Verification of String Manipulating Programs Using Multi-Track Automata

Fang Yu
University of California, Santa Barbara
yuf@cs.ucsb.edu

Tevfik Bultan
University of California, Santa Barbara
bultan@cs.ucsb.edu

Oscar H. Ibarra
University of California, Santa Barbara
ibarra@cs.ucsb.edu

Abstract

Verification of string manipulation operations is a crucial problem in computer security. We present a new symbolic string verification technique that can be used to prove that vulnerabilities that result from improper string manipulation do not exist in a given program. We formally characterize the string verification problem as the reachability analysis of *string systems*, programs that contain only string variables and allow a limited set of operations on them. We show that string analysis problem is undecidable with even three variables if branch conditions that compare different variables are allowed. We develop a sound symbolic analysis technique for string verification that over-approximates the reachable states of the string system. We represent the set of string values that string variables can take using *multi-track deterministic finite automata* and implement a forward fixpoint computation using an automata based widening operation. In order to handle branch conditions in string systems, we describe the precise construction of multi-track DFAs for linear word equations, such as $c_1X_1c_2 = c'_1X_2c'_2$, as well as Boolean combinations of these equations. We show that non-linear word equations (even the simple one $X_1 = X_2X_3$) cannot be characterized precisely as a multi-track DFA. We propose a regular approximation for non-linear equations, such as $X_1 \dots X_i = X_{1'} \dots X_{i'}$, as well as Boolean combinations of these equations. We present a summarization technique for inter-procedural analysis that generates a transducer characterizing the relationship between the input parameters and the return values of each procedure. We implemented these algorithms using the MONA automata package and analyzed several PHP programs. Unlike prior string analysis techniques, our approach is able to keep track of the relationships among the string variables, improving the precision of the string analysis and enabling verification of assertions such as $X_1 = X_2$ where X_1 and X_2 are string variables.

1. Introduction

Web applications provide critical services over the Internet and frequently handle sensitive data. Unfortunately, Web application development is error prone and results in applications that are vulnerable to attacks by malicious users. The global accessibility of critical Web applications make this an extremely serious problem. In fact, in the Common Vulnerabilities and Exposures (CVE) list [5] (which documents computer security vulnerabilities and exposures) Web application vulnerabilities have occupied the first three positions in recent years. The most important Web application vulnerabilities are due to inadequate manipulation of string variables. According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious web application vulnerabilities [10], the top three vulnerabilities are: 1) Cross Site Scripting (XSS), 2) Injection Flaws (such as SQL injection) and 3) Malicious File Execution. All these vulnerabilities involve string manipulation and they occur due to inadequate sanitization and use of input strings provided by users.

In this paper, we investigate the *string verification problem:* Given a program that manipulates strings, we want to verify invariants 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 identify and 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 identify and prevent malicious file execution 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*: programs that contain only string variables and consist of 1) operations for manipulation of string variables (such as concatenation) and 2) branch conditions that allow comparisons among string variables and constants. After demonstrating that the string analysis problem is undecidable in general, we present and implement a symbolic string analysis technique that computes an over-approximation of the reachable states of a string system.

We use multi-track deterministic finite automata (DFA) as a symbolic representation to encode the set of possible values that string variables can take at a given program point. We implement a forward symbolic fixpoint computation to compute the reachable states. Since convergence is not guaranteed without approximation, we use an automata based widening operation. Unlike prior string analysis techniques, our approach 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.

In order to handle branch conditions in string systems, we describe the precise construction of multi-track DFAs for linear word equations, such as $c_1X_1c_2 = c'_1X_2c'_2$, as well as Boolean combinations of these equations. We show that non-linear word equations (even the simple one $X_1 = X_2X_3$) cannot be characterized pre-

[Copyright notice will appear here once 'preprint' option is removed.]

cisely as a multi-track DFA and there does not exist a multi-track DFA that corresponds to the tightest possible approximation. We propose a non-trivial regular approximation for non-linear equations, such as $X_1 \ldots X_i = {X_1}' \ldots {X_i}'$, as well as Boolean combinations of these equations. We show how these constructions can be used to compute the post-condition of branch conditions and assignment statements that involve concatenation.

We present a summarization technique for inter-procedural analysis that generates a transducer characterizing the relationship between the input parameters and the return values of each procedure. We extend our symbolic analysis technique by presenting algorithms for computing the post condition of complex string manipulation operations such as replacement. We implemented these algorithms using the MONA automata package and analyzed several PHP programs.

Related Work The use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [3]). In this paper, we focus on verification of string manipulation operations. 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 [4]. In this approach, first the flow graph is converted into a context free grammar where each string variable corresponds to a nonterminal, and each string operation corresponds to a production rule. Then, this grammar is converted to a regular language by computing an over-approximation. This type of grammar-based string analysis has been used to check for various types of errors in Web applications [7,8,12]. There are also several recent string analysis tools that use symbolic string analysis based on DFA encodings [6,11,14,15]. Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs [6, 11]. 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, 15].

Unlike the approach we propose in this paper, all of the results mentioned above use single track DFA and encode the reachable configurations of each string variable separately. This can cause two problems: 1) Branch conditions that check relations among different string variables can lead to imprecision in the analysis, resulting with false positives. 2) It is not possible to check invariants that refer to more than one string variable using these earlier techniques. Our multi-track automata encoding both improves the precision of the string analysis and it also enables verification of properties that cannot be verified with the previous approaches. Moreover, none of the above work investigate the boundary of decidability for the string verification problem. In this paper we show that string verification problem is undecidable even for deterministic string systems with only three string variables and non-deterministic string systems with only two string variables.

2. String Systems

We first define the string systems. Figure 1 presents the syntax of the string systems. 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) at this point. (We will discuss how to extend our analysis to other string operations, such as replace, in Section 7). 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.

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*
atom ::= *id* | *c*, where *c* is a string constant

Figure 1. The syntax of string manipulating programs

2.1 Two Simple Examples

Consider the following program segment:

```
1: input X1;
2: input X2;
3: if (X1 = X2) goto 7;
4: output "not equal";
5: assert (!(X1=X2));
6: goto 9;
7: output "equal";
8: assert (X1 = X2);
9: ...
```

This is a very simple string system which just tests equality of two input strings. However, existing automata-based string analysis techniques are not able to prove these assertions. The problem is, all the existing techniques 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 7 is the following: $L(M_{X_1,7}) = L(M_{X_2,7}) = L(M_{X_1,3}) \cap L(M_{X_2,3})$, i.e., the automata for both of the variables at the beginning of statement 7 accept the intersection of the languages accepted by the automata for these variables at the beginning of statement 3. Unfortunately, this is not strong enough to prove the assertion in line 8 (unless the intersection contains a single string). 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 $L(M_{X_2,4}) = L(M_{X_2,3})$. This is clearly not strong enough to prove the assertion in statement 5 (unless the intersection of $L(M_{X_1,3})$ and $L(M_{X_2,3})$ is empty).

Using the techniques presented in this paper, we can verify the assertions 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, y where $x, y \in \Sigma^*$. However, the multi-track automaton at the beginning of statement 7 will only accept pairs of strings x, y where $x, y \in \Sigma^*$ and $x = y$. Similarly, the multi-track automaton at the beginning of statement 4 will only accept pairs of strings x, y where $x, y \in \Sigma^*$ and $x \neq y$. Hence, we are able to prove the assertions in statements 5 and 8 using the multi-track automata representation.

Consider another simple example:

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

There are several challenges in proving that the assertion above holds. First, this program contains an infinite loop and does not terminate. 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. However, there exists a fixpoint characterizing the reachable configurations at each program point. We incorporate a widening operator to accelerate our symbolic reachability computation and compute an over-approximation of the fixpoint that characterizes the reachable configurations. Second, the assertion is an implicit property, 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. Finally, 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 this 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. Decidability and Undecidability Results

Before discussing our symbolic string analysis technique we prove that string analysis is an undecidable problem and, therefore, any sound string analysis technique has to use conservative approximations in order to guarantee convergence.

