Symbolic String Verification: Combining String Analysis and Size Analysis[*]

Fang Yu, Tevfik Bultan, and Oscar H. Ibarra

Department of Computer Science
University of California, Santa Barbara, CA, USA
{yuf,bultan,ibarra}@cs.ucsb.edu

Abstract. We present an automata-based approach for symbolic verification of systems with unbounded string and integer variables. Particularly, we are interested in automatically discovering the relationships among the string and integer variables. The lengths of the strings in a regular language form a semilinear set. We present a novel construction for length automata that accept the unary or binary representations of the lengths of the strings in a regular language. These length automata can be integrated with an arithmetic automaton that recognizes the valuations of the integer variables at a program point. We propose a static analysis technique that uses these automata in a forward fixpoint computation with widening and is able to catch relationships among the lengths of the string variables and the values of the integer variables. This composite string and integer analysis enables us to verify properties that cannot be verified using string analysis or size analysis alone.

1 Introduction

Static analysis of strings in programs have been an active research area with the goal of finding and eliminating security vulnerabilities caused by misuse of string variables. There have been two separate branches of research in this area: 1) *String analysis* that focuses on statically identifying all possible values of a string expression at a program point in order to eliminate vulnerabilities such as SQL injection and cross-site scripting (XSS) attacks [1, 4, 14, 16], and 2) *Size analysis* that focuses on statically identifying all possible lengths of a string expression at a program point in order to eliminate buffer overflow errors [5, 7, 12]. In this paper we present an automata based composite symbolic verification technique that combines these two analyses with the goal of improving the precision of both. We use a forward fixpoint computation to compute the possible values of string and integer variables and to discover the relationships among the lengths of the string variables and integer variables.

Similar to prior size analysis techniques [5, 7, 12] we associate each string variable with an auxiliary integer variable that represents its length. At each program point, we symbolically compute all possible values of all integer variables (including the auxiliary variables), as well as all possible values of all string variables. The reachable values of all integer variables are over-approximated as a Presburger arithmetic (linear

[*] This work is supported by NSF grants CCF-0614002 and CCF-0716095.

arithmetic) formula and symbolically encoded as *arithmetic automata* [2,13]. Similar to some prior string analysis techniques [1,16], the values that string variables can take are over-approximated as regular languages and symbolically encoded as *string automata*. Our composite analysis is as a forward fixpoint computation with widening on these arithmetic and string automata.

There are two challenges we need to overcome to connect the information contained in the string automata and the arithmetic automata (hence, improving the precision of both) during our composite analysis: 1) Given a string automaton, we need to derive the arithmetic automaton that accepts the length of the language accepted by the string automaton, and 2) Given an arithmetic automaton, we need to restrict a string automaton so that the length of the language is accepted by the arithmetic automaton.

To tackle the first challenge, we present techniques for constructing a *length automata* for a given regular language. It is known that the length of the language accepted by a DFA forms a semilinear set. Given an arbitrary DFA, we are able to construct DFAs that accept either unary or binary representation of the length of its accepted words. The unary automaton can be used to identify the coefficients of the semilinear set, while the binary automaton can be composed with other arithmetic automata on integer variables to enforce or check length constraints.

To tackle the second challenge, we identify the boundary of the lengths of string variables from the arithmetic automaton. Precisely, we compute the lower and upper bound of the values of the string lengths accepted by the arithmetic automaton. We prove that, given a one-track arithmetic automaton, the lower bound forms a shortest path to an accepting state while the upper bound (if it exists) forms the longest loop-free path. Both can be computed in linear complexity to the size of the arithmetic automaton. We can restrict the target string automaton by intersecting the string automaton that accepts arbitrary strings within this boundary.

Finally, the performance of our analysis relies on efficient automata manipulation. We implement our analysis using a symbolic automata representation (MBDD representation from the MONA automata package) and leverage efficient manipulations on MBDDs, e.g., determinization and minimization.

Motivating Examples Below, we present two motivating examples to demonstrate the advantages of the composite string and size analysis technique proposed in this paper. Consider a *PHP segment* that secures an identified vulnerable point [16] at line 218 in `trans.php`, distributed with `MyEasyMarket-4.1`.

```
1: <?php $www = $_GET["www"];
2:       $l_otherinfo = "URL";
3:       $www = ereg_replace("[^A-Za-z0-9 .\-@://]","",$www);
4:       if(strlen($www)<$limit)
5:         echo "<td>" . $l_otherinfo . ": " . $www . "</td>"; ?>
```

Without proper sanitization (lines 3 and 4) of the user-controlled variable `$www`, an attacker can inject the string `<scriptsrc=http://evil.com/attack.js>` and perform a XSS attack at line 5. Above code prevents such attacks by: (1) removing abnormal characters from `$www` at line 3, and (2) limiting the length of `$www` at line 4. Our analysis shows that this code segment is free from attacks by showing that at line 5

(1) the length of the string `$www` is less than the allowed limit, and (2) under that limit the string variable `$www` cannot contain a value that matches the attack pattern. Note that if one performs solely size analysis, without knowing the contents of `$www`, the length of `$www` can not be determined precisely after line 3. On the other hand, if one performs solely string analysis, the branch condition at line 4 must be ignored. Both of these approximations may lead to false alarms.

