probabilistic symbolic execution, 3) adaptive attack synthesis for side-channels. Our experimental results demonstrate that our subformula caching approach significantly improves the performance of quantitative program analysis.**

***Index Terms*—formula caching, model counting, quantitative program analysis**

I. Introduction

In the last two decades, constraint solvers have had a significant influence in automated software engineering, especially in areas such as software verification, analysis and security. The key factor in increasing effectiveness of constraint solvers in automating software engineering tasks is the fact that the efficiency of the constraint solvers has improved significantly. These research results demonstrate that, despite the well known worst-case complexity results, in practice, many software engineering tasks benefit from constraint solvers.

A model counting constraint solver computes the number of solutions for a given constraint within a given bound [4], [7], [14], [17], [18], [28]. Recently, model counting constraint solvers have also been applied to automating quantitative software verification, analysis and security tasks. The goal in quantitative program analysis is not to just give a "yes" or "no" answer, but to also quantify the result. For example, rather than answering if there is information leakage in a program with a "yes" or "no" answer, quantitative analysis techniques can compute the amount of information leaked. This type of analysis is crucial for many domains since "yes" or "no" answers may not be possible. For example, every password checker leaks some information about the password (even saying a password does not match a guess leaks information) but a faulty password checker may leak more information than necessary. A quantitative vulnerability detection tool can use a model counting constraint solver to quantify the amount of information leakage. As another example, most symbolic execution tools cannot guarantee absence of an assertion failure in general since they search the state space up to a certain execution depth. When combined with a model counting constraint solver, a symbolic execution tool can quantify the likelihood of reaching an unexplored part of the state space, hence providing a probabilistic upper bound on observing an assertion violation. Model counting constraint solvers have been used in probabilistic analysis [13], [26], reliability analysis [23], quantitative information flow [6], [8], [30], [31], and attack synthesis [9], [29].

With respect to algorithmic complexity, model counting problem is at least as difficult as satisfiability problem, hence, in the worst case model counting problem is also intractable like satisfiability. However, as recent results demonstrate, like constraint solvers, model counting constraint solvers can also be applied to realistic software verification, analysis and security tasks. And, as with the constraint solvers, improving the efficiency of model counting constraint solver can have a significant impact on automating software engineering tasks.

In this paper, we focus on improving the performance of model counting constraint solvers. In particular, we present techniques for reusing results from prior model counting queries to solve new queries. The idea of memoization (caching prior results of expensive operations in order to reuse them to improve efficiency) has been used for constraints in the past. For example, BDDs, a data structure commonly used for representing satisfying solutions to Boolean logic formulas, uses the concept of a compute-cache to store prior results. Since BDDs are a canonical form for Boolean functions, they enable a caching approach that guarantees a cache hit if an equivalent formula has been analyzed before. However, BDDs

This material is based on research supported by an Amazon Research Award, by NSF under Grant CCF-1817242, and by DARPA under the agreement number FA8750-15-2-0087. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

can only handle bounded domains and require bit-blasting to handle numeric or string values which could be inefficient.

Caching prior formulas based on normalization of their syntax for constraint satisfiability and model counting queries has also been investigated for more expressive theories such as linear artithmetic and string constraints and their combinations. However, these approaches rely on syntactic matching and, hence, can miss hits for equivalent formulas. Moreover, they rely on full formula caching and therefore miss opportunities for cache hits among subformulas.

In this paper, we present a novel approach for formula caching that combines features of caching techniques that are based on syntax and canonical representations (building off of work done in Cashew [15]). Our approach has the following features that separates it from all prior results in this domain: First, our caching approach caches intermediate subformulas that arise in the pre-order traversal of the full formula enabling cache hits for common subformulas. Second, our approach combines syntax-based caching, with caching via a canonical representation in order to reduce the cost of caching while increasing the number of cache hits. Third, our approach uses an automata-based constraint representation which enables us to have a canonical representation of string and numeric constraints and their combinations.

We demonstrate the effectiveness of our approach in three different scenarios: 1) We consider model counting queries on constraints generated from programs via symbolic execution. In this scenario model counting queries are generated for full path constraints after the symbolic execution is over. 2) We consider model counting queries on constraints generated during symbolic execution. In this scenario model counting queries are generated on the current path constraint for each branch during symbolic execution in order to assess the reliability level that can be achieved by analyzing each branch. 3) We consider model counting queries on constraints generated while synthesizing side-channel attacks. In this scenario each attack step is generated by symbolic execution of the code followed by model counting queries to determine the input that reveals most information about the secret. Our experiments demonstrate that the subformula caching techniques we present in this paper can sometimes improve the performance for the first scenario, they improve the performance in most cases for the second scenario, and they improve the performance significantly (sometimes more than an order-of-magnitude) for the third scenario. We also observe that subformula caching is more effective for string constraints (for which automata construction can be very costly) than for numeric constraints.

The rest of the paper is organized as follows: In Section II we provide motivation for our subformula caching approach. In Section III we briefly go over automata-based caching. In Section IV we describe our subformula caching approach. In Section V we discuss thee application scenarios for model counting constraint solvers. In Section VI we discuss some of the implementation details. In Section VII we present our experimental results, in Section VIII we discuss the related work, and, finally, in Section IX we state our conclusions.

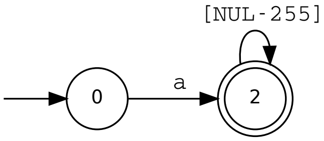


Fig. 1. DFA that accepts the solution sets of the formulas $\mathsf{char\_at}(v_0, 0) =$ "a" and $\mathsf{begins}(v_0,$ "a"$)$.

II. Motivation

Techniques we present in this paper aim to improve the performance of model counting queries generated during quantitative program analysis. In particular, we focus on automata-based model counting constraint solvers. In automata-based model counting the first task is to generate a deterministic finite automaton that accepts all solutions to a given formula.

Once the automaton accepting the solutions for a given formula is generated, the model counting query reduces to path counting: To find out the satisfying solutions within a bound, we need to count the number of accepting paths of a certain length or less that start with the initial state of the automaton and end in an accepting state. The path counting problem in graphs can be solved using matrix exponentiation (based on the adjacency matrix of the automaton), solving recurrence equations (automatically constructed based on the connections among the states of the automaton), or using generating functions (which can also be automatically constructed based on the connections among the states of the automaton).

Given a formula, the main difficulty in automata-based model counting is constructing a DFA accepting all solutions to that formula. Automata construction is exponential in the worst case as it may require determinization of an intermediate result automaton. Our caching techniques try to minimize the number of calls to automata construction operation.

We use two types of caching, which we call syntactic caching and automata caching, to characterize the way the keys are generated for the intermediate results we cache. In both cases the result we are caching is an automaton constructed for a given formula. In syntactic caching the key for storing the automaton constructed for a formula is generated based on the formula syntax. In automata caching we generate a key for each constructed automaton based on the structure of the automaton. We use minimized deterministic finite automata (DFA) which are a canonical representation. Hence, formulas with the same set of satisfying solutions are mapped to equivalent automata and the keys generated for them match if and only if the formulas are semantically equivalent.