Let $S(X_1, X_2, \ldots, X_n)$ denote a string system with string variables $X_1, X_2, \ldots, X_n$ and a finite set of labeled instructions. There are several attributes we can use to classify string systems. For example, as mentioned above, a string system can be deterministic or non-deterministic. We can also classify a string system based on the alphabet used by the string variables, such as a string system with a unary alphabet or a string system with a binary alphabet, etc. Additionally, we can restrict the number of variables in the string systems, such as a string system with only 2 variables ($S(X_1, X_2)$) or 3 variables ($S(X_1, X_2, X_3)$), etc. Finally, we can restrict the set of string expressions that can be used in the assignment and conditional branch instructions.

In order to identify different classes of string systems we will use the following notation. 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. The following result is rather unexpected:

THEOREM 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 [9]. 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 $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3, X_2=X_3}$. ∎

In fact, Theorem 1 can be strengthened: There is a *fixed* string system $S(X_1, X_2, X_3)$ in $DUS(X_1, X_2, X_3)^{X_i:=X_ia}_{X_1=X_3, X_2=X_3}$ such that it is undecidable to determine, given an arbitrary nonnegative integer d, whether $S(X_1, X_2, X_3)$ will halt when X_1 is initially set to string a^d and X_2 and X_3 are initially set to ϵ. This follows from the fact that there exists a fixed universal 2-counter machine M that can simulate a universal single-tape deterministic Turing machine. Given a description of a Turing machine TM as input, M halts if and only if TM halts on blank tape. Since it is undecidable to determine if a Turing machine halts on blank tape, it is undecidable to determine if M will halt on some input. Since, we can construct a fixed string system $S(X_1, X_2, X_3)$ simulating M, as in Theorem 1, it is undecidable to determine if $S(X_1, X_2, X_3)$ will halt starting from some initial configuration.

Next, we show that the three variables in Theorem 1 are necessary in the sense that when there are only two variables, reachability is decidable. This result does not hold when the system is nondetereministic, as we shall see in Theorem 3.

Consider the class of deterministic 2-variable string systems where the constants are over an alphabet with arbitrary cardinality, and we are allowed to use conditional branch instructions of the form: **if** $X_1 = X_2$ **goto** L. (Note that because the alphabet is not necessarily unary, this **if** statement is not equivalent to **if** $|X_1| = |X_2|$ **goto** L as in the case of the unary alphabet.) Assignment statements are of the form: $X_i := X_ia$ or $X_i := aX_i$, where a is a single symbol. And, there is a halt instruction, which we may assume occurs at the end of the program.

THEOREM 2. *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 to show the decidability of reachability for multi-variable string systems as long as in a conditional branch statement we allow equality check between only two specific variables, i.e., no other variables can be compared for equality.

In contrast to Theorem 2, we can show that the halting problem is undecidable for nondeterministic 2-variable string systems with constants over the alphabet $\{a, b\}$, by a reduction from the Post Correspondence Problem (PCP) which is undecidable.

THEOREM 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. ∎

Theorem 2 demonstrates that there are non-trivial string analysis problems that are decidable. Theorems 1 and 3, on the other hand, show that the string analysis problem can be undecidable even when we restrict a deterministic string system to three variables or a non-deterministic string system to two variables. Since the general string analysis problem is undecidable, it is necessary to develop conservative approximation techniques for verification of string systems. In the following sections we present a symbolic verification technique that conservatively approximates the reachable states of a string system.

4. Regular Approximation of Word Equations

To analyze string systems, we approximate configurations over string variables as a regular language accepted by a multi-track deterministic finite automaton (DFA). Our analysis is based on the facts that: (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, complement, projection, and (3) the operations required during reachability analysis (such as equivalence checking) can be computed on DFAs.

Before we discuss how to perform symbolic reachability analysis on string systems, we introduce the multi-track DFAs and word equations in this section. We characterize word equations that can be expressed using multi-track DFAs, as well as detail the construction of these multi-track DFAs. Using these constructions, in the next section, we show how to perform symbolic reachability analysis on string systems.

4.1 Aligned 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 $\Sigma^n = (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\}) \times \ldots \times (\Sigma \cup \{\lambda\})$ (n times), where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ $(1 \leq i \leq n)$ to denote the i^{th} track of $w \in \Sigma^n$. An *aligned* multi-track DFA is a multi-track DFA where all tracks are left justified (i.e., λ's are right justified). I.e., if w is accepted by an aligned n-track DFA M, then for $1 \leq i \leq n$, $w[i] \in \Sigma^*\lambda^*$. We say $L(M)$ is an n-track language. We also use $\hat{w}[i] \in \Sigma^*$ to denote the longest λ-free prefix of $w[i]$. For the following descriptions, a multi-track DFA is an aligned multi-track DFA unless we explicitly state otherwise.

Multi-track DFAs are closed under intersection, disjunction, complementation, and homomorphism. Precisely: Given two n-track DFAs M_1, M_2, there exists an n-track DFA M that accepts $L(M_1) \cup L(M_2)$, or accepts $L(M_1) \cap L(M_2)$. Given an n-track DFA M_1, there exists an n-track DFA M that accepts the complement set of $L(M_1)$, and also there exists an $(n-1)$-track DFA M' that accepts $L(M_1 \downharpoonright_i)$, where $M_1 \downharpoonright_i$ denotes the result of erasing the i^{th} track (by homomorphism) of M_1.

4.2 Word Equations

A word equation is an equality relation of two words that concatenate a finite set of variables $\mathcal{X}$ and a finite set of constants $\mathcal{C}$. The general form of word equations is defined as $v_1 \dots v_n = v'_1 \dots v'_m$, where $\forall i, v_i, v'_i \in \mathcal{X} \cup \mathcal{C}$.

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

DEFINITION 4. *A word equation f is regular expressible if and only if there exists a multi-track DFA M such that M is precise with respect to f.*

4.2.1 Linear Word Equations

A linear word equation is a word equation where either side of the equation contains at most one variable. A general form of linear word equation is $c_1X_1c_2 = d_1X_2d_2$. Any linear word equation is equivalent to one of the following:

- $c'_1X_1c'_2 = X_2$ if $c_1 = d_1c'_1$ and $c_2 = c'_2d_2$,
- $c'_1X_1 = X_2d'_2$ if $c_1 = d_1c'_1$ and $d'_2c_2 = d_2$,
- $X_1c'_2 = d'_1X_2$ if $c_1d'_1 = d_1$ and $c_2 = c'_2d_2$,
- $X_1 = d'_1X_2d'_2$ if $c_1d'_1 = d_1$ and $d'_2c_2 = d_2$,
- *false* otherwise.

It follows that all linear equations can be reduced into two forms: (1) $X_1 = cX_2d$ or (2) $cX_1 = X_2d$, which are equivalent to $\exists X_k.X_1 = cX_k \wedge X_k = X_2d$ and $\exists X_k.cX_1 = X_k \wedge X_k = X_2d$.

THEOREM 5. *Linear word equations and Boolean combinations of these equations can be expressed using equations of the form $X_1 = X_2c$ and $X_1 = cX_2$, Boolean combinations of such equations and existential quantification.*

4.2.2 Non-linear Word Equations

A non-linear word equation is a word equation where at least one side of the equation has at least two variables. There are two basic forms of non-linear equations: $c = X_1X_2$ and $X_1 = X_2X_3$.

