Relational String Verification Using Multi-track Automata *

Fang Yu[1], Tevfik Bultan[2], and Oscar H. Ibarra[2]

[1] National Chengchi University, Taipei, Taiwan
yuf@nccu.edu.tw
[2] University of California, Santa Barbara, CA, USA
{bultan, ibarra}@cs.ucsb.edu

Abstract. Verification of string manipulation operations is a crucial problem in computer security. In this paper, we present a new relational string verification technique based on multi-track automata. Our approach is capable of verifying properties that depend on relations among string variables. This enables us to prove that vulnerabilities that result from improper string manipulation do not exist in a given program. Our main contributions in this paper can be summarized as follows: (1) We formally characterize the string verification problem as the reachability analysis of *string systems* and show decidability/undecidability results for several string analysis problems. (2) We develop a sound symbolic analysis technique for string verification that over-approximates the reachable states of a given string system using multi-track automata and summarization. (3) We evaluate the presented techniques with respect to several string analysis benchmarks extracted from real web applications.

1 Introduction

The most important Web application vulnerabilities are due to inadequate manipulation of string variables [11]. In this paper we investigate the *string verification problem:* Given a program that manipulates strings, we want to verify assertions about string variables. For example, we may want to check that at a certain program point a string variable cannot contain a specific set of characters. This type of checks can be used to prevent SQL injection attacks where a malicious user includes special characters in the input string to inject unintended commands to the queries that the Web application constructs (using the input provided by the user) and sends to a backend database. As another example, we may want to check that at a certain program point a string variable should be prefix or suffix of another string variable. This type of checks can be used to prevent Malicious File Execution (MFE) attacks where Web application developers concatenate potentially hostile user input with file functions that lead to inclusion or execution of untrusted files by the Web server.

We formalize the string verification problem as reachability analysis of *string systems* (Section 2). After demonstrating that the string analysis problem is undecidable

* This work is supported by NSC grant 99-2218-E-004-002-MY3, and NSF grants CCF-0916112, CCF-0716095, and CCF-0524136.

in general, we present and implement a forward symbolic reachability analysis technique that computes an over-approximation of the reachable states of a string system using widening and summarization (Section 4). We use multi-track deterministic finite automata (DFAs) as a symbolic representation to encode the set of possible values that string variables can take at a given program point. Unlike prior string analysis techniques, our analysis is *relational*, i.e., it is able to keep track of the relationships among the string variables, improving the precision of the string analysis and enabling verification of invariants such as $X_1 = X_2$ where X_1 and X_2 are string variables. We develop the precise construction of multi-track DFAs for linear word equations, such as $c_1 X_1 c_2 = c'_1 X_2 c'_2$ and show that non-linear word equations (such as $X_1 = X_2 X_3$) cannot be characterized precisely as a multi-track DFA (Section 3). We propose a regular approximation for non-linear equations and show how these constructions can be used to compute the post-condition of branch conditions and assignment statements that involve concatenation. We use summarization for inter-procedural analysis by generating a multi-track automaton (transducer) characterizing the relationship between the input parameters and the return values of each procedure (Section 4). To be able to use procedure summaries during our reachability analysis we *align* multi-track automata so that normalized automata are closed under intersection. We implemented these algorithms using the MONA automata package [5] and analyzed several PHP programs demonstrating the effectiveness of our string analysis techniques (Section 5).

Related Work. The use of automata as a symbolic representation for verification has been investigated in other contexts [4]. In this paper, we focus on verification of string manipulation operations, which is essential to detect and prevent crucial web vulnerabilities. Due to its importance in security, string analysis has been widely studied. One influential approach has been grammar-based string analysis that statically computes an over-approximation of the values of string expressions in Java programs [6] which has also been used to check for various types of errors in Web applications [8, 9, 13]. In [9, 13], multi-track DFAs, known as *transducers*, are used to model replacement operations. There are also several recent string analysis tools that use symbolic string analysis based on DFA encodings [7, 12, 15, 16]. Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs [7, 12]. In our earlier work, we have used a DFA based symbolic reachability analysis to verify the correctness of string sanitization operations in PHP programs [14–16]. Unlike the approach we proposed in this paper, all of the results mentioned above use single track DFA and encode the reachable configurations of each string variable separately. Our multi-track automata encoding not only improves the precision of the string analysis but also enables verification of properties that cannot be verified with the previous approaches. We have also investigated the boundary of decidability for the string verification problem. Bjørner et al. [2] show the undecidability result with replacement operation. In this paper we consider only concatenation and show that string verification problem is undecidable even for deterministic string systems with only three unary string variables and non-deterministic string systems with only two string variables if the comparison of two variables are allowed.

prog ::= *decl** *func**
decl ::= `decl` *id*$^+$;
func ::= *id* (*id**) `begin` *decl** *lstmt*$^+$ `end`
lstmt ::= *l*:*stmt*
stmt ::= *seqstmt* | `if` *exp* `then goto` *l*; | `goto` *L*;   where *L* is a set of labels
    | `input` *id*; | `output` *exp*; | `assert` *exp*;
seqstmt::=*id* := *sexp*; | *id* := `call` *id* ($sexp^*$);
exp ::= *bexp* | *exp* ∧ *exp* | ¬ *exp*
bexp ::= *atom* = *sexp*
sexp ::= *sexp*.*atom* | *atom* | *suffix*(*id*) | *prefix*(*id*)
atom ::= *id* | *c*,   where *c* is a string constant

Fig. 1. The syntax of string systems

2 String Systems

We define the syntax of string systems in Figure 1. We only consider string variables and hence variable declarations need not specify a type. All statements are labeled. We only consider one string operation (concatenation) in our formal model; however, our symbolic string analysis techniques can be extended to handle complex string operations (such as replacement [15]). Function calls use call-by-value parameter passing. We allow goto statements to be non-deterministic (if a goto statement has multiple target labels, then one of them is chosen non-deterministically). If a string system contains a non-deterministic goto statement it is called a non-deterministic string system, otherwise, it is called a deterministic string system.