Consider the following formulas:

$$\mathsf{length}(x) \leq 10 \land \mathsf{char\_at}(x, 0) = \text{“}b\text{”} \qquad (1)$$

$$\mathsf{begins}(s, \text{“}d\text{”}) \land \mathsf{length}(s) \leq 10 \land s = t \qquad (2)$$

Existing syntax-based formula caching techniques can be used to normalize these formulas in order to detect equivalent formulas. Normalization involves transformations such as variable renaming, character renaming, and sorting of the operations. Let us assume that the normalized form for the above formulas are:

$$\mathsf{length}(v_0) \leq 10 \wedge \mathsf{char\_at}(v_0, 0) = \text{“}a\text{”} \quad (3)$$

$$\mathsf{length}(v_0) \leq 10 \wedge \mathsf{begins}(v_0, \text{“}a\text{”}) \wedge v_0 = v_1 \quad (4)$$

Note that, syntactic normalization enables us to detect that formulas (1) and (2) have a common subformula $\mathsf{length}(v_0) \leq 10$. However, with full formula caching, since these formulas (1) and (2) are not equivalent, the fact that they share a subformula will not be exploited during automata construction or model counting. In this paper, we demonstrate that subformula caching, which stores automata constructed for intermediate subformulas during evaluation of the model counting queries, enables the reuse of the result for subformula $\mathsf{length}(v_0) \leq 10$.

For the above example, if model counting query for constraint (1) is processed before the model counting query for constraint (2), then based on syntactic subformula caching, we can detect that the subformula $\mathsf{length}(s) \leq 10$ is equivalent to $\mathsf{length}(x) \leq 10$ and use the stored automaton constructed for $\mathsf{length}(x) \leq 10$ rather than constructing a new (and equivalent) automaton for $\mathsf{length}(s) \leq 10$.

Above discussion explains our motivation for syntactic subformula caching, however it does not explain why we need automata caching. In syntactic caching we generate keys for the intermediate results using normalized syntax of the formulas. By automata caching we refer to generation of keys based on the structure of the automata, not the syntax of the corresponding formula. For example, the formulas $\mathsf{char\_at}(v_0, 0) = \text{“}a\text{”}$ and $\mathsf{begins}(v_0, \text{“}a\text{”})$ are syntactically different but they are semantically equivalent. The set of solutions to both of these formulas is characterized by the automaton shown in Figure 1.

Again, assume that a model counting query for formula (1) is processed before a model counting query for the formula (2). The automata constructed for subformulas $\mathsf{length}(x) \leq 10$ and $\mathsf{char\_at}(x, 0) = \text{“}b\text{”}$ and the full formula $\mathsf{length}(x) \leq 10 \wedge \mathsf{char\_at}(x, 0) = \text{“}b\text{”}$ will be stored in the cache. If we process a model counting query for formula (2) next, then, syntactic caching will report a hit on subformula $\mathsf{length}(s) \leq 10$ and will return the cached automaton for $\mathsf{length}(x) \leq 10$ instead of reconstructing an equivalent one. Then, the syntactic caching will report a miss for the subformula $\mathsf{begins}(v_0, \text{“}a\text{”})$ and an automaton for that subformula will be constructed. Next step is to construct the automaton for the subformula $\mathsf{length}(v_0) \leq 10 \wedge \mathsf{begins}(v_0, \text{“}a\text{”})$. Now, syntax based caching will report a hit for the first argument of the conjunction operation and the automata based caching will report a hit for the second operand of the conjunction operation. Then, instead of reconstructing the automaton corresponding to the conjuction, the cached automaton for the formula $\mathsf{length}(v_0) \leq 10 \wedge \mathsf{char\_at}(v_0, 0) = \text{“}a\text{”}$ will be returned. Then, the automaton for the subformula $v_0 = v_1$ will be constructed, followed by the construction of the automaton for the second conjunction. Note that syntax-based caching is necessary to reduce the number of calls to automata construction, and automata caching is necessary to catch the cases where syntax based caching is not able to detect equivalent formulas. In the following sections we will discuss the implementation of this caching approach.

III. Automata-Based Model-Counting

In this section we given an overview of automata-based constraint solving and model counting techniques which have been implemented in prior tools [4], [5]. We implemented our sub-formula caching approach by extending an existing automata-based model counting constraint solver.

Given an automaton A, let $\mathcal{L}(A)$ denote the set of strings accepted by A, and given a formula F let $[\![F]\!]$ denote the set of values that satisfy the formula F. In automata-based model counting, given a formula F the goal is to construct an automaton A_F where $\mathcal{L}(A_F) = [\![F]\!]$. Note that this requires encoding of the solutions to formula F as strings accepted by the automaton A_F.

In order to construct automata for formulas including string constraints, it is necessary to handle string operations such as concat, substring, length, char_at, begins, contains, etc. Using standard automata construction techniques (such as concatenation) and their extensions, automata construction techniques for string constraints have been implemented in prior work [4], [5] where the alphabet for the constructed automaton corresponds to the string alphabet used for the string constraints. Boolean operators negation, conjunction and disjunction are handled using automata complement and automata product, respectively. For constraints that involve multiple variables one can construct multiple single-track automata (one for each variable) or one multi-track automata that accepts tuples of strings. Multi-track automata is a generalization of finite state automata. A multi-track automaton accepts tuples of values by reading one symbol from each track in each transition. I.e., given an alphabet Σ, a k-track automaton has an alphabet Σ^k. In order to achieve better precision, we use multi-track automata representation.

Automata can also be used to represent solutions to linear arithmetic constraints. Similar to string constraints, the goal is to create an automaton that accepts solutions to the given formula. For numeric constraints, one can use the binary alphabet $\Sigma = \{0, 1\}$ where the set of solutions to the given numeric constraint is represented as a string of binary symbols that corresponds to the 2s-complement representation of the number which is the solution to the constraint. The numeric automata accept tuples of integer values in binary form, starting from the least significant digit. Numeric constraints consist of basic numeric constraints and Boolean logic operators. Each basic numeric constraint is in the form $\sum_{i=1}^{n} a_i \cdot x_i + a_0 \; op \; 0$, where $op \in \{=, \neq, >, \geq, \leq, <\}$, a_i denote integer coefficients and x_i denote integer variables. The automata construction for basic numeric constraints can be implemented using a basic binary adder state machine construction [11].

The automata construction techniques we summarize above have been implemented in a tool called Automata Based Counter (ABC) model counting constraint solver [4], [5].

The ABC tool is a constraint solver for string and numeric constrains and their combinations with model counting capabilities. Given a formula F, ABC constructs a multi-track deterministic finite-state automaton (DFA) A_F characterizing the set of solutions which satisfy F. For each atomic formula f in F, ABC constructs a DFA A_f for each and combines them into one DFA using automata operations (complement, product). The resulting DFA is an over-approximation of the set of all solutions to F. Note that ABC supports both string and numeric constraints, and thus uses two different encodings: ASCII for strings, binary encoding for integers. ABC keeps two different automata, one for string constraints, and one for integer constraints, and implements special operations for keeping track of relations between the two [4], [5].