THEOREM 6. *Non-linear word equations and Boolean combinations of these equations can be expressed using equations of the form $c = X_1X_2$ and $X_1 = X_2X_3$, Boolean combinations of such equations and existential quantification.*

For example, $X_1 = X_2dX_3X_4$ is equivalent to $\exists X_{k_1}, X_{k_2}.X_1 = X_2X_{k_1} \wedge X_{k_1} = dX_{k_2} \wedge X_{k_2} = X_3X_4$.

In the following, we show how to construct the corresponding multi-track DFAs for the basic forms of linear and non-linear word equations: (1) $X_1 = X_2c$, (2) $X_1 = cX_2$, (3) $c = X_1X_2$, and (4) $X_1 = X_2X_3$. Note that, based on the fact that multi-track DFAs are closed under intersection, disjunction, complementation, and homomorphism, we can construct the corresponding multi-track DFAs for all word equations both linear and non-linear, as well as their Boolean combinations based on the constructions for these basic forms. Note that the boolean operations conjunction, disjunction and negation can be handled with intersection, disjunction, and complementation of the multi-track automata, 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 $\exists X_i.f$.

Before delving into these constructions, we summarize our results in the following theorem:

THEOREM 7. *(1) Linear word equations are regular expressible, as well as their Boolean combinations. (2) $X_1 = cX_2$ is regular expressible but the corresponding M has exponential number of states in the length of c. (3) $X_1 = X_2X_3$ is not regular expressible.*

4.3 Construction of Multi-track DFAs for Word Equations

Given a DFA $M = \langle Q, \Sigma, \delta, I, F\rangle$, Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $I \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (accepting) states. We say a state $q \in Q$ is a $sink$ state if $q \notin F$ and $\forall a \in \Sigma, \delta(q, a) = q$. The $sink$ states are also extended to multi-track DFAs. In the following constructions, we ignore transitions that go to sink states, and assume that all unspecified transitions go to sink states.

Before we give the constructions, we generalize the problem of constructing multi-track DFAs for word equations as follows. We assume that each variable in $\mathcal{X}= \{X_1, X_2, \dots, X_n\}$ is associated with an automaton $M_i = \langle Q_i, \Sigma, \delta_i, I_i, F_i\rangle$, where $L(M_i)$ denotes the set of values that the variable X_i can take. Then, given a word equation f over $\mathcal{X}= \{X_1, X_2, \dots, X_n\}$, we say that *an n-track DFA M under-approximates f within $M_1, \dots M_n$*, if for all $w \in L(M)$, $f[\hat{w}[1]/X_1, \dots, \hat{w}[n]/X_n]$ holds and for all $1 \le i \le n$, $\hat{w}[i] \in L(M_i)$. We say that *an n-track DFA M over-approximates f within $M_1, \dots M_n$*, if for any $s_1, \dots, s_n \in \Sigma^*$ where $f[s_1/X_1, \dots, s_n/X_n]$ holds and for all $1 \le i \le n$, $s_i \in L(M_i)$, there exists $w \in L(M)$ such that for all $1 \le i \le n, \hat{w}[i] = s_i$. Note that, for either case, for any word $w \in L(M)$, for all $1 \le i \le n, \hat{w}[i] \in L(M_i)$.

4.3.1 The Construction of $X_1 = X_2c$

Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1\rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2\rangle$ be two DFAs that accept possible values of variables X_1 and X_2, respectively. We present the construction of a 2-track DFA $M = \langle Q, \Sigma, \delta, I, F\rangle$, such that M is precise with respect to $X_1 = X_2c$ within M_1, M_2.

Let $sink_1$ be the sink state of M_1, and $sink_2$ be the sink state of M_2. Let $c = a_1a_2 \dots a_n$, where $\forall 1 \le i \le n, a_i \in \Sigma$ and n is the length of the constant string c. $M = \langle Q, \Sigma^2, \delta, q_0, F\rangle$ is constructed as:

- $Q \subseteq Q_1 \times Q_2 \times \{0, \dots, n\}$,
- $I = (I_1, I_2, 0)$,
- $\forall a \in \Sigma, \delta((r, p, 0), (a, a)) = (\delta_1(r, a), \delta_2(p, a), 0)$, if $\delta_1(r, a) \neq sink_1$ and $\delta_2(p, a) \neq sink_2$
- $\forall a_i, p \in F_2, \delta((r, p, i), (a_i, \lambda)) = (\delta_1(r, a_i), p, i + 1)$,
- $F = \{(r, p, i) \mid r \in F_1, p \in F_2, i = n\}$.

Note that M simulates M_1 and M_2 making sure that both tracks are the same until a final state of M_2 is reached. Then, the second track reads the symbol λ while the first track reads the constant c, and the automaton goes to a final state when c is consumed. $|Q|$ is $O(|Q_1| \times |Q_2| + n)$ since in the worst case Q will contain all possible combinations of states in Q_1 and Q_2 followed with a tail of n states for recognizing the constant c. For the automaton M resulting from the above construction we have, $w \in L(M)$ if and only if $\hat{w}[1] = \hat{w}[2]c$, $\hat{w}[1] \in L(M_1)$ and $\hat{w}[2] \in L(M_2)$, i.e., M is precise with respect to $X_1 = X_2c$ (within M_1, M_2), hence, $X_1 = X_2c$ is regular expressible.

4.3.2 The Construction of $X_1 = cX_2$

Let $M_1 = \langle Q_1, \Sigma, \delta_1, I_1, F_1\rangle$, $M_2 = \langle Q_2, \Sigma, \delta_2, I_2, F_2\rangle$ be two DFAs that accept possible values of variables X_1 and X_2, respectively. Below we present the construction of a 2-track DFA M, such that M is precise with respect to $X_1 = cX_2$ within M_1, M_2. Let $c = a_1a_2\ldots a_n$, where $\forall 1 \le i \le n, a_i \in \Sigma$ and n is the length of the constant string c.

The intuition behind the construction of M is as follows. In the initial stage (denoted as *init* below), M makes sure that the first track matches the constant c, while recording the string that is read in the second track in a buffer (a vector of symbols) stored in its state. After c is consumed, M goes to the next stage (denoted as *match* below) and matches the symbols read in the first track with the next symbol stored in the buffer while continuing to store the symbols read in the second track in the buffer. Note that, the kth symbol read in track 2 has to be matched with the $(k+n)$th symbol read in track 1. So, the buffer stores the symbols read in track 2 until the corresponding symbol in track 1 is observed.

Let $\vec{v}$ be a size n vector. For $1 \le i \le n$, $\vec{v}[i] \in \Sigma \cup \{\perp\}$. The vector $\vec{v}' = \vec{v}[i := a]$ is defined as follows: $\vec{v}'[i] = a$ and $\forall j \ne i$, $\vec{v}'[j] = \vec{v}[j]$. $M = \langle Q, \Sigma^2, \delta, I, F\rangle$ is constructed as:

- $Q \subseteq Q_1 \times Q_2 \times \{1, \ldots, n\} \times (\Sigma \cup \{\perp\})^n \times \{init, match\}$,
- $I = (I_1, I_2, 1, \vec{v}_\perp, init)$, where $\forall i, \vec{v}_\perp[i] = \perp$,
- $\forall a \in \Sigma, 1 \le i < n, \delta((r, p, i, \vec{v}, init), (a_i, a)) = (\delta_1(r, a_i), \delta_2(p, a), i+1, \vec{v}[i := a], init)$,
- $\forall a \in \Sigma, i = n, \delta((r, p, i, \vec{v}, init), (a_i, a)) = (\delta_1(r, a_i), \delta_2(p, a), 1, \vec{v}[i := a], match)$,
- $\forall a, b \in \Sigma, 1 \le i < n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, b)) = (\delta_1(r, a), \delta(p, b), i+1, \vec{v}[i := b], match)$,
- $\forall a, b \in \Sigma, i = n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, b)) = (\delta_1(r, a), \delta(p, b), 1, \vec{v}[i := b], match)$,
- $\forall a \in \Sigma, p \in F_2, 1 \le i < n, \vec{v}[i] = a$, $\delta((r, p, i, \vec{v}, match), (a, \lambda)) = (\delta_1(r, a), p, i+1, \vec{v}[i := \perp], match)$,
- $\forall a \in \Sigma, p \in F_2, i = n, \vec{v}[i] = a, \delta((r, p, i, \vec{v}, match), (a, \lambda)) = (\delta_1(r, a), p, 1, \vec{v}[i := \perp], match)$,
- $F = \{(r, p, i, \vec{v}_\perp, match) \mid r \in F_1, p \in F_2\}$.