In order to identify different classes of string systems we will use the following notation. Let $S(X_1, X_2, \ldots, X_n)$ denote a string system with string variables X_1, X_2, ..., X_n and a finite set of labeled instructions. We will use the letters D and N to denote deterministic and non-deterministic string systems, respectively. We will use the letters B and U to denote if the alphabet used by the string variables is the binary alphabet $\{a, b\}$ or the unary alphabet $\{a\}$, respectively. We will use K to denote an alphabet of arbitrary size. For example, $DUS(X_1, X_2, X_3)$ denotes a deterministic string system with three variables and the unary alphabet whereas $NBS(X_1, X_2)$ denotes a nondeterministic string system with two variables and the binary alphabet. We will denote the set of assignment instructions allowed in a string system as a superscript and the set of expressions involved in conditional branch instructions as subscript. Hence, $DUS(X_1, X_2, X_3)^{X_i:=X_i a}_{X_1=X_3, X_2=X_3}$ denotes a deterministic string system with three variables X_1, X_2, and X_3, and the unary alphabet $\{a\}$ where the assignment instructions are of the form $X_1 := X_1 a$, $X_2 := X_2 a$, or $X_3 := X_3 a$ (i.e., we only allow concatenation of one symbol to a string variable in each assignment instruction) and the conditional branch instructions can only be of the form: **if** $X_3 = X_1$ **goto** L or **if** $X_3 = X_2$ **goto** L (i.e., we only allow equality checks and do not allow comparison of X_1 and X_2.)

The *halting problem* for string systems is the problem of deciding, given a string system S, where initially the string variables are initialized to the null string, ϵ, whether S will halt on some execution. More generally, the *reachability problem for string systems* (which need not halt) is the problem of deciding, given a string system S and a

configuration C (i.e., the instruction label and the values of the variables), whether at some point during a computation, C will be reached. Note that we define the halting and the reachability conditions using existential quantification over the execution paths, i.e., the halting and the reachability conditions hold if there exists an execution path that halts or reaches the target configuration, respectively. Hence, if the halting problem is undecidable, then the reachability problem is undecidable.

Lemma 1. *The halting problem for* $DUS(X_1, X_2, X_3)^{X_i:=X_i a}_{X_1=X_3, X_2=X_3}$ *is undecidable.*

Proof. It is well-known that the halting problem for two-counter machines, where initially both counters are 0, is undecidable [10]. During the execution of a counter machine, at each step, a counter can be incremented by 1, decremented by 1, and tested for zero. The counters can only assume nonnegative values.

We will show that a two-counter machine M can be simulated with a string system $S(X_1, X_2, X_3)$ in $DUS(X_1, X_2, X_3)^{X_i:=X_i a}_{X_1=X_3, X_2=X_3}$. The states of M can be represented as labels in the string system S. The states where the counter-machine M halts will be represented with the halt instruction in string system S. We will use the lengths of the strings X_1, X_2 and X_3 to simulate the values of the counters C_1 and C_2. The value of C_1 will be simulated by $|X_1| - |X_3|$, and the value of C_2 will be simulated by $|X_2| - |X_3|$.

The counter machine M starts from the initial configuration $(q_0, 0, 0)$ where q_0 denotes the initial state and the two integer values represent the initial values of counters C_1 and C_2, respectively. The initial configuration of the string system S will be $(q_0, \epsilon, \epsilon, \epsilon)$ where q_0 is the label of the first instruction, and the strings $\epsilon, \epsilon, \epsilon$ are the initial values of the string variables X_1, X_2 and X_3, respectively. The instructions of the counter-machine C will be simulated as follows (where each statement is followed by a goto statement that transitions to the next state or instruction):

Counter machine	String system
inc C_1	$X_1 := X_1 a$
inc C_2	$X_2 := X_2 a$
dec C_1	$X_2 := X_2 a; X_3 := X_3 a$
dec C_2	$X_1 := X_1 a; X_3 := X_3 a$
if $(C_1 = 0)$	**if** $(X_1 = X_3)$
if $(C_2 = 0)$	**if** $(X_2 = X_3)$

Note that although this transformation will allow the simulated counter values to possibly take negative values, this can be fixed by adding a conditional branch instruction before each decrement that checks that the simulated counter value is not zero before the instructions simulating the decrement instruction is executed. The string system S constructed from M based on these rules will simulate M. Hence, halting problem is undecidable for the string systems in $DUS(X_1, X_2, X_3)^{X_i:=X_i a}_{X_1=X_3, X_2=X_3}$.

In the proof of the theorem above, comparisons are only between X_3 and X_1 and between X_3 and X_2. Suppose we have 4 variables X_1, X_2, X_3, X_4, and we only allow comparisons between X_1 and X_3 and between X_2 and X_4. Thus, comparisons are only between the same variables. We denote this class of string systems as: $DUS(X_1, X_2, X_3, X_4)^{X_i:=X_i a}_{X_1=X_3, X_2=X_4}$.