Now, consider a standard `strlen` routine in C that returns the length of a given string by traversing each character until hitting the end character, i.e., `'\0'`. This kind of standard string routines are widely used in legacy C systems, e.g., Apache, Samba, Sendmail, and WuFTP.

```
unsinged int strlen(char *s){
1:    char *ptr = s;
2:    unsigned int cnt =0;
3:    while(*ptr != '\0'){
4:        ++ptr;
5:        ++cnt;
6:    }
7:    return cnt;  }
```

Let `*s`*.length* denote the size of the string pointed by `s`. An essential property of this routine is that at line 7, `cnt = *s`*.length*, which can be used as the summary of this routine and significantly alleviates size analysis overhead [5, 15], however, none of the size analysis tools prove this property before using it. Our composite analysis is capable of proving this property. We first construct an assertion (arithmetic) automaton that accepts the values that satisfy `cnt = *s`*.length*. We then conduct our composite analysis by computing the forward fixpoint with widening. Upon reaching the fixpoint, at line 7, (1) the arithmetic automaton actually catches the relation that $\texttt{*s}.length = \texttt{*ptr}.length + \texttt{cnt}$, and (2) the string automaton of `*ptr` only accepts $\{\epsilon\}$. We prove the property by showing that the intersection of the language of (1) and the length of the language of (2) is included in the language of the assertion automaton.

In addition to earlier work on string analysis [1,4,14,16] and size analysis [5,7,12] that motivated our work, there has been some recent work on analyzing string and integer variables together during symbolic execution [6,11,15]. Unlike our approach, these are unsound techniques targeted towards testing and they do not try to compute an over-approximation of the reachable states via widening. Hence, they cannot prove properties of above program segments. Compared to [8,9] that use abstract interpretation for reasoning relational properties among the contents of symbolic intervals of arrays, our analysis traverses concrete values of string and integer variables using automata and addresses language properties.

This paper is organized as follows. We present the length automata construction in Section 2. We present our composite analysis technique that integrates string and arithmetic analyses in Section 3. We present our experiments with our prototype tool in verifying small C routines, buffer-overflow benchmarks and PHP web applications in Section 4. We conclude the paper in Section 5.

2 Length Automata Construction

Given a string automaton M, we want to construct a DFA M_b (over a binary alphabet) such that $L(M_b)$ is the set of binary representations of the lengths of the words accepted by M. We tackle this problem in two steps. We first construct a DFA M_u (over a unary alphabet) such that $L(M_u)$ is the set of unary representations of the lengths of the words accepted by M. It is known that this set is a semilinear set. We identify the formula that represents the semilinear set from M_u. We then construct M_b from the formula, such that $w \in L(M_b)$ if and only if the binary value of w satisfies the formula. I.e., the unary representation of the binary value of w is in $L(M_u)$.

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$ where Q is a finite set of states, q_0 is the initial state, Σ is a finite set of symbols. $F : Q \to \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \to Q$ is the transition function. The cardinality of a finite set A is denoted as $\sharp A$. The set of arbitrary words over a finite alphabet Σ is denoted as Σ^*. The length of a word $w \in \Sigma^*$ is denoted as $|w|$. A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a $sink$ state. In the constructions below, we also ignore the transitions that lead to a sink state.

A string automaton M is a DFA that consists of a tuple of $\langle Q, q_0, B^k, \delta, F\rangle$. M accepts a set of words, where each symbol is encoded as a k-bit string.

Length Constraints on String Automata We are interested in characterizing lengths of the accepted words. We characterize these lengths as a set of natural numbers by a *length constraint*. Formally speaking, the length constraint of a given string automaton M is a formula f over a variable x, such that $f[c/x]$ evaluates to true if and only if there exists a word w, such that $w \in L(M)$ and $c = |w|$.

Property 1: For any DFA M, $\{|w| \mid w \in L(M)\}$ forms a semilinear set.

Property 2: For any DFA M, f_M is in the form that $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = a_j + b_j \times k$, where a_j, b_j and c_i are constants. f_M can be written as $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, such that c_i, r_j, C, R are constants, and $\forall i, c_i < C$, and $\forall j, r_j < R$. We say that a semilinear set in this form is *well-formed*.

In the following, we give the algorithm to construct the automata that accept unary or binary representation of the length of the language accepted by a given string automata. This construction shows that the length constraint of a DFA is a well formed semilinear set, and hence gives a constructive proof of Property 1 and Property 2.

From String Automata to Unary Length Automata It is known that the unary representation of the values of a semilinear set can be uniquely identified by a unary automaton. In the following, we first show how to construct an automaton M_u (over a unary alphabet) from a given string automaton M, such that $L(M_u)$ is the set of unary representations of $\{|w| \mid w \in L(M)\}$. We say M_u is the unary length automaton of M.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F\rangle$, a naive construction of the unary length automaton is $M_u = \langle Q, q_0, B^1, \delta', F\rangle$, where $\delta'(q, 1) = q'$ if $\exists \alpha, \delta(q, \alpha) = q'$. However, M_u constructed this way will be an NFA. The MBDD representations that we use cannot encode NFAs. Instead, we use a construction which combines the projection and determinization steps as follows.

Given a string automaton $M = \langle Q, q_0, B^k, \delta, F\rangle$, we first construct an intermediate automaton $M' = \langle Q, q_0, B^{k+1}, \delta', F\rangle$, where

– $\forall q, q' \in Q$, and both are not sink states, $\delta^{'}(q, \alpha 1) = q'$, if $\delta(q, \alpha) = q'$.

M' is a DFA that accepts the same words as M except that each symbol in the word is appended with '1'. M_u can then be constructed from M' by projecting the first k bits away. This projection is done by iterative determinization and minimization. During determinization, the subset construction is applied on the fly.

From Unary Length Automata to Semilinear Set: Here we describe how to identify the well formed formula of a semilinear set from a unary automaton.

Property 3: A finite deterministic unary automaton $M = \langle Q, q_0, B^0, \delta, F\rangle$ can be in two forms: a linear list of states that starts from the initial state with finite length $\sharp Q$, or a linear list of states that starts from the initial state with finite length, C, and ends in a cycle with finite length, R, where $C + R = \sharp Q$ (i.e., a lasso).