Since M accepts the set $\{w \mid \hat{w}[1] = c\hat{w}[2], \hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2)\}$, $X_1 = cX_2$ is regular expressible. However, the number of states of M is exponential in c. Below, we show that the exponential number of states is inevitable.

4.3.3 Intractability of $X_1 = cX_2$

Consider the equation $X_1 = cX_2$, where c is a constant string of length n. Let $L(M_1)$ and $L(M_2)$ be regular languages. Define the 2-track language:

$$L = \{(x_1x_2, y_1y_2\lambda^n) \mid x_1x_2 \in L(M_1), y_1y_2 \in L(M_2), k \ge n, |x_1x_2| = k, |x_1| = |y_1| = n, x_1 = c, x_2 = y_1y_2\}$$

Note that any automaton m that accepts the language L defined above will be precise with respect to the the equation $X_1 = cX_2$ (within M_1 and M_2).

THEOREM 8. *Any nondeterministic finite automaton (NFA) M needs at least 2^n states to accept L.*

Proof: Let $c = 1^n$ and consider the regular languages $L(M_1) = (0+1)^+$ and $L(M_2) = (0+1)^+$. Suppose M is an NFA accepting L. Consider any pair of distinct strings y_1 and y_1' of length n. Then M will accept the following 2-track strings:

$(1^nx_2, y_1y_2\lambda^n)$, where $x_2, y_1, y_2 \in (0+1)^+$, $k \ge n$, $|1^nx_2| = k, |y_1| = n, x_2 = y_1y_2$, and

$(1^nx_2', y_1'y_2'\lambda^n)$, where $x_2', y_1', y_2' \in (0+1)^+$, $k \ge n$, $|1^nx_2'| = k, |y_1'| = n, x_2' = y_1'y_2'$

Suppose in processing $(1^nx_2, y_1y_2\lambda^n)$, M enters state q after processing the initial 2-track segment $(1^n, y_1)$, and in processing $(1^nx_2', y_1'y_2'\lambda^n)$, M enters state q' after processing the initial 2-track segment $(1^n, y_1')$. Then $q \ne q'$; otherwise, M will also accept $(1^nx_2, y_1'y_2\lambda^n)$. This is a contradiction, since $x_2 \ne y_1'y_2$.

Since there are 2^n distinct strings y of length n, it follows that M must have at least 2^n states. ∎

4.3.4 The Construction of $c = X_1X_2$

Below we briefly describe the construction of a 2-track DFA M, such that M is precise with respect to $c = X_1X_2$ within the given regular sets characterizing possible values of X_1 and X_2. Assume that $c = a_1\ldots a_n$. We can split c to two strings $a_1\ldots a_k$ and $a_{k+1}\ldots a_n$ so that $c = a_1\ldots a_ka_{k+1}\ldots a_n$. There are $n+1$ such splits. For each of them, if $a_1\ldots a_k \in L(M_1)$ and $a_{k+1}\ldots a_n \in L(M_2)$, then if $k \ge n-k$, $(a_1\ldots a_k, a_{k+1}\ldots a_n\lambda^{2k-n})$ should be accepted by M and if $k < n-k$, $(a_1\ldots a_k\lambda^{n-2k}, a_{k+1}\ldots a_n)$ should be accepted by M. We can construct an automaton M with $O(n^2)$ states that accepts this language by explicitly checking each of these $n+1$ cases. Since we can construct this 2-track DFA, it follows that $c = X_1X_2$ is regular expressible.

4.3.5 Non-Regularity of $X_1 = X_2X_3$

We first show that $X_1 = X_2X_3$ is not regular expressible, and later we give constructions of 3-track DFAs that over-approximate or under-approximate $X_1 = X_2X_3$.

Given M_1, M_2, M_3, let $L = \{w \mid \hat{w}[1] = \hat{w}[2]\hat{w}[3], \hat{w}[1] \in L(M_1), \hat{w}[2] \in L(M_2), \hat{w}[3] \in L(M_3)\}$.

THEOREM 9. *L is not necessarily a regular language.*

Proof: Let $L(M_1) = a^+b^+$, $L(M_2) = a^+$, and $L(M_3) = b^+$. Suppose L is regular and is accepted by a 3-track DFA M. Then M when given a 3-track string consisting of:
a^sb^t
$a^i\lambda^{s+t-i}$
$b^j\lambda^{s+t-j}$
accepts if and only if $s = i$ and $t = j$. Clearly, we can construct a 3-track DFA M' which accepts 3-track strings of the form:
a^sb^t
$a^i\lambda^{s+t-i}$
$b^i\lambda^{s+t-i}$
We can then construct another 3-track DFA M'' which accepts $L(M) \cap L(M')$. But $L(M'')$ consists of 3-track strings of the form:
a^ib^i
$a^i\lambda^{s+t-i}$
$b^i\lambda^{s+t-i}$
It follows that we can construct a 1-track NFA from M'' which accepts the language $\{a^ib^i \mid i \ge 1\}$ (by erasing the second and third tracks by homomorphism), which is not regular and leads to a contradiction. ∎

4.3.6 The Approximation of $X_1 = X_2X_3$

Below we 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. $M = \langle Q, \Sigma^3, \delta, I, F\rangle$ is constructed as follows.

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

$|Q|$ 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 properties 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 show a regular *under*-approximation construction of $X_1 = X_2X_3$. Note 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 can still use this idea to 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 $X_1 = X_2X_3$. The following lemma holds by construction.

LEMMA 10. *$L(M_{k_1}) \subseteq L(M_{k_2})$ if $k_1 \leq k_2$.*

To sum up, if $L(M_2)$ is a finite set language, there exists k (the length of the longest accepted word) so that $L(M_k)$ 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 $X_1 = X_2X_3$.

We say a regular under-approximation M_κ is *tightest* if $L(M_\kappa)$ is an under-approximation of $X_1 = X_2X_3$ and for all M' where M' is an under-approximation of $X_1 = X_2X_3$ we have $L(M') \subseteq L(M_\kappa)$. Since the precision of a regular under-approximation can be always improved by adding new words to the language, the tightest regular under-approximation does not exist if $L(M_2)$ is not finite.

5. Symbolic Reachability Analysis

In this section, we present our symbolic reachability analysis for string systems. Our approach consists of two phases. In the first phase, we use one multi-track DFA for each program point to symbolically represent possible values of string variables at that program point, where each track corresponds to one string variable. Our approach is based on a forward fixpoint computation on multi-track DFAs. We iteratively compute post-images of reachable states and join the results until we reach a fixpoint. We use summaries to handle functions calls. During the forward fixpoint computation if we encounter a call to a function that has not been summarized, we go to the second phase of the analysis, which is summarization. Each function is summarized when needed, and once a function is summarized, the summary DFA is used to compute the return values at the call sites without going through the body of the function. During the summarization phase, (recursive) functions are summarized as unaligned multi-track DFAs that specify the relations among their inputs and return values. We first build (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. 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.

5.1 Forward Fixpoint Computation

The first phase of our analysis is a standard forward fixpoint computation on multi-tack DFAs. Each program point is associated with a single multi-track DFA, where each track is associated with a single string variable $X \in \mathcal{X}$. We use $M[l]$ to denote the multi-track automaton at the program label l. The forward fixpoint computation algorithm is a standard work-queue algorithm as shown in Algorithm 3. 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. The process terminates when we reach a fixpoint.