ABC solves the the model counting problem using automata-based model-counting. Given a formula F constructing an automaton A_F for the set of solutions of F (where $\mathcal{L}(A_F) = [\![F]\!]$) reduces the model counting problem to a path counting problem. Note that the number of strings accepted by an automaton could be infinite in the presence of loops. In applications of model counting (such as probabilistic symbolic execution) a model counting query is accompanied with a bound that limits the domain of the variable. For string variables this is the length of the strings, whereas for numeric variables it is the number of bits. These correspond to the length of the accepted strings for our automata representation of string and numeric constraints.

The number of strings of length k in $[\![F]\!]$ corresponds to the number of accepting paths of length k in the DFA A_F. Since there is exactly one path for each string recognized by a DFA, if we can count the number of path in A_F precisely then we can answer the model counting query precisely.

Note that this approach works both for numeric and string constraint automata. Hence, using an automata-based constraint solver provides a general approach to model counting.

Computation of number of accepting paths within a bound can be done by constructing the adjacency matrix of the automaton based on its transition relation, and then using matrix exponentiation to compute the number of accepting paths. It is also possible to construct recurrence equations for the number of paths of a certain length that reach a particular state in terms of the number of paths that reach the adjacent states. The recurrence equation can be derived based on the connections among the states of the automaton. Finally, the number accepting paths of a certain length can be represented using generating functions where the generating function can be constructed based on the connections among the states of the automaton [4], [5].

IV. Caching for Model-Counting

Formula caching benefits quantitative program analyses by improving the performance of their enabling technology, model-counting constraint solvers. Formula caching frameworks allow model counters to reuse previously computed results and avoid performing expensive model counting. In the past, formula caching has been shown to improve the performance of model-counting constraint solvers by more than 10x [15].

Simple formula caching only attempts to reuse the results for the complete query (F, V, b). We instead integrate caching into the automata construction process of the model-counting constraint solver. This increases the potential for reuse. When constructing the automata for a formula F, we can reuse the automata of subformulas of F. For example, as we discussed earlier, in constructing the automata for the formula $\mathsf{begins}(s, \text{“}d\text{”}) \wedge \mathsf{length}(s) \leq 10 \wedge s = t$ we can reuse the automata constructed for the formula $\mathsf{length}(x) \leq 10 \wedge \mathsf{char\_at}(s, 0) = \text{“}b\text{”}$.

Extending formula caching with subformula caching allows us to avoid expensive construction steps by reusing results. To determine when results can be reused, caching frameworks must be able to quickly detect when two queries are equivalent with respect to model-counting. A formula F is said to be equivalent to formula G with respect to model counting if the cardinality of satisfying solutions to F matches that of G for any length bound b. Note that two formulas might be equivalent according to this criterion even if they do not possess the same solution set. Determining if two formulas satisfy this criteria is non-trivial. Syntactic caching and automata caching are two different normalization techniques to determine the equivalence of formula, both of which we use in conjunction with subformula caching.

A. Syntactic Caching

Under syntactic caching, the formulas of queries are transformed according to syntactic rules into a normal form. This normal form is then used as a key to the cache under which to store the automata. The constraint normalization procedure given in [15] provides an effective, albeit incomplete method of determining if two formula are equivalent with respect to model counting. The normalization procedure takes a query (F, V, b) and produces a normalized query $[F, V, b]$, with variables V and bound b. Two queries normalize to the same form only if they are equivalent with respect to model counting, that is, only if the cardinality of their solution sets match for every length bound.

We adopt the syntactic normalization procedure given in [15]. A query is normalized according to four subprocedures which act on its formula. First, the formula conjuncts are sorted. Then the variable names are normalized in order of appearance in the sorted formula. Third, alphabet constants are normalized again in order of appearance, and finally, arithmetic constraints are shifted by an integral amount to center them about the origin. Note that normalized alphabet characters are still treated as characters, regardless of which character they are normalized to.

As an example, consider formula F:

$$b = \text{“}.com\text{”} \wedge \mathsf{contains}(b, url)$$

and formula G:

$$\mathsf{contains}(s, link) \wedge s = \text{“}.net\text{”}$$

After sorting and renaming, both F and G normalize to the same form:

$$v_0 = \text{“}abcd\text{”} \wedge \mathsf{contains}(v_0, v_1)$$

which means that the automata constructed for one formula can be found in the cache and reused should a query be made on the other.

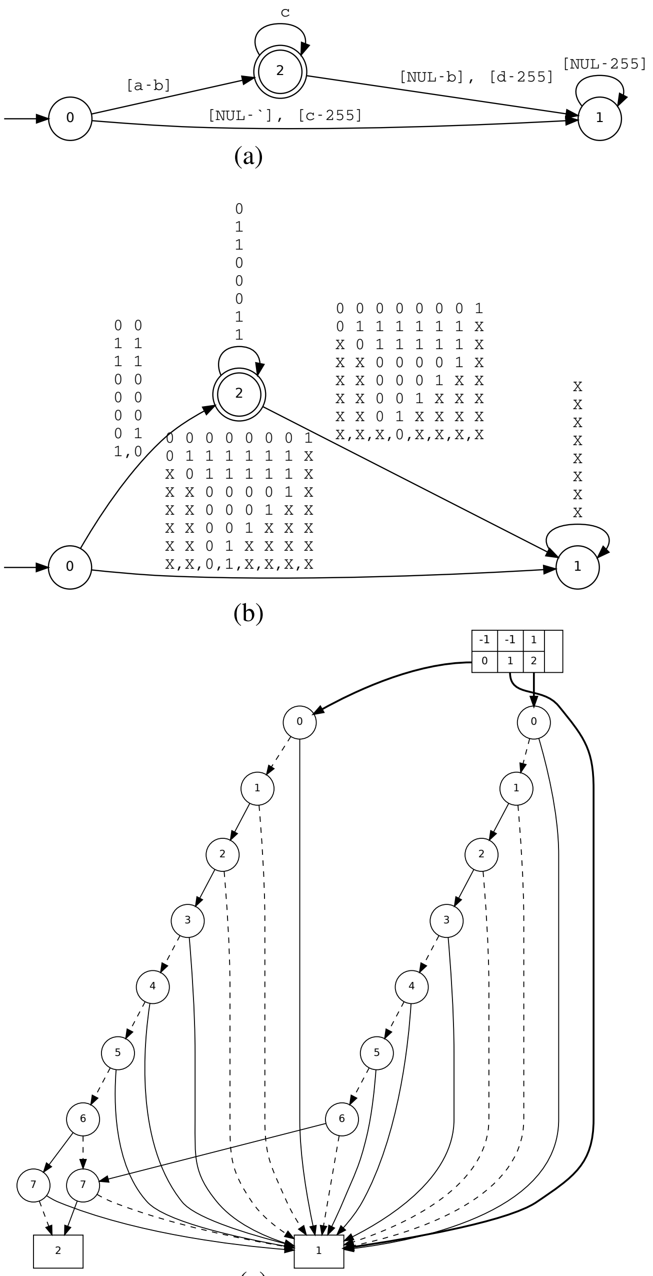
We use syntactic caching for both full-formula queries and sub-formula queries. When we receive a query on formula F, we first syntactically normalize F and use its normal form as a key to query the cache as given in Algorithm 3. When a hit occurs, we use the stored automata for path counting. If a miss occurs, we turn to subformula caching to determine if we can reuse intermediate results during automata construction of F. If $F \equiv \mathbf{op}\ F_1 \ldots F_n$ where **op** is any n-ary operator, then we perform two queries to the cache. One is on the syntactically normalized $\mathbf{op}\ F_1 \ldots F_{n-1}$ or if $n = 2$, F_1. The other is on F_n. When a hit occurs, the cached automata is used and the construction of $\mathbf{op}\ F_1 \ldots F_{n-1}$ or F_2 bypassed. If a miss occurs, querying continues recursively to $\mathbf{op}\ F_1 \ldots F_{n-2}$ and F_{n-1} until an atomic[1] formula is reached. When an atomic formula is reached, the automaton is constructed. Each time an automaton is constructed, we store the automaton in the cache under its syntactic key for future use.