Given a deterministic unary automaton, Q can be labeled such that

– $\sharp Q = n + 1$.
– $\forall 0 \le i < n, \delta(q_i, 1) = q_{i+1}$.

Cycle Case: If $\exists 0 \le m < n, \delta(q_n, 1) = q_m$, the well-formed formula of a unary automaton is $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$, where

– $C = m, R = n - m$.
– $\forall i, \exists q_t, t < m, F(q_t) = +, c_i = t$.
– $\forall j, \exists q_t, t \ge m, F(q_t) = +, r_j = t - m$.

No Cycle Case: Otherwise, the well-formed formula of a unary automaton is $\bigvee_i x = c_i$, where $\forall i, \exists q_t, t \le n, F(q_t) = +, c_i = t$.

From Semilinear Set to Binary Length Automata: We propose a novel construction to derive a DFA M such that $L(M)$ is equal to the set of binary representations (from the least significant bit) of a well-formed semilinear set. We say M is a binary length automaton of the string automaton, the length of whose accepted words forms the semilinear set.

Assume that we are given a well-formed semilinear set $\bigvee_i x = c_i \vee \bigvee_j \exists k.x = C + r_j + R \times k$. Let N be $max(C, R)$. A DFA M that accepts the binary representation of the given semilinear set can be constructed as a tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$, where:

– We assume that there exists a sink state $q_{sink} \in Q$, s.t., $F(q_{sink}) = -, \delta(q_{sink}, 0) = q_{sink}$ and $\delta(q_{sink}, 1) = q_{sink}$, and all transitions that are ignored in this construction are going to q_{sink}.
– Other than the sink state, each state $q \in Q$ is a tuple (t, v, b), where $t \in \{\mathtt{val}, \mathtt{rem}_t, \mathtt{rem}_f\}, v \in \{0, \ldots, N\}$, and $b \in \{\perp\} \cup \{1, \ldots, N\}$. $q.t$ is the type of state q, which indicates the meaning of the value of $q.v$ and $q.b$. While $q.t = \mathtt{val}$, $q.v$ is equal to

the value of the binary word accepted from the initial state to the current state, and $q.b$ is equal to the binary value of the previous bit in the word. We assume $2\perp = 1$. While $q.t = \mathtt{rem}_t$ or $\mathtt{rem}_f$, $q.v$ is equal to the remainder of which the dividend is the value of the binary word accepted from the initial state to the current state and the divisor is R; $q.b$ is the remainder of which the dividend is the binary value of the previous bit in the accepted word and the divisor is R. $q.t = \mathtt{rem}_t$ indicates the value of the binary word accepted from the initial state to the current state is greater or equal to C; $q.t = \mathtt{rem}_f$ indicates the value is less than C.

- q_0 is $(\mathtt{val}, 0, \perp)$.
- $\Sigma = \{0, 1\}$, i.e., B^1.
- $\delta(q, 1) = q'$ if and only if one of the following condition holds:
  - $q.t = \mathtt{val}, q.v + 2q.b \geq C, q'.t = \mathtt{rem}_t, q'.v = (q.v + 2q.b) \mod R$, $q'.b = (2q.b) \mod R$.
  - $q.t = \mathtt{val}, q.v + 2q.b < C, q'.t = \mathtt{val}, q'.v = q.v + 2q.b, q'.b = 2q.b$.
  - $q.t = \mathtt{rem}_t, q'.t = \mathtt{rem}_t, q'.v = (q.v{+}2q.b) \mod R, q'.b = (2q.b) \mod R$.
  - $q.t = \mathtt{rem}_f, q'.t = \mathtt{rem}_t, q'.v = (q.v{+}2q.b) \mod R, q'.b = (2q.b) \mod R$.
- $\delta(q, 0) = q'$ if and only if one of the following condition holds:
  - $q.t = \mathtt{val}, q.v + 2q.b \geq C, q'.t = \mathtt{rem}_f, q'.v = q.v \mod R, q'.b = (2q.b) \mod R$.
  - $q.t = \mathtt{val}, q.v + 2q.b < C, q'.t = \mathtt{val}, q'.v = q.v, q'.b = 2q.b$.
  - $q.t = \mathtt{rem}_t, q'.t = \mathtt{rem}_t, q'.v = q.v, q'.b = (2q.b) \mod R$.
  - $q.t = \mathtt{rem}_f, q'.t = \mathtt{rem}_f, q'.v = q.v, q'.b = (2q.b) \mod R$.
- $F(q) = +$, for all $q \in \{q \mid q.t = \mathtt{val}, \exists i, q.v = c_i\} \cup \{q \mid q.t = \mathtt{rem}_t, \exists j, q.v = (C + r_j) \mod R\}$; $F(q) = -$, o.w.

By definition, $\sharp Q$ is $O(N^2)$. Precisely, in our construction, the number of states that $q.t = \mathtt{val}$ is bounded by C. The number of states that $q.t = \mathtt{rem}_t$ is bounded by R^2 and the number of states that $q.t = \mathtt{rem}_f$ is bounded by $C \times R$. On the other hand, we have observed that after minimization, $\sharp Q$ is often reduced to N.