5.1.1 Widening Operation

Since string systems are infinite state systems, an iterative reachability computation may not terminate. We incorporate an automata widening operator, denoted as ∇, proposed in [1] to accelerate the fixpoint computation. Given two finite automata $M = \langle Q, \Sigma, \delta, I, F\rangle$ and $M' = \langle Q', \Sigma, \delta', I', F'\rangle$, we define an equivalence relation $\equiv_\nabla$ on $Q \cup Q'$ as follows: Given $q \in Q$ and $q' \in Q'$, we say that $q \equiv_\nabla q'$ and $q' \equiv_\nabla q$ if and only if 1) the language recognized by starting from the state q in M and the language recognized by starting from the state q' in M' are the same, or 2) there exists a word w such that M reaches q after consuming w from its initial state I and M' reaches q' after consuming w from its initial state I'. For $q_1 \in Q$ and $q_2 \in Q$ we say that $q_1 \equiv_\nabla q_2$ if and only if $\exists q \in Q \cup Q'.\ q_1 \equiv_\nabla q \wedge q_2 \equiv_\nabla q$.

Let C be the set of equivalence classes of $\equiv_\nabla$. We define $M \nabla M' = \langle Q'', \Sigma, \delta'', I'' F''\rangle$ as follows:

- $Q'' = C$
- $I'' = c$ such that $I \in c \wedge I' \in c$
- $\delta''(c_i, \sigma) = c_j$ if $(\forall q \in c_i \cap Q\ .\ \delta(q, \sigma) \in c_j \vee \delta(q, \sigma) = sink) \wedge (\forall q' \in c_i \cap Q'\ .\ \delta'(q', \sigma) \in c_j \vee \delta'(q', \sigma) = sink)$
- $c \in F''$ if $\exists q \in F \cup F'.\ q \in c$.

In other words, the set of states of $M \nabla M'$ is the set C of equivalence classes of $\equiv_\nabla$. Transitions are defined based on the transitions of M and M'. The initial state is the class containing the initial states I and I'. The set of final states is the set of classes that contain some of the final states in F and F'. It can be shown that, given two automata M and M', $L(M) \cup L(M') \subseteq L(M \nabla M')$.

In Figure 2, we give an example for the application of this widening operation to two 2-track DFAs. $L(M) = \{(a, a)(b, b)\}$ and $L(M') = \{(a, a)(b, b), (a, a)(b, b)(b, b)\}$. The set of equivalence classes for $M \nabla M'$ is $C = \{q''_0, q''_1, q''_2\}$, where $q''_0 =$

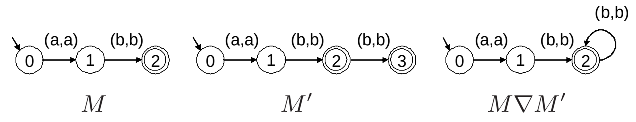

Figure 2. Widening automata

$\{q_0, q_0'\}$, $q_1'' = \{q_1, q_1'\}$, $q_2'' = \{q_2, q_2', q_3'\}$, and we have $L(M\nabla M') = (a,a)(b,b)^+$. Note that, these are the automata our symbolic analysis computes for the program point 5 of the second example program segment in Section 2.

5.1.2 Automata Construction

CONSTRUCT(exp, b) returns the DFA that accepts a regular approximation of exp. The sign $b \in \{+,-\}$ indicates the direction of approximation if needed. The operation $\bar{b}$ flips the sign. Algorithm 1 recursively pushes the negations ($\neg$) inside to the basic expressions ($bexp$). For each basic expression $bexp$, we have shown the corresponding construction of multi-track DFAs in the previous section. CONSTRUCT($bexp$, $+$) returns a DFA that over-approximates $bexp$, while CONSTRUCT($bexp$, $-$) returns a DFA that under-approximates $bexp$, for those cases where $bexp$ is not regular expressible. Both CONSTRUCT($bexp$, $+$) and CONSTRUCT($bexp$, $-$) return a DFA that is precise with respect to $bexp$ if $bexp$ is regular expressible. We also incorporate the standard DFA operations, e.g., intersection at line 2, union at line 4, and complement at line 10.

Algorithm 1 CONSTRUCT(exp, b)

```
1: if $exp$ is $exp_1 \wedge exp_2$ then
2:     return CONSTRUCT($exp_1$, $b$) ∩ CONSTRUCT($exp_2$, $b$);
3: else if $exp$ is $\neg(exp_1 \wedge exp_2)$ then
4:     return CONSTRUCT($\neg exp_1$, $b$) ∪ CONSTRUCT($\neg exp_2$, $b$);
5: else if $exp$ is $\neg(\neg exp_1)$ then
6:     return CONSTRUCT($exp_1$, $b$);
7: else if $exp$ is $bexp$ then
8:     return CONSTRUCT($bexp$, $b$);
9: else if $exp$ is $\neg bexp$ then
10:    return COMPLEMENT(CONSTRUCT($bexp$, $\bar{b}$));
11: end if
```

For a $stmt$ in the form: X:= $sexp$, the post-image is computed as follows:

$$\text{POST}(M, stmt) \equiv (\exists X. M \cap \text{CONSTRUCT}(X' = sexp, +))[X/X'].$$

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 will detail how to generate M_f in Section 6. For a $stmt$ in the form X:= `call` $f(e_1, \ldots, e_n)$, POST(M, $stmt$) returns the result of $(\exists X, X_{p_1}, \ldots X_{p_n}. M \cap M_I \cap M_f)[X/X_o]$, where M_I = CONSTRUCT($\bigwedge_i X_{p_i} = e_i$, $+$).

During the fixpoint computation, we report assertion failures if $M[l]$ accepts some string that violates the assertion labeled l. Note that at line 21 we compute an under approximation of the assertion expression to ensure the soundness of our analysis. Finally, a program label l is not reachable if $L(M[l])$ is empty. Our analysis is sound but incomplete due to the following approximations: (1) regular approximation for non-linear word equations, (2) the widening operation and (3) summarization.

6. Summarization

In this section, we discuss how to compute function summaries. We assume parameter-passing with call-by-value semantics and we are able to handle recursion. Each function f is summarized as an unaligned multi-track DFA, denoted as M_f, that captures the relation among its input variables and return values. An unaligned multi-track DFA is a multi-track DFA where λs are not 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. Once M_f has been computed, it is not necessary to reanalyze the body of f. Instead, one can intersect the values of input parameters with M_f to obtain the return values. Our approach consists of three steps: (1) Building the dependency graph, (2) generating the summary, and (3) alignment.

Algorithm 2 PROPAGATE(m, l)

```
1: $m'$ := $M[l] \nabla (m \cup M[l])$;
2: if $m' \not\subseteq M[l]$ then
3:     $M[l]$ := $m'$;
4:     $WQ$.enqueue($l$);
5: end if
```

Algorithm 3 FORWARDRECAHABILITYANALYSIS(l_0)

```
1: Init($M$);
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 $seqstmt$ 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 $L(m) \neq \emptyset$ then
13:            PROPAGATE($m$, $l'$);
14:        end if
15:        $m$ := $M[l]$∩ CONSTRUCT($\neg exp$, $+$);
16:        if $L(m) \neq \emptyset$ 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 $L(M[l]) \not\subseteq L(m)$ 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 $l' \in L$ do
30:            PROPAGATE($M[l]$,$l'$);
31:        end for
32:    end if
33: end while
```

6.1 Dependency Graph

Given a function f, the dependency graph G_f specifies how the inputs flow to the return values in f. Formally speaking, a dependency graph $G = \langle N, E\rangle$ is a directed graph, where N is a finite set of nodes and $E \subseteq N \times N$ is a finite set of directed edges. An edge $(n_i, n_j) \in E$ identifies that the value of n_j depends on the value of n_i. Each node $n \in N$ can be

- a `normal` node including `return`, `input`, `constant`, `variable`,
- an `operation` node including `concat` and `call`.