Lemma 2. *The halting problem for $DUS(X_1, X_2, X_3, X_4)^{X_i:=X_ia}_{X_1=X_3,X_2=X_4}$ is undecidable.*

Proof. We can modify the proof of Theorem **??** so that X_3 and X_4 are essentially identical copies (i.e., they are incremented simultaneously). The comparison between X_3 and X_1 (respectively, between X_3 and X_2) in the construction above is now reduced to comparison between X_3 and X_1 (respectively, between X_4 and X_2). It follows the halting problem for these programs is also undecidable. □

Lemma 3. *The halting problem for $NBS(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ is undecidable.*

Proof. Given an instance (C, D) of PCP, where $C = (c_1, ..., c_n)$ and $D = (d_1, ..., d_n)$, define constant strings $\{c_1, ..., c_n, d_1, ..., d_n\}$, where c_i, d_i are non-null strings over alphabet $\{a, b\}$, we construct a string system S in $NBS(X_1, X_2)^{X_i:=X_ic}_{X_1=X_2}$ as follows:
0: goto 1 or 2 or ... or n
1: $X_1 := X_1c_1$ and $X_2 := X_2d_1$; goto 0 or n+1
2: $X_1 := X_1c_2$ and $X_2 := X_2d_2$; goto 0 or n+1
...
n: $X_1 := X_1c_n$ and $X_2 := X_2d_n$; goto 0 or n+1
n+1: if $X_1 = X_2$ goto n+2 else go to 1
n+2: halt
Clearly, there is a computation that will reach the halt instruction if and only if the PCP instance (C, D) has a solution. The theorem follows.

Lemma 4. *The halting problem for $DKS(X_1, X_2)^{X_i:=X_ia,X_i:=aX_i}_{X_1=X_2}$ is decidable.*

Proof. Let S be a string system in $DKS(X_1, X_2)^{X_i:=X_ia,X_i:=aX_i}_{X_1=X_2}$ and k be its length (i.e., number of instructions), including the assignments, and the conditional and unconditional branch statements.

Label the instructions of S by $1, ..., k$. We can think of each assignment, $i :\ A$ as equivalent to the instruction, $i :\ A;$ **goto** $i + 1$. Hence, every instruction except the halt instruction and the **if** statements has a **goto**.

By an "execution of a positive **if** statement", we mean that when the **if** statement is executed, $X_1 = X_2$.

During the computation of S, if it is not in an infinite loop, then the interval (i.e., number of steps) between the executions of any two consecutive positive **if** statements is at most k. The reason for this is that during the interval, S executes only **goto**'s and assignment statements with **goto**'s (note that a non-positive **if** statement leads directly to the instruction following the **if**). Hence, the number of steps would be at most k, since there are at most k **goto**'s and assignments with **goto**'s.

Now, an execution of a positive **if** statement leads to a **goto** label, and there are at most k different labels. It follows that if S is not in an infinite loop, it cannot run more than $k.k = k^2$ steps.

The above theorem can be generalized. Consider the class of deterministic multi-variable programs, where the variables are $X_1, X_2, ..., X_k$. Assignments can be of the form: $X_i := X_ia$ or $X_i := aX_i$ where $1 \leq i \leq k$ and a is a symbol over some (not-necessarily) unary alphabet Σ. Conditional branch instructions can contain expressions

```
1: input X1;
2: input X2;
3: if (X1 = X2) goto 6;
4: X1:=X2.c;
5: goto 7;
6: X1:=X1.c;
7: assert (X1 = X2.c);
```

Fig. 2. An example with a branch statement

```
1: X1 := a;
2: X2 := a;
3: X1 := X1.b;
4: X2 := X2.b;
5: assert (X1=X2);
6: goto 3;
```

Fig. 3. An example with a loop

of the form $X_1 = X_2$ or $c\ \theta\ X_i$ where c is a constant string and θ is $=$, $prefix$ or $suffix$. Note that only in a conditional branch statement we only allow equality check between the two variables X_1 and X_2, i.e., no other variables can be compared for equality. Label all the instructions in the program with $1, 2, ..., n$, where n is the number of instructions. The program may or may not have a halt instruction. Consider the following reachability problem: Given configuration $(L, x_1, x_2, ..., x_k\ )$ (where L is a label in the program, and the x_i's are strings representing the values of the variables X_i's), will the program starting from the initial configuration $(1, \epsilon, \epsilon, .., \epsilon)$ ever enter configuration $(L, x_1, x_2, ..., x_k)$? We can show the following:

Lemma 5. *The reachability problem for* $DKS(X_1, X_2, \ldots, X_k)^{X_i:=X_ia, X_i:=aX_i}_{X_1=X_2, c=X_i, c=prefix(X_i), c=suffix(X_i)}$ *is decidable.*

Now let us consider the set of multi-variable (i.e., not restricted to two or three variables) nondeterministic string systems where the only assignments are of the form $X_i := dX_ic$, where d, c are constant strings over some alphabet (not necessarily unary or binary), and the **if** statements only involve comparing a variable with a constant string, i.e., of the form $c\ \theta X_i$ where c is a constant string, and θ can be $=$, $prefix$ or $suffix$.