In the example given above, the constraint F has two subformulas: b = “.com” and $\mathsf{contains}(b, url)$. In the case where the normalized form of F is not found in the cache, the normal forms of these two subformulas would be queried. b = “.com” normalizes to v_0 = “abcd” and $\mathsf{contains}(b, url)$ normalizes to $\mathsf{contains}(v_0, v_1)$. During the construction of G we get a hit since, after normalization, the key generated for G matches the key for F which means that the two formulas are equivalent as far as model counting is concerned.

B. Automata Caching

Formulas that are semantically equivalent can have different syntactic normal forms. To capture additional equivalent formulas, we use automata caching. Under this caching, the normal form of a formula is its automaton itself. For deterministic and minimized automata, two formula have the same automaton if and only if they are semantically equivalent formulas. This is true since minimized deterministic DFAs provide a canonical form for regular languages. Unlike syntactic caching, this type of equivalence check captures all semantically equivalent formulas.

When syntactic caching results in a cache hit, it is preferable to automata caching as its normalization is less expensive. We use automata caching when syntactic caching has failed on a query on a formula $F \equiv \mathbf{op}\ F_1 \ldots F_n$ where **op** is any n-ary operator. We construct the automata for each F_i. We then generate a key based on those automata and the operator **op** and query the cache with this key. If the resulting automaton for **op** has been previously constructed, we can reuse the result. This procedure is given in Algorithm 4. In cases where constructing the automaton for **op** is costly, the overhead of the caching queries is a beneficial trade-off. Each time we construct an automaton for a formula $F = \mathbf{op}\ F_1 \ldots F_n$, we generate its key for automata caching and store the result.



c
2
[a-b]
[NUL-b], [d-255]
[NUL-255]
0
[NUL-`], [c-255]
1
(a)
0 1 1 0 0 0 1 1
0 0 1 1 1 1 0 0 0 0 0 0 0 1 1,0
0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 X X 0 1 1 1 1 1 X X X 0 0 0 0 1 X X X 0 0 0 1 X X X X 0 0 1 X X X X X 0 1 X X X X X,X,0,1,X,X,X,X
0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 X X 0 1 1 1 1 1 X X X 0 0 0 0 1 X X X 0 0 0 1 X X X X 0 0 1 X X X X X 0 1 X X X X X,X,X,0,X,X,X,X
X X X X X X X X
2
0
1
(b)
-1 -1 1
0 1 2
0
1
2
3
4
5
6
7
2
1
(c)

Fig. 2. DFA constructed for the formula $\mathsf{match}(s, (a|b)c^*)$. (a) DFA with ASCII alphabet, (b) DFA with binary encoding of the ASCII symbols, (c) Multi-terminal BDD that encodes the DFA.

In our implementation of the automata caching we use the automata package provided by the MONA tool [16]. Generation of keys for deterministic finite automata require us to encode the automaton as a string. Consider the formula $\mathsf{match}(s, (a|b)c^*)$ which states that string variable s can take any value that matches the regular expression $(A|b)c^*$. In Fig. 2, we show the automaton constructed for this constraint. Fig. 2(a) shows the minimized DFA with the ASCII alphabet.

[1] For the definition of atomic formula, see [5]

```
{["{-1|<0> 0}|{-1|<1> 1}|{1|<2> 2}"];
  node; 0 [idx="0"]; 2 [idx="1"]; 3 [idx="2"]; 4 [idx="3"];
  5 [idx="4"]; 6 [idx="5"]; 7 [idx="6"]; 8 [idx="7"];
  10 [idx="7"]; 11 [idx="0"]; 12 [idx="1"]; 13 [idx="2"];
  14 [idx="3"]; 15 [idx="4"]; 16 [idx="5"]; 17 [idx="6"];
  terminal;1 ["1"];9 ["2"];
s:0 -> 0; s:1 -> 1; s:2 -> 11; 0 -> 2 [lo];
0 -> 1 [hi];2 -> 1 [lo];2 -> 3 [hi];3 -> 1 [lo];3 -> 4 [hi];
4 -> 5 [lo];4 -> 1 [hi];5 -> 6 [lo];5 -> 1 [hi];6 -> 7 [lo];
6 -> 1 [hi];7 -> 10 [lo];7 -> 8 [hi];8 -> 9 [lo];8 -> 1 [hi];
10 -> 1 [lo];10 -> 9 [hi];11 -> 12 [lo];11 -> 1 [hi];
12 -> 1 [lo];12 -> 13 [hi];13 -> 1 [lo];13 -> 14 [hi];
14 -> 15 [lo];14 -> 1 [hi];15 -> 16 [lo];15 -> 1 [hi];
16 -> 17 [lo];16 -> 1 [hi];17 -> 1 [lo];17 -> 10 [hi];}
```

Fig. 3. Key generated for the automaton in Figure 2 based on its multi-terminal BDD representation.

Initial state is 0, 2 is an accepting state and 1 is the sink state. Transitions are labeled with character ranges for readability.

In order to improve the efficiency of automata manipulation, MONA uses a symbolic DFA representation. The basic idea is to represent the transition relation of the automata symbolically using Multi Terminal Binary Decision Diagrams (MTBDDs) [25]. In order to do this, we first have to use a binary encoding of the set of characters that can appear in a string. Fig. 2(b) shows the DFA that is equivalent to the DFA shown in Fig. 2(a) where the ASCII symbols are encoded using 8-bit binary numbers (X denotes a “don’t care” value). Finally, Fig. 2(c) shows the symbolic DFA representation based on the MTBDD data structure. The second row in the table at the top represents the DFA states while the first row in that table represents state types which are either accepting state 1 or rejecting state -1. The circle-shaped nodes are the BDD nodes. Each circle-shaped node has a number n that represents its level i.e., which BDD variable n (in other words, which bit n in an alphabet symbol it corresponds to. Each rectangle-shaped leaf node has a number n that represents the destination state that the node corresponds to. Dashed line represents a BDD variable (bit) value of 0 while a regular line represents a BDD variable (bit) value of 1.