An Incremental Algorithm Below we give an incremental algorithm to construct a Binary Length Automaton (BLA) M. The construction is achieved by calling the procedure CONSTRUCT_BLA . The input is given as a well-formed semilinear formula, $\bigvee_{0\leq i\leq n} x = c_i \vee \bigvee_{0\leq j\leq m} \exists k.x = \mathrm{C} + \mathrm{r_j} + \mathrm{R} \times \mathrm{k}$. At line 3, we first build Q^b, the set of binary states that will be reached by calling the procedure ADD_BSTATE. A binary state is actually the value of the tuple (t, v, b) as described in the previous section. Each binary state is further associated with an index, a true branch and a false branch, which are used to construct the state graph. Briefly, ADD_BSTATE is a recursive function which incrementally adds the reached binary state if it has never been explored. Initially, the binary state is $(\mathtt{val}, 0, \perp)$. Note that ADD_BSTATE is guaranteed to terminate since the number of binary states are bounded. Upon termination, all reached binary states will have been added to Q^b. For each binary state in Q^b, as line 4 to 9, we iteratively generate a state q and set its transition relation and accepting status, which are used to construct the final automaton at line 10.

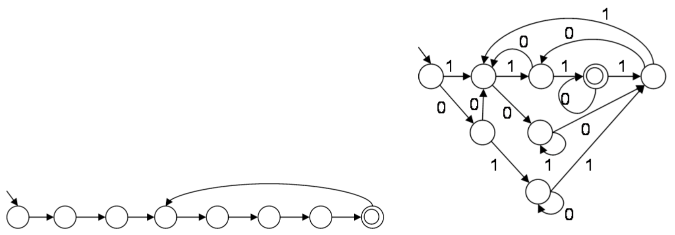
We have implemented the above algorithms using the MONA DFA package. Minimal unary and binary length automata for a regular language are shown Figure 1 where the set recognized by these automata are $\{7 + 5k \mid k \geq 0\}$.

Algorithm 1 ADD_BSTATE(Q, C, R, t, v, b)

```
1: if $\exists q = (t, v, b) \in Q$ then
2:     return $q.index$;
3: else
4:     Create $q = (t, v, b)$;
5:     $q.index = \sharp Q$;
6:     $q.true = -1$;
7:     $q.false = -1$;
8:     Add $q$ to $Q$;
9:     if $t ==$ val $\wedge (v + 2 \times b \geq$C) then
10:        $q.true$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
11:        $q.false$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
12:    else if $t ==$ val $\wedge (v + 2 \times b < C)$ then
13:        $q.true$ =ADD_BSTATE( $Q$, C, R, val, $v + 2 \times b$, $2 \times b$);
14:        $q.false$ =ADD_BSTATE( $Q$, C, R, val, $v$, $2 \times b$);
15:    else if $t ==$ $\texttt{rem}_t$ then
16:        $q.true$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
17:        $q.false$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_t$, $v\%$R, $(2 \times b)\%$R);
18:    else if $t ==$ $\texttt{rem}_f$ then
19:        $q.true$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_t$, $(v + 2 \times b)\%$R, $(2 \times b)\%$R);
20:        $q.false$ =ADD_BSTATE( $Q$, C, R, $\texttt{rem}_f$, $v\%$R, $(2 \times b)\%$R);
21:    end if
22:    return $q.index$;
23: end if
```

Algorithm 2 CONSTRUCT_BLA(C, R, $\mathcal{C} = \{c_1, c_2, \ldots c_n\}$, $\mathcal{R} = \{r_1, r_2, \ldots r_m\}$)

```
1: $Q^b = \emptyset$;
2: $Q = \emptyset$;
3: $init$ =ADD_BSTATE( $Q^b$, C, R, val, 0,$\perp$);
4: for each $q^b \in Q^b$ do
5:     Add $q = q_{q.index}$ to Q;
6:     $\delta(q, 1) = (q^b.true \neq -1 ? q_{q^b.true} : q_{sink})$;
7:     $\delta(q, 0) = (q^b.false \neq -1 ? q_{q^b.false} : q_{sink})$;
8:     $F(q) = ((q^b.t == 0 \wedge \exists c \in \mathcal{C}. q^b.v == c) \vee (q^b.t == 1 \wedge \exists r \in \mathcal{R}. q^b.v ==$
       $(r$+C$)\%$R$) :' +' ?' -')$;
9: end for
10: Construct $M = \langle Q \cup \{q_{sink}\}, q_{init}, B^1, \delta, F \rangle$;
```



(a) Unary (b) Binary

Fig. 1. The Length Automata of $(baaab)^{+}ab$

3 Composite Verification

We first introduce a simple imperative language (the syntax is similar to the one used in [15]) as our target language. This language consists of a set of labeled statements $l : stat$. Labels correspond to instruction addresses. We use s to denote a string variable, i to denote an integer variable, and c to denote a constant. Each $s \in S$ is associated with one auxiliary integer variable, denoted as $s.length$. Let S denote the set of string variables and I denote the set of integer variables, and I_L denote the set of auxiliary variables. A statement can be one of the following:

- A termination statement `halt` or `abort`.
- A string assignment statement $s := strexp$, where $strexp$ is a string expression that can be one of the following:
  - `input`(i) which returns an arbitrary string value up to the length equal to the value of i.
  - a string variable $s \in S$.
  - a regular expression $regexp$ over S.
  - `prefix`(s, i) which returns the prefix of s up to the first c characters where c is equal to the value of i.
  - `suffix`(s, i) which returns the the suffix of s starting from the c^{th} character, where c is equal to the value of i.
  - `concat`(s_1, s_2) that returns the concatenation of the value of s_1 and the value of s_2.
  - `replace`(s_1, s_2, s_3) that returns the result of the following actions: (1) scan the value of s_1 and find the substrings that match to the value of s_2, and (2) replace the matched substrings with the value of s_3.
- An integer assignment statement $i := intexp$, where $intexp$ is an integer expression in the form $\sum_t c_t * i_t$ that returns a value of the linear function $\sum_t c_t * i_t$, where each variable $i_t \in I \cup I_L$.
- A conditional statement `if` $(bexp)$ `goto` l', where $bexp$ is a binary expression (defined below). l' is a program label which indicates the label of the next statement when $bexp$ evaluates to true.
- An assertion statement `assert`$(\bigwedge bexp)$. An assertion holds if $\bigwedge bexp$ evaluates to true. A program is correct if all assertions hold on all executions.