Lemma 6. *The reachability problem for* $NKS(X_1, X_2, \ldots, X_k)^{X_i:=dX_ic}_{c=X_i, c=prefix(X_i), c=suffix(X_i)}$ *is decidable.*

Proof. Let F be a program in $\mathcal{H}_{\mathcal{N}}$ with variables $X_1, ..., X_k$. Assume for now $k = 2$. We construct a nondeterministic 4-tape finite automaton M, wheres tapes T_1, T_2, T_3, T_4 are used to simulate the concatenations to variables X_1 and X_2 as in the proof of the above corollary. Let m be the maximum length of all constant strings appearing in the program F. M simulates the computation of F by reading symbols on the four tapes, as in proof of the corollary. It also keeps track of bounded prefixes of strings in the the variables, which change dynamically. During the simulation of F, we will be concatenating constant strings to the variables. The first time a variable has a string of length $\geq m$, we can stop updating the prefixes without affecting the simulation of the "if" statements. It follows that the amount of memory we need to keep track of the prefixes is finite and depends only on the specifications of F. Hence the number of states of M is finite. Since emptiness for multi-tape finite automata is decidable, it follows that reachability is decidable. The construction when $k \geq 2$ is obvious (now M will have $2k$ tapes.) $\square$

Examples of String Systems Consider the string system in Fig. 2. Existing automata-based string analysis techniques are not able to prove the assertion at the end of this program segment since they use single-track automata. Consider a symbolic analysis technique that uses one automaton for each variable at each program point to represent the set of values that the variables can take at that program point. Using this symbolic representation we can do a forward fixpoint computation to compute the reachable state space of the program. For example, the automaton for variable X_1 at the beginning of statement 2, call it $M_{X_1,2}$, will recognize the set $L(M_{X_1,2}) = \Sigma^*$ to indicate that the input can be any string. Similarly, the automaton for variable X_2 at the beginning of statement 3, call it $M_{X_2,3}$, will recognize the set $L(M_{X_2,3}) = \Sigma^*$. The question is how to handle the branch condition in statement 3. If we are using single track automata, all we can do at the beginning of statement 6 is the following: $L(M_{X_1,6}) = L(M_{X_2,6}) = L(M_{X_1,3}) \cap L(M_{X_2,3})$. The situation with the else branch is even worse. All we can do at line 4 is to set $L(M_{X_1,4}) = L(M_{X_1,3})$ and $L(M_{X_2,4}) = L(M_{X_2,3})$. Both branches will result in $L(M_{X_1,7}) = \Sigma^*.c$ and $L(M_{X_2,7}) = \Sigma^*$, which is clearly not strong enough to prove the assertion.

Using the techniques presented in this paper, we can verify the assertion in the above program. In our approach, we use a single multi-track automaton for each program point, where each track of the automaton corresponds to one string variable. For the above example, the multi-track automaton at the beginning of statement 3 will accept any pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$. However, the multi-track automaton at the beginning of statement 6 will only accept pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2$. We compute the post-condition $(\exists X_1.(X_1 = X_2) \wedge (X_1' = X_1.c))[X_1/X_1']$ and generate the multi-track automaton that only accepts pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2.c$. Similarly, the multi-track automaton at the beginning of statement 4 will only accept pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 \neq X_2$, and after the assignment, we will generate the multi-track automaton that only accepts pairs of strings X_1, X_2 where $X_1, X_2 \in \Sigma^*$ and $X_1 = X_2.c$. Hence, we are able to prove the assertion in statement 7.

Now, consider the string system in Fig. 3. which contains an infinite loop. If we try to compute the reachable configurations of this program by iteratively adding configurations that can be reached after a single step of execution, our analysis will never terminate. We incorporate a widening operator to accelerate our symbolic reachability computation and compute an over-approximation of the fixpoint that characterizes the reachable configurations. Note that the assertion in this program segment is not explicitly established, i.e., there is no assignment, such as $X_1 := X_2$, or branch condition, such as $X_1 = X_2$, that implies that this assertion holds. Also, the assertion specifies the equality among two string variables. Analysis techniques that encode reachable states using multiple single-track DFAs will raise a false alarm, since, individually, X_1 can be abb and X_2 can be ab at program point 5, but they cannot take these values at the same time. It is not possible to express such a constraint using single-track automata.

For this example, our multi-track automata based string analysis technique terminates in three iterations and computes the precise result. The multi-track automaton that characterizes the values of string variables X_1 and X_2 at program point 5, call it M_5, recognizes the language: $L(M_5) = (a, a)(b, b)^+$. Since $L(M_5) \subseteq L(X_1 = X_2)$,

we conclude that the assertion holds. Although in this case the result of our analysis is precise, it is not guaranteed to be precise in general. However, it is guaranteed to be an over-approximation of the reachable configurations. Hence, our analysis is sound and if we conclude that an assertion holds, the assertion is guaranteed to hold for every program execution.

3 Regular Approximation of Word Equations

Our string analysis is based on the following observations: (1) The transitions and the configurations of a string system can be symbolically represented using word equations with existential quantification, (2) word equations can be represented/approximated using multi-track DFAs, which are closed under intersection, union, complement, and projection, and (3) the operations required during reachability analysis (such as equivalence checking) can be computed on DFAs.

Multi-track DFAs A multi-track DFA is a DFA but over the alphabet that consists of many tracks. An n-track alphabet is defined as $(\Sigma \cup \{\lambda\})^n$, where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ $(1 \leq i \leq n)$ to denote the ith track of $w \in (\Sigma \cup \{\lambda\})^n$. An *aligned* multi-track DFA is a multi-track DFA where all tracks are left justified (i.e., λ's are right justified). That is, if w is accepted by an aligned n-track DFA M, then for $1 \leq i \leq n$, $w[i] \in \Sigma^*\lambda^*$. We also use $\hat{w}[i] \in \Sigma^*$ to denote the longest λ-free prefix of $w[i]$. It is clear that aligned multi-track DFA languages are closed under intersection, union, and homomorphism. Let M_u be the aligned n-track DFA that accepts the (aligned) universe, i.e., $\{w \mid \forall i.w[i] \in \Sigma^*\lambda^*\}$. The complement of the language accepted by an aligned n-track DFA M is defined by *complement modulo alignment*, i.e., the intersection of the complement of $L(M)$ with $L(M_u)$. For the following descriptions, a multi-track DFA is an aligned multi-track DFA unless we explicitly state otherwise.