In Fig. 3 we show the key generated from the symbolic automata representation shown in Fig. 2(c). The key is a string that represents the nodes and the transitions in the MTBDD representation of the minimized DFA. Since minimized DFA representation is a canonical representation, given two formulas, the keys generated for them are identical if and only if the DFAs generated for them are identical.

C. Formula Caching Algorithm

Algorithm 2 outlines how we leverage syntactic caching and automata caching in conjunction with subformula and full-formula caching. Given a formula F, we first query whether the full formula F can be found in the cache through syntactic caching. This is the cheapest normalization scheme and would provide the most benefit, so we check it first. If this check fails and F is atomic, the cache can be of no further use to us, so we construct the automata and store it in the cache under the syntactic normal form of F. Otherwise, F is of the form $F \equiv$ **op** $F_1 \ldots F_n$ where **op** is some n-ary operator. In this case, sub-formula caching may benefit our construction process. As described above, we first syntactically query for the normalized form of **op** $F_1 \ldots F_{n-1}$ or if $n = 2$, F_1 and F_n. This querying continues recursively until either an atomic formula is reached or a cache hit occurs. Once the two automata have been either retrieved or constructed and stored under the syntactic normal form of the subformula, we use automata caching to potentially avoid an expensive construction of the **op** automata. We query using a key generated from the two automata and the **op** operator and either use the stored result or construct and store the automata.

Algorithm 1 ModelCounting(F, V, b):

Input: A formula F, set of variables V, and bound b
Output: The number of solutions to V that satisfy F within bound b.

```
$A_F$ = AutomataConstruction($F$)
return PathCount($A_F, V, b$)
```

Algorithm 2 AutomataConstruction(F):

Input: A formula F.
Output: An automata accepting all solutions of F.

```
$A_F$ = SyntaxCaching($F$)
if $A_F$ is not NULL then
    return $A_F$
end if
if IsAtomic($F$) then
    $A_F$ = ConstructDFA($F$)
    Store(Normalize($F$), $A_F$)
    return $A_F$
else
    $F$ = op $F_1 \ldots F_n$
    $A_1$ = AutomataConstruction(op $F_1 \ldots F_{n-1}$)
    $A_2$ = AutomataConstruction($F_n$)
    $A_F$ = AutomataCaching(op, $A_1$, $A_2$)
    Store(Normalize($F$), $A_F$)
    return $A_F$
end if
```

Algorithm 3 SyntaxCaching(F):

Input: A formula F.
Output: A cached automata that accepts all solutions of F or NULL.

```
$K_F$ = Normalize($F$)
if Hit($K_F$) then
    return $A_F$ = Load($K_F$)
end if
return NULL
```

Algorithm 4 AutomataCaching$(A_1, A_2,$ **op**$)$:

Input: Two automata A_1, A_2 and an operator, **op**.
Output: Automata for A_1 op A_2.

```
$K_F$ = GenerateKey(op, $A_1$, $A_2$)
if Hit($K_F$) then
    return $A$ = Load($K_F$)
end if
$A$ = ConstructDFA($A_1$ op $A_2$)
Store($K_F$, $A$)
return $A$
```

V. Applications of Model Counting

In this section, we describe three different quantitative program analysis scenarios which use model-counting constraint solvers. For each scenario, we introduce the experimental benchmark we use to evaluate the effectiveness of our caching technique for the scenario.

A. Model Counting Constraints

The most straightforward application of model counting is, given a set of constraints, to simply count the number of accepting solutions for each. This is common in symbolic execution, where model counting queries are generated for full path constraints after symbolic execution has completed. For this scenario we consider two sets of full path constraints, each generated from a different symbolic execution engine.

Kaluza Benchmark. The Kaluza benchmark is widely used benchmark for evaluating constraint solvers and model counting constraint solvers. The benchmark is a set of satisfiable constraints generated via symbolic execution of JavaScript programs and were originally solved by Kaluza string solver [34]. The constraints in this benchmark require a constraint solver to be able to reason over string and numeric constraints and their combinations. All the constraints from this benchmark were later divided into two sets: KaluzaSmall and KaluzaBig. The input format of these constraints were translated into the SMTLib2 input format by the authors of ABC [4], [5]. The KaluzaSmall set contains 28059 constraints, while KaluzaBig 7061 constraints. Each constraint contains a query variable for which to model count. We evaluate the performance of different caching techniques on this benchmark, comparing the time taken to count the number of solution strings of length less than or equal to 50 for each constraint.

Sorting Constraints. We investigate the performance of our approach on constraints generated from symbolic execution of four different Java sorting programs: Quicksort, Bubblesort, Insertionsort, and Selectionsort. We fixed the array size of each to 7 elements and symbolic execution to a depth of 30. Each consists solely of numeric constraints, with the total number of constraints 12856, 5041, 5041, and 5041, respectively.

B. Reliability Analysis

One measure of program reliability is the probability that the program executes successfully. Symbolic execution provides a means to compute program reliability. One run of symbolic execution generates a series of path constraints characterizing complete program paths. Because symbolic execution requires a depth bound, it is possible that not all complete program paths will be generated. Performing model counting over the generated path constraints and dividing the count by the domain size gives the probability that a randomly chosen input will execute that particular program path. By computing this probability for each complete program path, we can determine what percentage of the input space is captured by the path constraints generated by symbolic execution and therefore provide a lower bound on the reliability of the program.

As an example, consider the password checking function in Figure 4. If this function were symbolically executed with length bound 4 for h, five path constraints would be generated. These constraints are given in Table I. The probability of a given path can be computed by diving the model count of the path constraint by the size of the domain. The bound of 4 on h is a small bound. In general, we have no guarantee on the length of h, meaning symbolic execution will require a depth bound to terminate. However, by leveraging model counting, we can execute a bounded symbolic execution and then compute what percentage of the input space leads to a program path that terminates within our depth bound. This gives us the percentage of input space we can confidently say will execute without failure and thus provides a lower bound on the reliability of the program. For the PasswordChecker example, imagine we limit the search depth so that the loop symbolically executes only 3 times. In this case, all program paths for which the first three characters match would not complete their symbolic execution. Covered probability p_c for reliability analysis will be then the summation of the probability of path constrains 1, 2 and 3 from Table I.

In practice, we are also often interested in guaranteeing a lower bound for program reliability. In this case, we can perform model counting at each step of symbolic execution to determine what percentage of input follows which path. This would allow us to guide the symbolic execution along the most probable paths in order to increase coverage most efficiently and stop execution once a certain coverage is reached. Conversely, one could also guide symbolic execution towards highly improbable paths in order to test corner cases.