A `return` node is a sink node (no successors) that corresponds to a return statement. An `input` node corresponds to a parameter of the function f, labeled as $f.p_i$, where i indicates the i^{th} parameter. A `constant` node is associated with a constant value. Both `input` and `constant` nodes have no predecessors. A `concat` node n has two predecessors labeled as the prefix node ($n.p$) and the suffix node ($n.s$), and stores the concatenation of any value of the prefix node and any value of the suffix node in n. A `call` node is associated with a function $callee$. If $callee$ has m parameters, there are m predecessors of a `call` node as its arguments (labeled as $n.a_1, \ldots, n.a_m$).

Assume that we want to compute the summary of a given function $main$. Let F denote the set of related functions that include $main$ and its $callees$ (including nested function calls). Our first step is generating the dependency graph for each $f \in F$, which is done by a bottom-up dependency analysis starting from the return statements.

Let the dependency graph of f be $G_f = \langle N_f, E_f\rangle$. To simplify the description, we use $Input(G_f)$ to denote the set of its `input` nodes, $Call(G_f)$ to denote the set of its `call` nodes, and $Return(G_f)$ to denote the set of its `return` nodes. For each function f (callee), we use $Caller(f)$ to denote the set of `call` nodes that are associated with f.

Our second step is generating a composed dependency graph G_F from $\{G_f \mid f \in F\}$. $G_F = \langle N_F, E_F\rangle$ is constructed as follows:

- $N_F = \cup_{f\in F} N_f$.
- $E_F = E_n \cup E_i \cup E_r$, where
  - $E_n = \{(n, n') \mid f \in F, (n, n') \in E_f, n' \notin Call(G_f)\}$.
  - $E_i = \{(n.a_i, callee.p_i) \mid f \in F, n \in Call(G_f)\}$. $callee.p_i$ is the `input` node that identifies the i_{th} parameter of the function $callee$ associated with n.
  - $E_r = \{(n, n') \mid f \in F, n \in Return(G_f), n' \in Caller(f)\}$.

Briefly, G_F connects the set of G_f by (1) redirecting the predecessors of `call` nodes to the `input` nodes of their callees, and (2) adding edges that direct `return` nodes of callees to the `call` nodes of their callers. For $n \in N_F$, $Succ(n) = \{n' \mid (n, n') \in E_F\}$ is the set of successors of n and $Pred(n) = \{n' \mid (n', n) \in E_F\}$ is the set of predecessors of n. We also define $Input(G_F) = \{n \mid Pred(n) = \emptyset\}$. Note that after composition, a `return` node may have successors and an `input` node may have predecessors.

6.2 Generating Function Summaries

In this section, we describe how to compute a summary on G_F, given two sets of nodes In and Out. If we aim to summarize function f ($f \in F$), $In \subseteq N_f$ is the set of its `input` nodes and $Out \subseteq N_f$ is the set of its `return` nodes in G_f. Each $n \in In$ recognizes one input variable, denoted as X_n, and the summary of $\langle G_F, In, Out\rangle$ is an unaligned $(|In|+1)$-track DFA. The first $|In|$ tracks are labeled as X_n for each $n \in In$. The extra track, labeled as X_o, is used to record the output values.

The algorithm to generate the summary is shown in Algorithm 4. We use a DFA vector S to record the reachable summary at each node. We initialize S at line 1. Initially, for each $n \in In$, $S[n]$ is a 2-track (associated with X_n and X_o) DFA that accepts the identity relation on X_n and X_o. For each $n \in Input(G_F)\backslash In$, $S[n]$ is a 1-track (associated with X_o) DFA that accepts Σ^* if n is a `variable` node, or a constant value if n is a `constant` node. For the rest, i.e., $n \notin In$, $S[n]$ accepts an empty set. Similar to Algorithm 3, the algorithm is a standard work queue algorithm incorporating the automata widening operator. We iteratively update the summary at each node until reaching a fixpoint.

We only consider one string operation: concatenate. Note that summaries may have tracks that are associated with different variables. Assume that we wish to compute ConcatSummary(S_1, S_2) where S_1 represents the summary at the prefix node and S_2 represents the summary at the suffix node. Let $S_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1\rangle$ be a multi-track DFA whose tracks are associated with the set of input variables χ_1 and X_o where $\Sigma_1 = (\Sigma \cup \lambda)^{|\chi_1|} \times \Sigma$. Let $S_2 = \langle Q_2, \Sigma_2, \delta_2, I_2, F_2\rangle$ be a multi-track DFA whose tracks are associated with the set of input variables χ_2 and X_o where $\Sigma_2 = (\Sigma \cup \lambda)^{|\chi_2|} \times \Sigma$. We first extend S_1 and S_2 to the DFAs that have common tracks, so that both are associated with $\chi_1 \cup \chi_2$ and X_o.

The extension of S_1, denoted as S_1^λ, is $\langle Q_1, \Sigma_1^\lambda, \delta_1^\lambda, I_1, F_1\rangle$, where

- $\Sigma_1^\lambda = (\Sigma \cup \lambda)^{|\chi_1|} \times \lambda^{|\chi_2 - \chi_1|} \times \Sigma$, and
- $\delta_1^\lambda(q, \alpha) = q'$ if $\delta_1(q, \beta) = q'$ and $\alpha[X] = \beta[X]$ if $X \in \chi_1 \cup X_o$, and $\alpha[X] = \lambda$, otherwise.

The extension of S_2, denoted as S_2^λ, is $\langle Q_2, \Sigma_2^\lambda, \delta_2^\lambda, I_2, F_2\rangle$, where

- $\Sigma_2^\lambda = \lambda^{|\chi_1|} \times (\Sigma \cup \lambda)^{|\chi_2 - \chi_1|} \times \Sigma$, and
- $\delta_2^\lambda(q, \alpha) = q'$ if $\delta_2(q, \beta) = q'$ and $\alpha[X] = \lambda$ if $X \in \chi_1$, and $\alpha[X] = \beta[X]$, otherwise.

Intuitively, we extend S_1 (prefix) by allowing only λ in the added tracks, while we extend S_2 (suffix) by allowing only λ in both the added tracks and the common tracks that are also associated with S_1. ConcatSummary(S_1, S_2) returns the $(|\chi_1 \cup \chi_2| + 1)$-track DFA that accepts the concatenation of S_1^λ and S_2^λ.

To deal with the union or widening operator on S_1 and S_2 that are associated with different variables, we extend both tracks to $\chi_1 \cup \chi_2$ and X_o by allowing arbitrary symbols in the added tracks. I.e., the value of an unspecified track is not restricted. We then perform union or widening on these extension DFAs. Finally, the summary of $\langle G_F, In, Out\rangle$ is the union of the DFAs that are associated with nodes in Out.

In sum, to summarize a specific function f, we first find the set of related functions F. The summary of f, denoted as M_f, is the result of GenerateSummary(G_F, In, Out), where $In = \{n \mid n \in Input(G_f)$, where n is not a `constant` node$\}$, and $Out = \{n \mid n \in Return(G_f)\}$. The alphabet of M_f is $(\Sigma \cup \lambda)^{|In|} \times \Sigma$. Let $w[X]$ be a word projected to the track associated with X. For any $w \in L(M_f)$, we have the following:

- $\forall X \in In, w[X] \in \lambda^*\Sigma^*\lambda^*$, and
- $w[X_o] \in \Sigma^*$.