Word Equations A word equation is an equality relation of two words that concatenate variables from a finite set $\mathbf{X}$ and words from a finite set of constants $\mathcal{C}$. The general form of word equations is defined as $v_1 \ldots v_n = v'_1 \ldots v'_m$, where $\forall i, v_i, v'_i \in \mathbf{X} \cup \mathcal{C}$. The following theorem identifies the basic forms of word equations. For example, a word equation $f : X_1 = X_2 d X_3 X_4$ is equivalent to $\exists X_{k_1}, X_{k_2}.X_1 = X_2 X_{k_1} \wedge X_{k_1} = dX_{k_2} \wedge X_{k_2} = X_3 X_4$.

Theorem 1. *Word equations and Boolean combinations ($\neg$, $\wedge$ and $\vee$) of these equations can be expressed using equations of the form $X_1 = X_2 c$, $X_1 = cX_2$, $c = X_1 X_2$, $X_1 = X_2 X_3$, Boolean combinations of such equations and existential quantification.*

Let f be a word equation over $\mathbf{X}= \{X_1, X_2, \ldots, X_n\}$ and $f[c/X]$ denote a new equation where X is replaced with c for all X that appears in f. We say that an n-track DFA M *under-approximates* f if for all $w \in L(M)$, $f[\hat{w}[1]/X_1, \ldots, \hat{w}[n]/X_n]$ holds. We say that an n-track DFA M *over-approximates* f if for any $s_1, \ldots, s_n \in \Sigma^*$ where $f[s_1/X_1, \ldots, s_n/X_n]$ holds, there exists $w \in L(M)$ such that for all $1 \leq i \leq n, \hat{w}[i] = s_i$. We call M *precise* with respect to f if M both under-approximates and over-approximates f.

Definition 1. *A word equation f is regularly expressible if and only if there exists a multi-track DFA M such that M is precise with respect to f.*

Theorem 2. *1. $X_1 = X_2c$, $X_1 = cX_2$, and $c = X_1X_2$ are regularly expressible, as well as their Boolean combinations.*
2. $X_1 = cX_2$ is regularly expressible but the corresponding M has exponential number of states in the length of c.
3. $X_1 = X_2X_3$ is not regularly expressible.

We are able to compute multi-track DFAs that are precise with respect to word equations: $X_1 = X_2c$, $X_1 = cX_2$, and $c = X_1X_2$. Since $X_1 = X_2X_3$ is not regularly expressible, below, we describe how to compute DFAs that approximate such non-linear word equations. Using the DFA constructions for these four basic forms we can construct multi-track DFAs for all word equations and their Boolean combinations (if the word equation contains a non-linear term then the constructed DFA will approximate the equation, otherwise it will be precise). The Boolean operations conjunction, disjunction and negation can be handled with intersection, union, and complement modulo alignment of the multi-track DFAs, respectively. Existential quantification on the other hand, can be handled using homomorphism, where given a word equation f and a multi-track automaton M such that M is precise with respect to f, then the multi-track automaton $M \downharpoonright_i$ is precise with respect to $\exists X_i.f$ where $M \downharpoonright_i$ denotes the result of erasing the ith track (by homomorphism) of M.

Construction of $X_1 = X_2X_3$ Since Theorem 2 shows that $X_1 = X_2X_3$ is not regularly expressible, it is necessary to construct a conservative (*over* or *under*) approximation of $X_1 = X_2X_3$. We first propose an *over* approximation construction for $X_1 = X_2X_3$. Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1\rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2\rangle$, and $M_3 = \langle Q_3, \Sigma, \delta_3, I_3, F_3\rangle$ accept values of X_1, X_2, and X_3, respectively. $M = \langle Q, (\Sigma \cup \{\lambda\})^3, \delta, I, F\rangle$ is constructed as follows.

- $Q \subseteq Q_1 \times Q_2 \times Q_3 \times Q_3$,
- $I = (I_1, I_2, I_3, I_3)$,
- $\forall a, b \in \Sigma, \delta((r, p, s, s'), (a, a, b)) = (\delta_1(r, a), \delta_2(p, a), \delta_3(s, b), s')$,
- $\forall a, b \in \Sigma, p \in F_2, s \notin F_3, \delta((r, p, s, s'), (a, \lambda, b)) = (\delta_1(r, a), p, \delta_3(s, b), \delta_3(s', a))$,
- $\forall a \in \Sigma, p \in F_2, s \in F_3, \delta((r, p, s, s'), (a, \lambda, \lambda)) = (\delta_1(r, a), p, s, \delta_3(s', a))$,
- $\forall a \in \Sigma, p \notin F_2, s \in F_3, \delta((r, p, s, s'), (a, a, \lambda)) = (\delta_1(r, a), \delta_2(p, a), s, s')$,
- $F = \{(r, p, s, s') \mid r \in F_1, p \in F_2, s \in F_3, s' \in F_3\}$.