TABLE I
Path constraints for program in Figure 4

i	Path Constraint	Observation	Probability
1	$\mathsf{char\_at}(l,0) \neq \mathsf{char\_at}(h,0)$	63	0.9000
2	$\mathsf{char\_at}(l,0) = \mathsf{char\_at}(h,0) \wedge$ $\mathsf{char\_at}(l,1) \neq \mathsf{char\_at}(h,1)$	78	0.0900
3	$\mathsf{char\_at}(l,0) = \mathsf{char\_at}(h,0) \wedge$ $\mathsf{char\_at}(l,1) = \mathsf{char\_at}(h,1) \wedge$ $\mathsf{char\_at}(l,2) \neq \mathsf{char\_at}(h,2)$	93	0.0090
4	$\mathsf{char\_at}(l,0) = \mathsf{char\_at}(h,0) \wedge$ $\mathsf{char\_at}(l,1) = \mathsf{char\_at}(h,1) \wedge$ $\mathsf{char\_at}(l,2) = \mathsf{char\_at}(h,2) \wedge$ $\mathsf{char\_at}(l,3) \neq \mathsf{char\_at}(h,3)$	108	0.0009
5	$\mathsf{char\_at}(l,0) = \mathsf{char\_at}(h,0) \wedge$ $\mathsf{char\_at}(l,1) = \mathsf{char\_at}(h,1) \wedge$ $\mathsf{char\_at}(l,2) = \mathsf{char\_at}(h,2) \wedge$ $\mathsf{char\_at}(l,3) = \mathsf{char\_at}(h,3)$	123	0.0001

Reliability Analysis Benchmark. This benchmark is a modified version of the experimental benchmark used in [24]. The original benchmark consists of numeric constraints only. We add more example programs involving string constraints. Examples with numeric constraints cover couple of sorting algorithms plus DaisyChain, a small program simulating a simplified flap controller of an aircraft and RobotGame, a program to determine and execute robot movements. Examples with string constraints cover several string manipulating methods: PasswordCheck compares secret password and user's input, StringEquals is a string library function which

```
public Boolean PasswordCheck(String h, String l) {
  for (int i = 0; i < h.length(); i++)
    if (h.charAt(i) != l.charAt(i))
      return false;
  return true;
}
```

Fig. 4. Password Checking example.

checks if two strings are equal or not, StringInequality checks lexicographical order of two strings character by character, EditDistance checks minimum edit distance of two strings, IndexOf is another string library function and Compress is a simple string compression function.

C. Attack Synthesis

We focus on adaptive attack synthesis for side-channel vulnerabilities. Attack synthesis techniques generate inputs in an iterative manner which, when fed to code that accesses the secret, reveal information about the secret based on the side-channel observations [9], [29], [32]. Symbolic execution is used to extract path constraints, automata-based model counting is used to estimate probabilities of execution paths, and optimization techniques are used to maximize information gain based on entropy. Consider the password checking function in Figure 4. The function has a timing side-channel and one can reveal the secret by measuring execution time. If h and l have no common prefix, the program will have the fastest execution since the loop body will be executed only once; If h and l have a common prefix of one character, a longer execution will be observed since the loop body executes twice. The case when h and l match completely, the program has the longest execution. An attacker can choose an input and use the timing observation to determine how much of a prefix of the input has matched the secret. Adaptive attack synthesis approach starts by automatically generating the path constraints using symbolic execution. It then uses these constraints to synthesize an attack which determines the value of the secret (h). Based on Shannon entropy, the remaining uncertainty of h can be computed to measure the progress of an attack.

At each step of an adaptive attack, attacker learns new information about h represented as a constraint on h based on the observed execution time. Suppose that the secret is “1337”. The initial uncertainty is $\log_2 10^4 = 13.13$ bits of information (assuming uniform distribution). Attack synthesis generates input “8229” at the first step and makes an observation with cost 63, which corresponds to constraint $\mathsf{char\_at}(h, 0) \neq 8$. Similarly, a second input, “0002”, implies $\mathsf{char\_at}(h, 0) \neq 0$. At the third step the input “1058” yields a different observation leading to updated constraint on h as below:

$\mathsf{char\_at}(h, 0) \neq 8 \wedge \mathsf{char\_at}(h, 0) \neq 0 \wedge \mathsf{char\_at}(h, 0) = 1 \wedge \mathsf{char\_at}(h, 1) \neq 0$

The updated constraint at an attack step has subformula from the previous step. For example, at attack step 2, constraint $\mathsf{char\_at}(h, 0) \neq 8 \wedge \mathsf{char\_at}(h, 0) \neq 0$ has subformula $\mathsf{char\_at}(h, 0) \neq 8$ from earlier step and at attack step 3, constraint $\mathsf{char\_at}(h, 0) \neq 8 \wedge \mathsf{char\_at}(h, 0) \neq 0 \wedge \mathsf{char\_at}(h, 0) = 1 \wedge \mathsf{char\_at}(h, 1) \neq 0$ has subformula $\mathsf{char\_at}(h, 0) \neq 8 \wedge \mathsf{char\_at}(h, 0) \neq 0$. A model counting tool without caching will re-count a number of formulas which was counted in the earlier steps. This is redundant and reduces the efficiency of attack synthesis. Results can be reused from prior iterations. Model counting is in the core of the attack synthesis process as it is repeatedly used to calculate information gain and progress of attack synthesis. Reusing model counting query results from earlier steps should improve the effectiveness of attack synthesis by reducing attack synthesis time.

Attack Synthesis Benchmark. This benchmark was previously used in [32], [33] to synthesize attacks for programs vulnerable to side-channels. Example functions used in this benchmark includes different string manipulation and arithmetic operations, setting different sizes and lengths to define the domain of secret value. The function PCI is an implementation of password checker comparing a user input and secret password but inducing a timing side channel due to early termination optimization. SE is a method from the Java String library to check equality of two strings and known to be vulnerable to timing side-channel [20]. A similar side-channel was discovered in indexOf (IO) method from the Java String library. Function ED is an implementation of a dynamic programming algorithm to compute minimum edit distance between two strings. Function CO is a basic compression algorithm which collapses repeated sub-strings within two strings. SI, SCOI and SCI functions check lexicographic inequality (<,>=) of two strings whereas first one compares the strings, second one includes concatenation operation with inequality and third one compares characters in the strings.

VI. Implementation

We implemented[2] the caching techniques presented in this paper into the Automata Based Model Counter (ABC) [4], [5]. Internally, automata within ABC are represented as multi-terminal binary decision diagrams, implemented using the tool MONA [16]. Given the constraint formula F in SMTLib2 format, ABC first constructs the abstract syntax tree (AST) in negation normal form representing F where the root node represents the satisfiability of F, leaf nodes correspond to variables and constants, and intermediate nodes represent string or integer terms with boolean connectives (and, or). The AST is simplified before DFA construction using several heuristics. *Dependency analysis* identifies independent components which may be solved separately. *Equivalence class generation* detects equivalent variables through equality clauses and chooses a single representative for the class, and *term re-write rules* eliminate redundant terms and propagate constants. ABC then performs post-order traversal on the simplified AST, where the DFA for each node is constructed from the DFAs of its children nodes.