A $bexp$ is either a string or an integer formula defined as follows:

– A string formula can be in two forms: (1) $s \in regexp$, or (2) $s[c_1, c_2] \in regexp$, which specifies that the value of s or the value of the substring (from the c_1^{th} to c_2^{th} character) of s is within a regular language. $s \notin regexp$ is an abbreviation of $s \in regexp'$, where $regexp'$ is the complement set of $regexp$. $s = c$ is an abbreviation of $s \in \{c\}$ and $s \neq c$ is an abbreviation of $s \notin \{c\}$, where c is a constant string.
– An integer formula can be in the form: $\sum_t c_t * i_t \sim c$, where $i_t \in I \cup I_L$ and $\sim \in \{=, <, \leq, \geq, >\}$.

We assume that for each $l : stmt$, $l + 1$ is a valid label if $stmt$ is not a termination statement. For each conditional statement `if` $(bexp)$ `goto` l', l' is a valid label.

Modeling the C Example To analyze normal C programs, one can consider each dereference of a pointer, e.g., $*p$, as a string variable. A sequence value from the address pointed by the pointer is a string value of the string variable. The pointer arithmetic operation, e.g., $p_1 := p_2 + i$, can be considered as a string suffix statement that assigns the suffix of the dereference of p_2 to the dereference of p_1. The previous example can be rewritten using this simple language as follows:

```
strlen(s1){
1: cnt := 0;
2: s2:=s1;
3: if(s2='\0') goto 7;
4: s2:=suffix(s2, 1);
5: cnt := cnt +1;
6: if(s2 != '\0') goto 4;
7: assert(s1.length = cnt);
8: halt;  }
```

3.1 Verification Framework

Assume that $S = \{s_1, \ldots, s_m\}$ and $I = \{i_1, \ldots, i_n\}$ denote the set of string and integer variables in our target program, respectively. In our analysis, each string variable s_k, $1 \leq k \leq m$, is associated with an auxiliary integer variable i_{n+k} as its length $s_k.length$. Hence, we also have the set of auxiliary integer variables $I_L = \{i_{n+1}, \ldots i_{n+m}\}$. A state for each program label consists of a string-automata vector $\boldsymbol{\alpha} = \langle \alpha_1, \ldots, \alpha_m \rangle$ and an $n + m$-track arithmetic automaton a.

Each string variable s_k is associated with the string automaton α_k in $\boldsymbol{\alpha}$, which accepts an over approximation of the set of all possible values that s_k can take at the corresponding program label. Each track of the arithmetic automaton a is a binary encoding starting from the least significant bit of the value of an integer variable (the first n tracks) or the value of the length of a string variable (the last m tracks).

A word accepted by the arithmetic automaton corresponds to a valid valuation for the integer variables and the lengths of string variables at the corresponding program point during the execution of the program. The arithmetic automaton accepts an over approximation of the set of possible words at the corresponding program label. Each word w is an assignment of the integer variables and the lengths of the string variables; and each track of w is actually the value that $i \in I \cup I_L$ can take at the corresponding program label. We use $w[k]$ to denote the k^{th} track of the word w. For $1 \leq k \leq n, w[k]$ is the value of the integer variable i_k. For $n+1 \leq k \leq n+m, w[k]$ is the length of the string variable s_k. We say a string w is the value of a string variable s_k if $w \in L(\alpha_k)$, and $\exists w' \in L(a)$ such that $w'[k]$ is equal to the binary encoding of $|w|$ starting from the least significant bit.

Forward Fixpoint Computation Our analysis is based on a standard forward fixpoint computation on $\boldsymbol{\alpha}$ and a for all program labels. For simplicity, we use $\nu[l]$ to denote $\boldsymbol{\alpha}[l]$ and $a[l]$, where $\boldsymbol{\alpha}[l]$ is the string-automaton vector and $a[l]$ is the arithmetic automaton at the program label l. The algorithm is a standard work-queue algorithm as shown in table 3.

For sequential operations (string/integer assignments), we are continuously computing the post image of $\nu[l]$ against $l : stmt$, and join the result to $\nu[l+1]$ where $l+1$ is the label of the next statement. For branch statement l : $\texttt{if}(bexp)$ $\texttt{goto}$ l', if the intersection of the language of $\nu[l]$ and $bexp$ is not an empty set, we add the result to $\nu[l']$. If the intersection of the language of $\nu[l]$ and the complement set of $bexp$ is not an empty set, we add the result to $\nu[l+1]$. For checking statement l : $\texttt{assert}(\phi)$, if the language of $\nu[l]$ is not included in ϕ, we raise an alarm.

Upon joining the results, we check whether a fixpoint of that program point is reached. If it is not, we update ν at that program point and push its labeled statement into the queue. Since we target infinite state systems, the fixpoint computation may not terminate. We incorporate an automata widening operator, denoted as ∇_A, proposed by Bartzis and Bultan in [3] to accelerate the fixed point computation. $\nu \nabla \nu'$ is implemented as $\alpha_1 \nabla_A \alpha'_1, \ldots, \alpha_m \nabla_A \alpha'_m$ [16] and $a \nabla_A a'$ [3].

Finally, we detail how to compute post and restrict computations, i.e., post$(\nu, stmt)$ and $\nu \wedge bexp$, in the following paragraphs.