The intuition is as follows: M traces M_1, M_2 and M_3 on the first, second and third tracks, respectively, and makes sure that the first and second tracks match each other. After reaching an accepting state in M_2, M enforces the second track to be λ and starts to trace M_3 on the first track to ensure the rest (suffix) is accepted by M_3. $|Q|$ is $O(|Q_1| \times |Q_2| \times |Q_3| + |Q_1| \times |Q_3| \times |Q_3|)$. For all $w \in L(M)$, the following hold:

- $\hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2), \hat{w}[3] \in L(M_3)$,
- $\hat{w}[1] = \hat{w}[2]w'$ and $w' \in L(M_3)$,

Note that w' may not be equal to $\hat{w}[3]$, i.e., there exists $w \in L(M)$, $\hat{w}[1] \neq \hat{w}[2]\hat{w}[3]$, and hence M is not precise with respect to $X_1 = X_2X_3$. On the other hand, for any w such that $\hat{w}[1] = \hat{w}[2]\hat{w}[3]$, we have $w \in L(M)$, hence M is a regular *over*-approximation of $X_1 = X_2X_3$.

Below, we describe how to construct a regular *under*-approximation of $X_1 = X_2X_3$ (which is necessary for conservative approximation of its complement set). We use the idea that if $L(M_2)$ is a finite set language, one can construct the DFA M that satisfies $X_1 = X_2X_3$ by explicitly taking the union of the construction of $X_1 = cX_3$ for all $c \in L(M_2)$. If $L(M_2)$ is an infinite set language, we construct a regular *under*-approximation of $X_1 = X_2X_3$ by considering a (finite) subset of $L(M_2)$ where the length is bounded. Formally speaking, for each $k \geq 0$ we can construct M_k, so that $w \in L(M_k), \hat{w}[1] = \hat{w}[2]\hat{w}[3]$, $\hat{w}[1] \in L(M_1)$, $\hat{w}[3] \in L(M_3)$, $\hat{w}[2] \in L(M_2)$ and $|\hat{w}[2]| \leq k$. It follows that M_k is a regular *under*-approximation of $X_1 = X_2X_3$. If $L(M_2)$ is a finite set language, there exists k (the length of the longest accepted word) where $L(M_k)$ is precise with respect to $X_1 = X_2X_3$. If $L(M_2)$ is an infinite set language, there does not exist such k so that $L(M_k)$ is precise with respect to $X_1 = X_2X_3$, as we have proven non-regularity of $X_1 = X_2X_3$.

4 Symbolic Reachability Analysis

Our symbolic reachability analysis involves two main steps: forward fixpoint computation and summarization.

Forward Fixpoint Computation The first phase of our analysis is a standard forward fixpoint computation on multi-track DFAs. Each program point is associated with a single multi-track DFA, where each track is associated with a single string variable $X \in \mathbf{X}$. We use $M[l]$ to denote the multi-track automaton at the program label l. The forward fixpoint computation algorithm (Algorithm 1) is a standard work-queue algorithm.

Initially, for all labels l, $L(M[l]) = \emptyset$. We iteratively compute the post-images of the statements and join the results to the corresponding automata. For a $stmt$ in the form: X:= $sexp$, the post-image is computed as:

$$\textsc{post}(M, stmt) \equiv (\exists X.M \cap \textsc{Construct}(X' = sexp, +))[X/X'].$$

$\textsc{Construct}(exp, b)$ returns the DFA that accepts a regular approximation of exp, where $b \in \{+, -\}$ indicates the direction (*over* or *under*, respectively) of approximation if needed. During the construction, we recursively push the negations ($\neg$) (and flip the direction) inside to the basic expressions ($bexp$), and use the corresponding construction of multi-track DFAs discussed in the previous section. We use function summaries to handle function calls. Each function f is summarized as a finite state transducer, denoted as M_f, which captures the relations among input variables (parameters), denoted as X_p, and return values. The return values are tracked in the output track, denoted as X_o. We discuss the generation of the transducer M_f below. For a $stmt$ in the form X:= `call` $f(e_1, \ldots, e_n)$, the post-image is computed as:

$$\textsc{post}(M, stmt) \equiv (\exists X, X_{p_1}, \ldots X_{p_n}.M \cap M_I \cap M_f)[X/X_o],$$

Algorithm 1 FORWARDRECAHABILITYANALYSIS(l_0)

```
1: Init(M);
2: queue WQ;
3: WQ.enqueue(l_0 : stmt_0);
4: while WQ ≠ NULL do
5:    e := WQ.dequeue(); Let e be l : stmt;
6:    if stmt is seqstmt then
7:       m := POST(M[l], stmt);
8:       PROPAGATE(m, l + 1);
9:    end if
10:   if stmt is if exp goto l' then
11:      m := M[l]∩ CONSTRUCT(exp, +);
12:      if L(m) ≠ ∅ then
13:         PROPAGATE(m, l');
14:      end if
15:      m := M[l]∩ CONSTRUCT(¬exp, +);
16:      if L(m) ≠ ∅ then
17:         PROPAGATE(m, l + 1);
18:      end if
19:   end if
20:   if stmt is assert exp then
21:      m := CONSTRUCT(exp, −);
22:      if L(M[l]) ⊈ L(m) then
23:         ASSERTFAILED(l);
24:      else
25:         PROPAGATE(M[l],l + 1);
26:      end if
27:   end if
28:   if stmt is goto L then
29:      for l' ∈ L do
30:         PROPAGATE(M[l],l');
31:      end for
32:   end if
33: end while
```

Algorithm 2 PROPAGATE(m, l)

```
1: m' := (m ∪ M[l])∇M[l];
2: if m' ⊈ M[l] then
3:    M[l] := m';
4:    WQ.enqueue(l);
5: end if
```

```
f(X)
begin
1: goto 2, 3;
2: X: = call f(X.a);
3: return X;
end
```

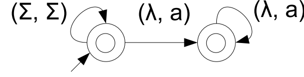

Fig. 4. A function and its summary DFA

where $M_I = \textsc{Construct}(\bigwedge_i X_{p_i} = e_i, +)$. The process terminates when we reach a fixpoint. To accelerate the fixpoint computation, we extend our automata widening operator [15], denoted as ∇, to multi-track automata. We identify equivalence classes according to specific equivalence conditions and merge states in the same equivalence class [1, 3]. The following lemma shows that the equality relations among tracks are preserved while widening multi-track automata.