We modify the constraint solving algorithm of ABC with support for both syntactic and automata caching on the nodes of the AST. In our implementation, we use the popular open

[2] subformula caching implementation and dataset available at https://github.com/vlab-cs-ucsb/ABC

source in-memory database store Redis [1] as the cache. We set the maximum database size to 8 GB, with a least recently used eviction policy (LRU). Note that the LRU algorithm Redis uses approximates the LRU set using sampling, 3 in this case. Given a constraint formula, ABC constructs the simplified AST representing the formula using the approach mentioned above. Prior to the post-order traversal for DFA construction, the cache is recursively queried for a smaller subset of the original formula until either an atomic formula is found, or a DFA is returned from a cache hit. Note that the key for each query is simply the string representation of the AST corresponding to the normalized form of a particular subformula. In either case, ABC begins its post-order DFA construction traversal from the corresponding AST node. For each subformula solved from this point, ABC stores the solution DFA into the cache. By exploiting the natural post-order traversal of ABC's constraint solving algorithm we maximize the probability of a cache hit while minimizing the number of cache queries.

VII. Experiments

We evaluate our caching technique across the three different quantitative program analysis scenarios described above. For each experimental scenario, we evaluate four different caching approaches. The NoCaching or NC approach performs the analysis with no caching of model-counting queries and serves as a baseline for comparison. The FullFormula or FF approach is an identical re-implementation of Cashew performs only syntactic normalization and only queries the cache for hits of the full formula of the model-counting query. The SubFormula or SF approach is also limited to syntactic normalization but performs recursive queries on the sub-formulas of the query formula when the full formula is not found in the cache. Finally, the SubFormula + Automata or SFA approach extends the SF approach with automata caching. The SFA approach is the most expressive caching scheme. We report the time in seconds for each benchmark program to complete (end-to-end) across these four caching scenarios. We also report the speedup demonstrated by the SFA approach versus both the NC and FF approaches.

A. *Experimental Setup*

For all experiments, we use a desktop machine with an Intel Core i5-2400S 2.50 GHz CPU and 32 GB of DDR3 RAM running Ubuntu 16.04, with a Linux 4.4.0-81 64-bit kernel. We used the OpenJDK 64-bit Java VM, build 1.8.0 171.

B. *Experimental Results*

We discuss how each of the four caching approaches perform across the three different quantitative program analysis scenarios. We evaluate under what kinds of analyses the SF and SFA approaches prove highly beneficial versus FF and NC and examine cases where the improvement was only marginal.

Model Counting. The results for model counting constraints generated by symbolic execution are given in Table II. We show results the simplified Kaluza benchmark and linear arithmetic constraints generated from running symbolic execution on a suite of sorting benchmarks. We found that out of 28059 of the constraints in KaluzaSmall, 647 were unique constraints after normalization, with the other 27412 being trivially satisfiable. KaluzaBig contained 376 unique constraints out of 7061 constraints, with the other 6685 constraints being of reasonable complexity. For both cases, SFA outperforms NC and FF. For the numeric constraints, SFA outperforms FF and NC in only one case. For the other three cases, the overhead of subformula caching outweighs any benefits gained due to the simplicity of the numeric constraints.

Reliability Analysis. The results on the reliability analysis benchmark are given in Table III. The upper half of the table shows the results on programs that produce only numeric constraints and the bottom half on programs that also contain string and mixed string and numeric constraints. We found no caching approach to be significantly beneficial in the benchmarks where only numeric constraints are encountered. In fact, because of the additional overhead of the FF and SFA approaches, we even observed a slight slowdown versus the NC or the more light-weight FF approach on some benchmark programs. Nevertheless, the additional overhead was never hugely debilitating and the SFA approach never took more than 15% longer than the NC or FF approaches.

On benchmarks with string or mixed string and numeric constraints, the SF approach demonstrated notable improvement over both the NC and FF approaches, and the SFA approach was even more successful. In some cases, the SFA approach was more than four-fold faster than either the NC or FF approaches. In all cases, some improvement was observed with the SFA approach. The reason for the significant improvement observed on benchmarks with string and mixed constraints lies in the expensive automata constructions demanded by those constraints. Numeric constraints, however, do not require expensive automata constructions making the effects of caching less beneficial. From these experiments, we learned that the SFA approach potentially provides enormous benefit when string or mixed constraints are encountered during the course of the analyses and does not significantly degrade performance when only numeric constraints are encountered. From this, we believe that enabling SFA caching is generally beneficial for reliability analysis but also note that the analyst could make an informed choice to enable should they have suspicions about the type of constraints likely to be encountered.

Attack Synthesis. The results on the attack synthesis benchmark are given in Table IV. As shown in the execution time under the NC approach, this quantitative program analysis is the most expensive of the three with some benchmark programs taking 5 hours to run when no caching is enabled. In all cases, the SF approach improved on the NC and FF approaches, even reducing a run-time of five hours to less than eighteen minutes for the SCI benchmark program. The SFA approach was able to even further improve these already impressive results. On some benchmarks, SFA demonstrated a more than twenty-fold improvement versus the NC and FF approaches. In all cases, the SFA approach was the fastest evaluated caching approach.

TABLE II
EXPERIMENTAL RESULTS FOR MODEL COUNTING CONSTRAINTS

Benchmark	NC Time(s)	FF Time(s)	SF Time(s)	SFA Time(s)	SFA Speedup v NC	SFA Speedup v FF
QuickSort	195.4	195.2	220.2	225.6	0.87x	0.87x
BubbleSort	124.1	123.8	127.1	133.2	0.93x	0.93x
InsertionSort	129.9	125.3	119.1	123.4	1.05x	1.02x
SelectionSort	122.2	121.8	131.6	144.4	0.85x	0.84x
KaluzaSmall	1173.6	1075.2	1065.3	990.6	1.18x	1.09x
KaluzaBig	5730.7	1247.3	1193.3	1176.1	4.87x	1.06x

TABLE III
EXPERIMENTAL RESULTS FOR RELIABILITY ANALYSIS

Benchmark	SE Depth	NC Time(s)	FF Time(s)	SF Time(s)	SFA Time(s)	SFA Speedup v. NC	SFA Speedup v FF
BubbleSort	20	4573.4	2364.8	2372.3	2335.1	1.96x	1.01x
InsertionSort	15	4183.6	4364.3	4303.9	4311.1	0.99x	1.01x
DaisyChain	30	106.4	107.7	108.3	122.1	0.87x	0.88x
RobotGame	30	80.7	80.7	79.2	80.8	1.00x	1.00x
PasswordCheck	50	830.9	836.0	932.9	648.9	1.29x	1.29x
StringEquals	50	1142.2	1196.8	1269.7	893.1	1.28x	1.34x
StringInequality	10	319.2	324.1	251.6	89.7	3.56x	3.61x
EditDistance	8	19241.7	19876.6	15764.1	8384.4	2.29x	2.37x
IndexOf	15	25451.2	26116.3	21457.1	8384.6	3.04x	3.11x
Compress	30	5342.2	5435.9	1868.9	1213.8	4.40x	4.48x

All benchmark programs evaluated under this program analysis scenario contain string constraints. Based on our observations from the reliability program analysis benchmarks, we think that the more expensive automata construction required for these constraints is part of the reason the SF and SFA approaches are so successful for these benchmarks.