Basic Operations Before we detail the algorithms of post and restrict computations, we first define some notations and basic operations to simplify our presentation. We use a to denote the arithmetic automaton, and a_k to denote the one-track arithmetic automaton that accepts the values of the k^{th} track of the arithmetic automaton a. We use α to denote a string automaton and $\boldsymbol{\alpha}$ to denote a vector of string automata. α_k is the k^{th} string automaton of $\boldsymbol{\alpha}$. $\texttt{bla}(\alpha)$ returns the binary length automaton of the string automaton α. The binary length automaton can be considered as an one-track arithmetic automaton. We use α^c, where c is an integer constant, to denote the string automaton which accepts arbitrary words having length equal to c. That is $L(\alpha^c) = \{w \mid w \in \Sigma^*, |w| = c\}$. This notation is also extended to a range $[c_1, c_2]$, where c_1, c_2 are integer constants. We say that $\alpha^{[c_1, c_2]}$ is the string automaton that accepts $\{w \mid w \in \Sigma^*, c_1 \leq |w| \leq c_2\}$.

- Extraction: $a \downharpoonright_k$, returns an one-track arithmetic automaton a_k so that $w \in L(a_k)$ if $\exists w' \in L(a)$ and $w'[k] = w$. a_k is constructed by projecting away all tracks except the k^{th} track of the arithmetic automaton a.
- Projection: $a \upharpoonright_k$, returns a new arithmetic automaton a' which accepts $\{w | w' \in L(a), \forall 1 \leq t \leq m+n, t \neq k, w'[t] = w[t]\}$. a' is constructed by projecting away the track k of the arithmetic automaton a.

Algorithm 3 CompositeAnalysis(l_0)

```
1:  Init($\nu$);
2:  queue $WQ$;
3:  $WQ$.enqueue($l_0 : stmt_0$);
4:  **while** $WQ \neq NULL$ **do**
5:      $e$ := $WQ$.dequeue(); Let $e$ be $l : stmt$;
6:      **if** $stmt$ is sequential operation **then**
7:          $tmp$ := post($\nu[l], stmt$);
8:          $tmp$ := $(tmp \cup \nu[l+1])\nabla\nu[l+1]$;
9:          **if** $tmp \not\subseteq \nu[l+1]$ **then**
10:             $\nu[l+1] := tmp$;
11:             $WQ$.enqueue($l+1$);
12:         **end if**
13:     **end if**
14:     **if** $stmt$ is if $bexp$ goto $l'$ **then**
15:         **if** CheckIntersection($\nu[l], bexp$) **then**
16:             $tmp$ := $\nu[l] \wedge bexp$;
17:             $tmp$ := $(tmp \cup \nu[l'])\nabla\nu[l']$;
18:             **if** $tmp \not\subseteq \nu[l']$ **then**
19:                 $\nu[l'] := tmp$;
20:                 $WQ$.enqueue($l'$);
21:             **end if**
22:         **end if**
23:         **if** CheckIntersection($\nu[l], \neg bexp$) **then**
24:             $tmp$ := $\nu[l] \wedge \neg bexp$;
25:             $tmp$ := $(tmp \cup \nu[l+1])\nabla\nu[l+1]$;
26:             **if** $tmp \not\subseteq \nu[l+1]$ **then**
27:                 $\nu[l+1] := tmp$;
28:                 $WQ$.enqueue($l+1$);
29:             **end if**
30:         **end if**
31:     **end if**
32:     **if** $stmt$ is assert($\phi$) **then**
33:         **if** $\neg$ CheckInclusion($\nu[l], \phi$) **then**
34:             Assertion violated!
35:         **end if**
36:     **end if**
37: **end while**
```

- Composition: $a \circ \alpha_k$, returns a new arithmetic automaton a' so that $L(a') = \{w \mid w \in L(a), w[k] \in L(\mathtt{bla}(\alpha_k))\}$. a' is constructed by intersecting a with an arithmetic automaton that the track k is accepted by the binary length automaton of the string automaton α_k, and other tracks are unrestricted. This composition restricts $L(a)$ to a smaller set where the length of s_k (the value of the track k) is accepted by the binary length automaton of α_k.
- Boundary: $\mathtt{min}(a_k)$ returns the lower bound of the set of integer values whose binary encodings from the least significant bit are accepted by the one-track automaton a_k. $\mathtt{max}(a_k)$ returns the upper bound.

Post Image Recall that there are m string variables and n integer variables. Given $stmt$ and the state ν that consists of $\boldsymbol{\alpha} = \langle \alpha_1, \dots, \alpha_m \rangle$ and the arithmetic automaton a, we want to compute $\boldsymbol{\alpha'} = \langle \alpha'_1, \dots, \alpha'_m \rangle$ and a' as the result of the post image against $stmt$. We assume that the automata that are not specified remain the same. Let $stmt$ be one of the following:

- $s_k := \mathtt{input}(i_p)$. $\alpha'_k := \alpha^{[c_1,c_2]}$, where $c_1 = \mathtt{min}(a_p)$ and $c_2 = \mathtt{max}(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k} := i_p)$.
- $s_{k_1} := s_{k_2}$. $\alpha'_{k_1} := \alpha_{k_2}$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_{n+k_2})$.
- $s_k := regex$. $\alpha'_k := \text{CONSTRUCT}(regexp)$. $a' := a \upharpoonleft_{n+k} \circ \alpha'_k$.
- $s_{k_1} := \mathtt{prefix}(s_{k_2}, i_p)$. $\alpha'_{k_1} := \text{PREFIX}(\alpha_{k_2}, [c_1, c_2])$, where $c_1 = \mathtt{min}(a_p)$ and $c_2 = \mathtt{max}(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_p) \wedge \text{CONSTRUCT}(i_{n+k_2} - i_p \geq 0)$.
- $s_{k_1} := \mathtt{suffix}(s_{k_2}, i_p)$. $\alpha'_{k_1} := \text{SUFFIX}(\alpha_{k_2}, [c_1, c_2])$, where $c_1 = \mathtt{min}(a_p)$ and $c_2 = \mathtt{max}(a_p)$. $a' := \text{CONSTRUCT}(a, i_{n+k_1} := i_p) \wedge \text{CONSTRUCT}(i_{n+k_2} - i_p \geq 0)$.
- $s_k := \mathtt{strcat}(s_{k_1}, s_{k_2})$. $\alpha'_k := \text{CONCAT}(\alpha_{k_1}, \alpha_{k_2})$. $a' := \text{CONSTRUCT}(a, i_{n+k} := i_{n+k_1} + i_{n+k_2})$.
- $s_k := \mathtt{replace}(s_{k_1}, s_{k_2}, s_{k_3})$. $\alpha'_k := \text{REPLACE}(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$. $a' := a \upharpoonleft_{n+k} \wedge a_{tmp}$, where a_{tmp} accepts $\{w \mid w[k] \in L(\mathtt{bla}(\alpha'_k))\}$.
- $i_p := intexp$. $a' := \text{CONSTRUCT}(a, i_p := intexp)$.

Restriction Here we describe the result of $\nu \wedge bexp$, where ν is the state consists of $\boldsymbol{\alpha}$ and a. Let $bexp$ be one of the following:

- $s_k \in regexp$. $\alpha'_k = \alpha_k \wedge \text{CONSTRUCT}(regexp)$. $a' = a \circ \alpha'_k$.
- $s_k[c_1, c_2] \in regexp$. $\alpha'_k = \alpha_k \wedge \alpha_{tmp}$, where α_{tmp} is constructed by $\text{CONCAT}(\text{CONCAT}(\alpha^{[c_1,c_2]}, \text{CONSTRUCT}(regexp)), \alpha^*)$. $a' = a \circ \alpha'_k$.
- $\sum_t c_t * i_t \sim c$. $\forall t > n. \alpha'_t = \alpha_t \wedge \alpha^{[c_1,c_2]}$, where $c_1 = \mathtt{min}(a' \downharpoonleft_t)$ and $c_2 = \mathtt{max}(a' \downharpoonleft_t)$. $a' = a \wedge \text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$.

3.2 Implementation

Automaton Construction In this section, we describe how to construct the corresponding arithmetic and string automata used in our composite analysis. The constructions of arithmetic automata including $\text{CONSTRUCT}(\sum_t c_t * i_t \sim c)$ and $\text{CONSTRUCT}(a, i := \sum_t c_t * i_t)$ are detailed in [2]. The latter returns an arithmetic automaton which accepts the result of the post image computation on a against the integer assignment $i :=$

$\sum_t c_t * i_t + c$. This construction is implemented by quantifier elimination and variable renaming, i.e., $(\exists i, \Phi(a) \wedge i' = \sum_t c_t * i_t)[I'/I]$. For some special cases, the time complexity of this construction is linear to the size of a [2]. The constructions of string automata including CONSTRUCT$(regexp)$, CONCAT$(\alpha_{k_1}, \alpha_{k_2})$, and REPLACE$(\alpha_{k_1}, \alpha_{k_2}, \alpha_{k_3})$ have been detailed in [16]. We describe the implementation of PREFIX$(\alpha, [c_1, c_2])$ and the implementation of SUFFIX$(\alpha, [c_1, c_2])$ below.

Prefix. Formally speaking, α' is a prefix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid w \in \Sigma^{[c_1,c_2]}, \exists w', ww' \in L(\alpha)\}$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F\rangle$ and $[c_1, c_2]$, we first construct $\alpha' = \langle Q, q_0, \Sigma, \delta, F'\rangle$, where $\forall q \in Q, F'(q) =' +'$. α' accepts the prefix of $L(\alpha)$. The next step is restricting its length to the range $[c_1, c_2]$. PREFIX$(\alpha, [c_1, c_2])$ returns the the result of the intersection of α' and $\alpha^{[c_1,c_2]}$, which is exactly the prefix-DFA of α regarding to the range $[c_1, c_2]$.

Suffix. Formally speaking, α' is a suffix-DFA of α regarding to the range $[c_1, c_2]$, if $L(\alpha') = \{w \mid \exists w' \in \Sigma^{[c_1,c_2]}, w'w \in L(\alpha)\}$. We first introduce the function REACH$(\alpha, [c_1, c_2])$. REACH$(\alpha, [c_1, c_2])$ returns the set of all $[c_1, c_2]$-reachable states. We say a state is $[c_1, c_2]$-reachable if it is reachable from the initial state by k steps and $c_1 \leq k \leq c_2$. Given $\alpha = \langle Q, q_0, \Sigma, \delta, F\rangle$ and $[c_1, c_2]$, we first compute $R =$ REACH$(\alpha, [c_1, c_2])$ via a breadth-first search. We then construct the following finite automaton $\alpha' = \langle Q', q_0', \Sigma, \delta', F'\rangle$, where

- $Q' = Q \cup \{q_0'\}$
- $\forall q, q' \in Q, \delta'(q, \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $\forall q \in R, q' \in Q, \delta'(q_0', \alpha) = q'$, if $\delta(q, \alpha) = q'$.
- $F'(q_0) =' +'$, if $\exists q \in R, F(q) =' +'$.
- $\forall q \in Q, F'(q) = F(q)$.