Lemma 7. *if* $L(M) \subseteq L(x = y)$ *and* $L(M') \subseteq L(x = y)$, $L(M \nabla M') \subseteq L(x = y)$.

Summarization We compute procedure summaries in order to handle procedure calls. We assume parameter-passing with call-by-value semantics and we are able to handle recursion. Each function f is summarized as a multi-track DFA, denoted as M_f, that captures the relation among its input variables and return values.

Consider the recursive function f shown in Figure 4 with one parameter. f nondeterministically returns its input (goto 3) or makes a self call (goto 2) by concatenating its input and a constant a. The generated summary for this function is also shown in Figure 4. M_f is a 2-track DFA, where the first track is associated with its parameter X_{p_1}, and the second track is associated with X_o representing the return values. The edge (Σ, Σ) represents a set of identity edges. In other words, $\delta(q, (\Sigma, \Sigma)) = q'$ means $\forall a \in \Sigma, \delta(q, (a, a)) = q'$. The summary DFA M_f precisely captures the relation $X_o = X_{p_1}.a^*$ between the input variable and the return values.

During the summarization phase, (possibly recursive) functions are summarized as unaligned multi-track DFAs that specify the relations among their inputs and return values. We first build (possibly cyclic) dependency graphs to specify how the inputs flow to the return values. Each node in the dependency graph is associated with an unaligned multi-track DFA that traces the relation among inputs and the value of that node. An unaligned multi-track DFA is a multi-track DFA where λs might not be right justified. Return values of a function are represented with an auxiliary output track. Given a function f with n parameters, M_f is an unaligned $(n+1)$-track DFA, where n tracks represent the n input parameters and one track X_o is the output track representing the return values. We iteratively compute post images of reachable relations and join the results until we reach a fixpoint. Upon termination, the summary is the union of the unaligned DFAs associated with the return nodes. To compose these summaries at the call site, we also propose an alignment algorithm to align (so that λ's are right justified) an unaligned multi-track DFA.

Once the summary DFA M_f has been computed, it is not necessary to reanalyze the body of f. To compute the post-image of a call to f we intersect the values of

input parameters with M_f and use existential quantification to obtain the return values. Let M be a one-track DFA associated with X where $L(M) = \{b\}$. POST(M, $X :=$ `call` $f(X)$) returns M' where $L(M') = ba^*$ for the example function shown above. As another example, let M be a 2-track DFA associated with X, Y that is precise with respect to $X = Y$. Then POST(M, $X :=$ `call` $f(X)$) returns M' which is precise with respect to $X = Y.a^*$ precisely capturing the relation between X and Y after the execution of the function call. As discussed above, M' is computed by $(\exists X, X_{p_1}.M \cap M_I \cap M_f)[X/X_o]$, where $L(M_I) = $ CONSTRUCT($X_{p_1} = X$, $+$).

5 Experiments

We evaluate our approach against three kinds of benchmarks: 1) Basic benchmarks, 2) XSS/SQLI benchmarks, and 3) MFE benchmarks. These benchmarks represent typical string manipulating programs along with string properties that address severe web vulnerabilities.

Basic benchmarks. In the first set, we demonstrate that our approach can prove implicit equality properties of string systems. We wrote two small programs. CheckBranch (B1) has if branch ($X_1 = X_2$) and else branch ($X_1 \neq X_2$). In the else branch, we assign a constant string c to X_1 and then assign the same constant string to X_2. We check at the merge point whether $X_1 = X_2$. In CheckLoop (B2) we assign variables X_1 and X_2 the same constant string at the beginning, and iteratively append another constant string to both in an infinite loop. We check whether $X_1 = X_2$ at the loop exit. Let M accept the values of X_1 and X_2 upon termination. The equality assertion holds when $L(M) \subseteq L(M_a)$, where M_a is CONSTRUCT($X_1 = X_2$, $-$). We use ”$-$” to construct (under approximation) automata for assertions to ensure the soundness of our analysis. Using multi-track DFAs, we prove the equality property (result “true”) whereas we are unable to prove it using single-track DFAs (result “false”) as shown in Table 1 (B1 and B2). Though these benchmark examples are simple, to the best of our knowledge, there are no other string analysis tools that can prove equality properties in these benchmarks.

XSS/SQLI benchmarks. In the second set, we check existence of Cross-Site Scripting (XSS) and SQL Injection (SQLI) vulnerabilities in Web applications with known vulnerabilities. We check whether at a specific program point, a sensitive function may take an attack string as its input. If so, we say that the program is vulnerable (result “vul”) with respect to the given attack pattern. To identify XSS/SQLI attacks, we check intersection emptiness against all possible input values that reach a sensitive function at a given program point and the attack strings specified as a regular language. Though one can check such vulnerabilities using single-track DFAs [15], using multi-track automata, we can precisely interpret branch conditions, such as `$www=$url`, that cannot be precisely expressed using single-track automata, and obtain more accurate characterization of inputs of the sensitive functions. For the vulnerabilities identified in these benchmarks (S1 to S4), we did not observe false alarms that result from the approximation of the branch conditions.