6.3 Another Simple Example

Consider another simple example given below. Function f has one parameter X, which non-deterministically returns its input (goto 2) or makes a self call (goto 3) by concatenation its input and the constant a. Let $F = \{f\}$. G_f and G_F are shown in Figure 3.

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

The generated summary is shown in Figure 4. M_f is an aligned 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. I.e., if $\delta(q, (\Sigma, \Sigma)) = q'$ then $\forall a \in \Sigma, \delta(q, (a, a)) = q'$. The summary DFA M_f precisely captures the relation $X_o = X_{p_1}.a^*$ between the input variable and the return values. We can use this

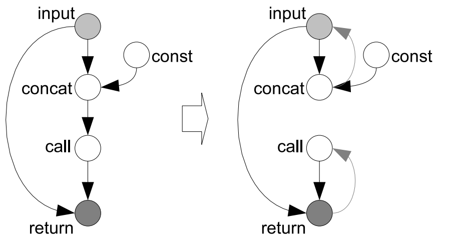

Figure 3. G_f and G_F: The dependency graphs

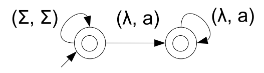

Figure 4. M_f: The summary DFA

summary DFA to compute the post-image of a call to f without analyzing f further. For example, let M be a one-track DFA associated with X where $L(M) = \{b\}$. POST(M, X := `call` $f(X)$) returns M' where $L(M') = ba^*$. 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. Recall that M' is computed by $(\exists X, X_{p_1}.M \cap M_I \cap M_f)[X/X_o]$, where $L(M_I) = \text{CONSTRUCT}(X_{p_1} = X, +)$.

In general, M_f can be an unaligned multi-track DFA (λs are not right justified). The final step of our construction is to align M_f before composition. Unfortunately, an unaligned multi-track DFA may not be definable by an aligned multi-track DFA. We discuss how to align an unaligned multi-track DFA in the next section.

Algorithm 4 GENERATESUMMARY(G_F, In, Out)

```
1:  INIT($S, Input(G_F), In$);
2:  queue $WQ := NULL$;
3:  for $n \in In \cup Input(G_F)$ do
4:      $WQ$.enqueue($Succ(n)$);
5:  end for
6:  while $WQ \neq NULL$ do
7:      $n := WQ$.dequeue();
8:      if $n$ is concat then
9:          $tmp$ : = CONCATSUMMARY($S[n.p], S[n.s]$);
10:     else
11:         $tmp$ : = $\bigcup_{n' \in Pred(n)} S[n']$;
12:     end if
13:     $tmp := (tmp \cup S[n]) \nabla S[n]$;
14:     if $tmp \not\subseteq S[n]$ then
15:         $S[n] := tmp$;
16:         $WQ$.enqueue($Succ(n)$);
17:     end if
18: end while
19: return $\bigcup_{n \in Out} S[n]$;
```

6.4 Alignment

In this section, we discuss how to align an unaligned multi-track DFA M so that $\lambda's$ of M are right justified. First, we show that there are languages recognized by unaligned multi-track DFAs that cannot be recognized by any aligned multi-track DFA.

THEOREM 11. *For any $n \geq 2$, there exists a language L accepted by an n-track DFA M that cannot be converted to any aligned DFA M'.*

Proof: Let $L = \{(a\lambda)^i(cc)^k \mid i, k \geq 1\}$. Clearly, L can be accepted by an unaligned 2-track DFA M. Suppose we can convert M to an aligned 2-track DFA M'. Let M' have s states. Consider the string $w = (ac)^s(c\lambda)^s$. Then w is accepted by M'. Then there exist $i, k \geq 0$ and $j \geq 1$ such that w decomposes into $w = (ac)^i(ac)^j(ac)^k(c\lambda)^s$, where $i + j + k = s$, and $(ac)^i(ac)^{mj}(ac)^k(c\lambda)^s$ is accepted by M' for every $m \geq 0$. Let $m = 2$. Then $w' = (ac)^i(ac)^{2j}(ac)^k(c\lambda)^s$ is accepted by M'. But now, the first track of w' contains the string $a^{s+j}c^s$, and the second track contains c^{s+j}. Since $j \geq 1$, this is a contradiction since the number of c's in the first track is less than the number of c's in the second track. ∎

Since the above result shows that precise alignment is not possible in general, we propose an approximate *k-alignment* construction. Given an unaligned multi-track DFA M and a bound k, we construct M' that accepts an *over* or *under* approximation of $L(M)$ based on k. We associate a bounded FIFO queue ϱ (up to size k) with the states of M' to record the symbols seen on the track that is being aligned when a transition that contains the symbol λ for that track is taken. Later, when a non-λ symbol is seen on that track, it has to match the symbol that is at the head of the queue if the queue is not empty. An *active* queue $(+)$ can enqueue and dequeue once. After dequeuing, it becomes an *inactive* queue $(-)$ that can only dequeue. Initially, all queues are *active*.

M' simulates M. The idea is to output possible characters while encountering λ during the construction. The output word is kept in a queue in each state. Upon seeing a character $a \neq \lambda \in \Sigma$, we set the queue to *inactive*, and start to output λ and match the contents of the queue against the seen characters until the queue is empty. A word is accepted if the queue is empty and M' is in a final state of M.

During the construction, if no queue exceeds size k, then we say M is *k-alignable*, and the construction returns the precise aligned M' such that $L(M') = L(M)$. If M is not *k-alignable*, the *under*-approximation construction rejects all words that cause a queue to exceed k and returns an M' such that $L(M') \subseteq L(M)$, while the *over*-approximation construction accepts those words that partially match the contents of the queue (up to size k) and returns an M' such that $L(M) \subseteq L(M')$. The precision improves when we increase k.

Let $M = \langle Q, \Sigma^n, \delta, I, F\rangle$ and $\Sigma^n \subseteq (\Sigma \cup \{\lambda\}) \times \ldots \times (\Sigma \cup \{\lambda\})$. For $\alpha \in \Sigma^n$, $\alpha[i] \in \Sigma \cup \{\lambda\}$ denotes the i^{th} character of α and $\alpha[i := a]$ denotes $\alpha' \in \Sigma^n$ such that $\alpha'[i] = a$ and $\forall i \neq j, \alpha'[j] = \alpha[j]$. We align one track of M at a time. To align M completely, we iteratively align each track. Given a bound k and a track i, we construct M' such that the track i is aligned in M'. We assume that there is a sink state and all unspecified transitions go to the sink state. Let $\varrho_\perp$ be an empty queue and $*$ denote $+$ or $-$. We construct $M' = \langle Q', \Sigma^n, \delta', I', F'\rangle$ as follows:

- $Q' \subseteq Q \times Q_{queue}$, where $Q_{queue} \subseteq \{+, -\} \times \Sigma^k$.
- $I' = (I, (+, \varrho_\perp))$.
- $F' = \{(q, (*, \varrho_\perp)) \mid q \in F)\}$

For each $\delta(q, \alpha) = q'$,

- if $\alpha[i] \in \Sigma$,
  - $\delta'((q, (*, \varrho_\perp)), \alpha) = (q', (-, \varrho_\perp))$,
  - $\delta'((q, (*, \varrho)), \alpha[i := \lambda]) = (q', (-, \varrho')$, if $\alpha[i] = \varrho$.head and $\varrho' = \varrho$.dequeue.
- if $\alpha[i] = \lambda$,
  - $\delta'((q, (-, \varrho_\perp)), \alpha) = (q', (-, \varrho_\perp))$,
  - $\delta'((q, (+, \varrho)), \alpha) = (q', (-, \varrho))$,

  - $\forall a \in \Sigma_\varrho$, $\varrho' = \varrho$.enqueue(a) and $|\varrho'| \leq k$, $\delta'((q, (+, \varrho)), \alpha[i := a]) = (q', (+, \varrho'))$.