Note that α' constructed by the above construction may be a nondeterministic finite automaton. We add auxiliary bits to resolve nondeterminism as proposed in [16]. SUFFIX$(\alpha, [c_1, c_2])$ returns the result of the minimization and determinization of α'.

Boundary Below we describe how to identify the boundary of a one-track arithmetic automaton, which accepts the binary encodings of a set of integer values from the least significant bit.

Property 4: For an one-track minimized DFA $a = \langle Q, q_0, B^1, \delta, F\rangle$: $\forall q, q' \in Q$, if $\delta(q, 0) = q'$, then $F(q) = F(q')$.

Property 4 states that transitions labelled by 0 cannot change accepting status, which holds due to the fact that by definition, the arithmetic automaton accepts a word and any number of 0 in its higher significant bits. It follows that for any accepted integer value (except 0), the word from the least significant bit up to the most non-zero significant bit of its binary encoding forms a unique path (ended by 1) from the initial state to an accepting state. Furthermore, an accepted non-zero minimal integer value forms the shortest path from the initial state to an accepting state. On the other hand, if there exists an accepted non-zero maximal integer value, the maximal value forms the longest loop-free path from the initial state to an accepting state. Note that if there exists an accepted path containing a loop, a accepts an infinite set and the maximal value does not exist. In this case, we use *inf* to denote the maximal value.

For $\texttt{min}(a)$ and $\texttt{max}(a)$, we have implemented two functions $\text{MIN}(a)$ and $\text{MAX}(a)$. Let m_s be the length of the shortest path that ends with 1 and m_l be the length of the longest loop-free path that ends with 1. Both m_s and m_l can be determined by a breadth first search up to $\sharp Q$ steps. In our implementation, we first check whether a accepts any non-zero integer value. If this is the case, $\text{MIN}(a)$ returns 2^{m_s-1}, which is a lower bound for the shortest path. If there exists a path containing a loop, $\text{MAX}(a)$ returns *inf*. Otherwise $\text{MAX}(a)$ returns $2^{m_l+1}-1$, which is an upper bound for the longest path. Note that our implementation is a conservative approximation. These bounds can be tightened by tracing the values along paths.

4 Experiments

We experimented with our composite analysis tool on a number of test cases extracted from C string library, buffer overflow benchmarks [10] and web vulnerability benchmarks [16]. These test cases are rather small but involve pointer arithmetic, string content constraints, length constraints, loops, and replacement operations. We manually convert them to our simple imperative language.

For `int strlen(char *`s`)`, we verify the invariant that the return value is equal to the length of the input string. For `char *strrchr(char *`s`, int `c`)`, we verify whether the language accepted by the return string is included in $\{cx \mid x \in \Sigma^*\} \cup \{\epsilon\}$ upon reaching the fixpoint. For buffer overflow benchmarks, we check whether the identified memory may overflow its buffer upon reaching the fixpoint for both buggy (*bad*) and modified (*ok*) cases. For web vulnerability benchmarks, we check whether the identified sensitive function may take any attack string as its input before (*bad*) and after (*ok*) inserting limit constraints and sanitization routines. If it does not, the sensitive function is SQL attack free with respect to the attack pattern Σ^*`<script`Σ^*. Limit constraints are written as new statements that limit the length of string variables using a `$limit` variable. Table 1 shows that our composite analysis works well in these test cases in terms of both accuracy and performance. As a final remark, for web vulnerability benchmarks, one may restrict limit constraints, e.g., set `$limit` less than 7, to prevent the specified attacks without adding/modifying sanitization routines. In this case, pure string analysis [16] will raise false alarms.

Test case (*bad*/*ok*)	Result	Time (s)	Memory (kb)
int strlen(char *s)	T	0.037	522
char *strrchr(char *s, int c)	T	0.011	360
gxine (CVE-2007-0406)	F/T	0.014/0.018	216/252
samba (CVE-2007-0453)	F/T	0.015/0.021	218/252
MyEasyMarket-4.1 (trans.php:218)	F/T	0.032/0.041	704/712
PBLguestbook-1.32 (pblguestbook.php:1210)	F/T	0.021/0.022	496/662
BloggIT 1.0 (admin.php:27)	F/T	0.719/0.721	5857/7067

Table 1. Preliminary experimental results. T: buffer overflow free or SQL attack free

5 Conclusion

We presented an automata-based approach for symbolic verification of infinite-state systems with unbounded string and integer variables. Our approach combines string and size analyses and is able to verify properties that cannot be verified with either analysis alone. We demonstrated the effectiveness of our approach on several examples.

References

1. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the Symposium on Security and Privacy*, 2008.
2. Constantinos Bartzis and Tevfik Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.
3. Constantinos Bartzis and Tevfik Bultan. Widening arithmetic automata. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 321–333, 2004.
4. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.
5. Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. *SIGPLAN Not.*, 38(5):155–167, 2003.
6. Xiang Fu, Xin Lu, Boris Peltsverger, Shijun Chen, Kai Qian, and Lixin Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.
7. Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, 2003.
8. Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *35th ACM Symposium on Principles of Programming Languages*, pages 235–246. ACM, January 2008.
9. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 339–348, 2008.
10. Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392, 2007.
11. Daryl Shannon, Sukant Hajra, Alison Lee, Daiqian Zhan, and Sarfraz Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.
12. David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *In Network and Distributed System Security Symposium*, pages 3–17, 2000.
13. Pierre Wolper and Bernard Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS*, pages 1–19, 2000.
14. Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.
15. Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *ISSTA '08: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2008.
16. Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra. Symbolic string verification: An automata-based approach. In *15th International SPIN Workshop on Model Checking of Software*, 2008.