MFE benchmarks. The last set of benchmarks show that the precision that is obtained using multi-track DFAs can help us in removing false alarms generated by single-track automata based string analysis. These benchmarks represent *malicious file execution* (MFE) attacks. Such vulnerabilities are caused because developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files. We systematically searched web applications for program points that execute file functions, such as `include` and `fopen`, whose arguments may be influenced by external inputs. At these program points, we check whether the retrieved files and the external inputs are consistent with what the developers intend. We manually generate a multi-track DFA M_{vul} that accepts a set of possible violations for each benchmark, and apply our analysis on the sliced program segments. Upon termination, we report that the file function is vulnerable (result "vul") if $L(M) \cap L(M_{vul}) \neq \emptyset$. M is the composed DFA of the listed single-track DFAs in the single-track analysis. As shown in Table 1 (M1 to M5), using multi-track DFAs we are able to verify that MFE vulnerabilities do not exist (result "no") whereas string analysis using single-track DFAs raises false alarms for all these examples.

Performance Evaluation. We have shown that multi-track DFAs can handle problems that cannot be handled by multiple single-track DFAs, but at the same time, they use more time and memory. For these benchmarks, the cost seems affordable. As shown in Table 1, in all tests, the multi-track DFAs that we computed (even for the composed ones) are smaller than the product of the corresponding single-track DFAs. One advantage of our implementation is symbolic DFA representation (provided by the MONA DFA library [5]), in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs). Using the symbolic DFA representation we avoid the potential exponential blow-up that can be caused by the product alphabet. However, in the worst case the size of the MBDD can still be exponential in the number of tracks.

6 Conclusion

In this paper, we presented a formal characterization of the string verification problem, investigated the decidability boundary for string systems, and presented a novel verification technique for string systems. Our verification technique is based on forward symbolic reachability analysis with multi-track automata, conservative approximations of word equations and summarization. We demonstrated the effectiveness of our approach on several benchmarks.

References

1. C. Bartzis and T. Bultan. Widening arithmetic automata. In *CAV*, pages 321–333, 2004.
2. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.
3. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, pages 372–386, 2004.

	Single-track				Multi-track			
Ben	Result	DFAs/ Composed DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Result	DFA state(bdd)	Time user+sys(sec)	Mem (kb)
B1	false	15(107), 15(107) /33(477)	0.027 + 0.006	410	true	14(193)	0.070 + 0.009	918
B2	false	6(40), 6(40) / 9(120)	0.022+0.008	484	true	5(60)	0.025+0.006	293
S1	vul	2(20), 9(64), 17(148)	0.010+0.002	444	vul	65(1629)	0.195+0.150	1231
S2	vul	9(65), 42(376)	0.017+0.003	626	vul	49(1205)	0.059+0.006	4232
S3	vul	11(106), 27(226)	0.032+0.003	838	vul	47(2714)	0.153+0.008	2684
S4	vul	53(423), 79(633)	0.062+0.005	1696	vul	79(1900)	0.226+0.003	2826
M1	vul	2(8), 28(208) / 56(801)	0.027+0.003	621	no	50(3551)	0.059+0.002	1294
M2	vul	2(20), 11(89) / 22(495)	0.013+0.004	555	no	21(604)	0.040+0.004	996
M3	vul	2(20), 2(20) / 5(113)	0.008+0.002	417	no	3(276)	0.018+0.001	465
M4	vul	24(181), 2(8), 25(188) / 1201(25949)	0.226+0.025	9495	no	181(9893)	0.784+0.07	19322
M5	vul	2(8), 14(101), 15(108) / 211(3195)	0.049+0.008	1676	no	62(2423)	0.097+0.005	1756

Table 1. Experimental results. DFA(s): the minimized DFA(s) associated with the checked program point. state: number of states. bdd: number of bdd nodes. Benchmark: Application, script (line number). S1: MyEasyMarket-4.1, trans.php (218). S2: PBLguestbook-1.32, pblguestbook.php (1210), S3:Aphpkb-0.71, saa.php (87), and S4: BloggIT 1.0, admin.php (23). M1: PBLguestbook-1.32, pblguestbook.php (536). M2: MyEasyMarket-4.1, prod.php (94). M3: MyEasyMarket-4.1, prod.php (189). M4: php-fusion-6.01, db_backup.php (111). M5: php-fusion-6.01, forums_prune.php (28).

4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.
5. BRICS. The MONA project. `http://www.brics.dk/mona/`.
6. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.
7. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC*, pages 87–96, 2007.
8. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654, 2004.
9. Y. Minamide. Static approximation of dynamically generated web pages. In *WWW*, pages 432–441, 2005.
10. M. Minsky. Recursive unsolvability of Post's problem of Tag and other topics in the theory of Turing machines. In *Ann. of Math (74)*, pages 437–455, 1961.
11. Open Web Application Security Project (OWASP). Top ten project. `http://www.owasp.org/`, May 2010.
12. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION*, pages 13–22, DC, USA, 2007.
13. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.
14. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, pages 154–157, 2010.
15. F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN*, pages 306–324, 2008.
16. F. Yu, T. Bultan, and O. H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *TACAS*, pages 322–336, 2009.