VIII. RELATED WORK

Model Counting: As the enabling technology for quantitative program analyses, model-counting constraint solvers have received increasing focus from the research community. SMC [28] and S3# [35] are two model-counting constraint solvers over the string domain. LattE [7] is a model-counting constraint solver for linear integer arithmetic that uses the Barvinok [12] algorithm. ABC, which can handle string, numeric and mixed constraints, is more expressive than any of these model-counting constraint solvers and more precise than either of the string model counters.

Caching: Cashew [15] is a caching framework for model-counting queries which provides notable improvement on a variety of program analyses. Cashew is built atop Green [36], an external solver interface for reusing the results of satisfiability or model counting queries. Cashew introduces an aggressive normalization scheme and parameterized caching, allowing it to outperform Green. We adopt the normalization scheme used by Cashew, but introduce subformula caching into the automata construction process to enable more reuse of computation. We also leverage automata caching, a normalization technique guaranteeing completeness to leverage more information from the cache. We show how both of these techniques benefits three different program analyses scenarios with a direct comparison to the full-formula-only caching implemented by Cashew.

GreenTrie [27], another extension of Green, and Recal [2] are caching frameworks that detect implication between constraints to improve caching for satisfiability queries. Their techniques are specific to satisfiability queries and, in the general case, do not apply to model-counting queries considered in this paper. Utopia [3] proposes a technique to reuse results across formulas with similar solution sets but again, is specific to satisfiability queries and would not aid in model counting.

Incremental Solving: Many modern SMT solvers have built-in support to expedite the solving of similar constraints. CVC4 [10], Z3 [21], Yices [22] and MathSAT5 [19] are SMT solvers with incremental capabilities. These tools learn lemmas which can later be (re)used to solve similar constraints. During constraint solving, these solvers use a stack-based approach to keep track of the current solver context, pushing and popping learned lemmas as conjuncts are added or removed respectively. Incremental attack synthesis is an alternative approach that enables reuse of intermediate results obtained during attack synthesis [32]. However, incremental attack synthesis approach is a specialized heuristic for attack synthesis, whereas the subformula caching approach we present in this paper is general, and it is applicable to any quantitative program analysis technique that relies on model counting queries.

IX. CONCLUSIONS

Quantitative program analysis techniques rely on model counting constraint solvers and model counting queries can be very expensive. In this paper we introduced sub-formula

TABLE IV
Experimental Results for Attack Synthesis

Benchmark	NC Time(s)	FF Time(s)	SF Time(s)	SFA Time(s)	SFA Speedup v NC	SFA Speedup v FF
PCI	8227.5	2936.6	1363.7	1013.8	8.12x	2.90x
SE	7386.3	2968.7	3186.1	2283.2	3.24x	1.30x
SI	232.9	178.7	88.7	54.6	4.27x	3.27x
ED	18000.0	24126.2	8000.1	1652.2	10.89x	14.60x
IO	11167.8	3719.9	3603.1	1163.4	9.60x	3.20x
CO	1908.5	2239.9	1273.1	92.2	20.7x	24.30x
SCOI	320.3	207.1	75.1	56.8	5.64x	3.65x
SCI	18000.0	11155.6	1076.6	617.9	29.13x	18.05x

caching to improve the efficiency of quantitative program analysis techniques. We focus on automata-based model counting for string and numeric constraints. We use both syntactic and automata-based caching in order to reduce the number of times automata are constructed. We evaluate our approach in different scenarios and demonstrate that subformula caching can significantly improve the performance of quantitative program analysis techniques.

References

[1] Redis. https://redis.io/.

[2] Andrea Aquino, Francesco A Bianchi, Meixian Chen, Giovanni Denaro, and Mauro Pezzè. Reusing constraint proofs in program analysis. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 305–315. ACM, 2015.

[3] Andrea Aquino, Giovanni Denaro, and Mauro Pezzè. Heuristically matching solution spaces of arithmetic formulas to efficiently reuse solutions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 427–437. IEEE, 2017.

[4] Abdulbaki Aydin, Lucas Bang, and Tevfik 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.

[5] Abdulbaki Aydin, William Eiers, Lucas Bang, Tegan Brennan, Miroslav Gavrilov, Tevfik Bultan, and Fang Yu. Parameterized model counting for string and numeric constraints. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 400–410, 2018.

[6] Michael Backes, Boris Köpf, and Andrey 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.

[7] V Baldoni, N Berline, JD Loera, B Dutra, M Köppe, S Moreinis, G Pinto, M Vergne, and J Wu. Latte integrale v1. 7.2, 2004.

[8] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Pasareanu, and Tevfik Bultan. String analysis for side channels with segmented oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2016.

[9] Lucas Bang, Nicolás Rosner, and Tevfik Bultan. Online synthesis of adaptive side-channel attacks based on noisy observations. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 307–322, 2018.

[10] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[11] Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.

[12] Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. *Math. Oper. Res.*, 19(4):769–779, November 1994.

[13] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina 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.

[14] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Pasareanu. Model-counting approaches for nonlinear numerical constraints. In *Proceedings of the 9th International NASA Formal Methods Symposium*, pages 131–138, 2017.

[15] Tegan Brennan, Nestan Tsiskaridze, Nicolás Rosner, Abdulbaki Aydin, and Tevfik Bultan. Constraint normalization and parameterized caching for quantitative program analysis. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 535–546. ACM, 2017.

[16] BRICS. The MONA project. {http://www.brics.dk/mona/.

[17] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. Distribution-aware sampling and weighted model counting for SAT. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 1722–1730, 2014.

[18] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate probabilistic inference via word-level counting. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 3218–3224, 2016.

[19] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[20] Joel Sandin Daniel Mayer. Time trial: Racing towards practical remote timing attacks. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf, 2014.

[21] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[23] Antonio Filieri, Corina S. Pasareanu, and Willem 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.

[24] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. Reliability analysis in symbolic pathfinder. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 622–631, Piscataway, NJ, USA, 2013. IEEE Press.

[25] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. Multiterminal binary decision diagrams: An efficient data structure for matrix representation. *Formal Methods in System Design*, 10(2/3):149–169, 1997.

[26] Jaco Geldenhuys, Matthew B. Dwyer, and Willem 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.

[27] Xiangyang Jia, Carlo Ghezzi, and Shi 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] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian 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.

[29] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. Synthesis of adaptive side-channel attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pages 328–342, 2017.

[30] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Pasareanu, and Marcelo 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.

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

[32] Seemanta Saha, William Eiers, Burak Kadron, Lucas Bang, and Tevfik Bultan. Incremental adaptive attack synthesis. *http://arxiv.org/*, 2019.

[33] Seemanta Saha, Ismet Burak Kadron, William Eiers, Lucas Bang, and Tevfik Bultan. Attack synthesis for strings using meta-heuristics. *ACM SIGSOFT Software Engineering Notes*, 43(4):56–56, 2019.

[34] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*, pages 513–528. IEEE, 2010.

[35] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. Model counting for recursively-defined strings. In *International Conference on Computer Aided Verification*, pages 399–418. Springer, 2017.

[36] Willem Visser, Jaco Geldenhuys, and Matthew 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.