$\Sigma_\varrho \subseteq \Sigma$ is the set of characters that can be reached in track i after seeing the sequence of symbols stored in ϱ. Precisely, let $M_i = \langle Q_i, \Sigma, \delta, I_i, F_i \rangle$ accept $\{\hat{w}[i] \mid w \in L(M)\}$, then $\Sigma_\varrho = \{a \mid q' \neq sink, \delta_i(I, \varrho a) = q'\}$. Using Σ_ϱ (instead of Σ) prevents the construction from adding useless states that will end up transitioning to the sink state.

The above construction returns an *under*-approximation if M is not *k-alignable*. To return an *over*-approximation, we make the following modifications. We first add two extra states to the queue, $\{e, e'\}$, to denote that the queue capacity has been exceeded. After the queue capacity is exceeded, we will stop enqueuing symbols to the queue when we see λ. We continue to match and dequeue when we see $a \in \Sigma$ until the queue is empty. In both cases, we can output arbitrary character $a \in \Sigma$ or λ (e), but once we output λ, we can only output λ thereafter (e').

For each $\delta(q, \alpha) = q'$,

- if $\alpha[i] \in \Sigma$,
  - $\delta'((q, (\{e, e'\}, \varrho_\perp)), \alpha[i := \lambda]) = (q', (e', \varrho_\perp))$,
  - $\forall a \in \Sigma, \delta'((q, (e, \varrho_\perp)), \alpha[i := a]) = (q', (e, \varrho_\perp))$,
  - if $\alpha[i] = \varrho$.head and $\varrho' = \varrho$.dequeue.
    - $\delta'((q, (\{e, e'\}, \varrho)), \alpha[i := \lambda]) = (q', (e', \varrho')$,
    - $\forall a \in \Sigma, \delta'((q, (e, \varrho)), \alpha[i := a]) = (q', (e, \varrho'))$,
- if $\alpha[i] = \lambda$,
  - if $|\varrho| = k$,
    - $\forall a \in \Sigma, \delta'((q, (+, \varrho)), \alpha[i := a]) = (q', (e, \varrho))$,
    - $\delta'((q, (+, \varrho)), \alpha) = (q', (e', \varrho))$,
  - $\delta'((q, (\{e, e'\}, \varrho_\perp)), \alpha) = (q', (e', \varrho_\perp))$,
  - $\forall a \in \Sigma, \delta'((q, (e, \varrho)), \alpha[i := a]) = (q', (e, \varrho))$.

7. Implementation

We have implemented the multi-track automaton construction and symbolic reachability algorithms described above in our string analysis tool [13–15].

Since a multi-track DFA contains all information that single-track DFAs have, we can extend our analysis to support complex string operations implemented for single-track DFAs [14]. For example, the single-track `replace`(M_1, M_2, M_3) operation (proposed in [14]) returns a DFA M, so that $L(M) = \{w_1 c_1 w_2 c_2 \ldots w_k c_k w_{k+1} \mid k > 0, w_1 x_1 w_2 x_2 \ldots w_k x_k w_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.

To use single-track automata operations, we implemented two mapping functions between a single-track automaton and a multi-track automaton. Extraction(M, i), takes an n-track DFA M and an index i, and returns a single-track DFA that accepts $\{\hat{w}[i] \mid w \in L(M)\}$. Extension$(M, i, n)$, takes a single-track DFA M, an index i and the number of tracks n, and returns an n-track DFA that accepts $\{w \mid \hat{w}[i] \in L(M), \forall 1 \leq k \leq n, w[k] \in \Sigma^* \lambda^*\}$. The post-images of the complex string functions on an n-track DFA M can be implemented by: (1) extracting single-track DFAs from M, (2) computing post-images on single-track DFAs, (3) extending the resulting DFAs to n-track DFAs and (4) using intersection to get the final post-image.

Consider the following statement X_i := REPLACE(X_1, X_2, X_3). Let M' be the result of Extension(`replace`$(M_1, M_2, M_3), i, n)$, where M_1, M_2, M_3 is the result of Extraction$(M, 1)$, Extraction$(M, 2)$, and Extraction$(M, 3)$, respectively. The post-image of the replacement statement is constructed as $M \downharpoonright_i \cap M'$. Note that the result is an *over*-approximation since $w \in L(M'')$ does not imply that $\hat{w}[i]$ = REPLACE$(\hat{w}[1], \hat{w}[2], \hat{w}[3])$.

8. Experiments

We evaluate our approach against three kinds of benchmarks: (1) Basic benchmarks, (2) SQLCI/XSS benchmarks and (3) MFE benchmarks. Table 1 summarizes the results of using single-track DFAs and multi-track DFAs to analyze these benchmarks.

Basic benchmarks: These examples demonstrate that our approach can prove implicit equality properties of string systems. We wrote two small programs. CheckBranch is similar to the first example from Section 2 but in the else branch ($X_1 \neq X_2$), we assign a constant c to X_1 and then assign the same constant to X_2. We check at the merge point whether $X_1 = X_2$. CheckLoop is similar to the second simple example from Section 2, where we assign X_1 and X_2 the same constant at the beginning, and iteratively append another constant to both in an infinite loop. We check at the end point of the loop whether $X_1 = X_2$. 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$(X_1 = X_2, -)$. While using single-track DFAs, $M = \cap_{i=1,2}$ Extension$(M_i, i, 2)$ is a *composed DFA*. Using multi-track DFAs, we prove the equality property that we fail to prove using single-track DFAs for these benchmarks as shown in Table 1. Although these benchmarks are simple, to the best of our best knowledge, there are no other string analysis tools that can prove the assertions in these benchmarks.

SQLCI/XSS benchmarks: In the second set, we model branch conditions while checking SQL Command Injection (SQLCI) and Cross-Site Scripting (XSS) attacks against known vulnerable Web applications. 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 against the given attack pattern. To identify SQLCI/XSS attacks, we check intersection emptiness against all possible values of the input of the 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 [14], using multi-track DFAs, we can interpret branch conditions, e.g., `$www=$url`, that cannot be precisely expressed using single-track DFAs. Hence, the result obtained using multi-track DFAs is a more precise representation of reachable values of the input of the sensitive function. However, for these benchmarks added precision did not change the results since the single-track analysis does not generate any false positives. These results are still valuable in demonstrating the increase in analysis cost when multi-track DFAs are used instead of single-track DFAs.

MFE benchmarks: In the last set, we show that the precision that is obtained using multi-track DFAs can help us in removing false positives 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 (include, fopen, etc) 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. For instance, in pblguestbook.php distributed with Pblguestbook-1.32, one possible violation is that `$_GET['type']` is `A` but the retrieved file is `pblguestbook_back_up_B.txt`. 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

	Single-track				Multi-track			
Benchmark, file (line)	Result	DFAs/ Composed DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Result	DFA state(bdd)	Time user+sys(sec)	Mem (kb)
CheckBranch	false	15(107), 15(107) / 33(477)	0.027 + 0.006	410	true	14(193)	0.070 + 0.009	918
CheckLoop	false	6(40), 6(40) / 9(120)	0.022+0.008	484	true	5(60)	0.025+0.006	293
MyEasyMarket-4.1, trans.php (218)	vul	2(20), 9(64), 17(148)	0.010+0.002	444	vul	65(1629)	0.195+0.150	1231
PBLguestbook-1.32, pblguestbook.php(1210)	vul	9(65), 42(376)	0.017+0.003	626	vul	49(1205)	0.059+0.006	4232
Aphpkb-0.71, saa.php(87)	vul	11(106), 27(226)	0.032+0.003	838	vul	47(2714)	0.153+0.008	2684
BloggIT 1.0, admin.php (23)	vul	53(423), 79(633)	0.062+0.005	1696	vul	79(1900)	0.226+0.003	2826
PBLguestbook-1.32, pblguestbook.php(536)	vul	2(8), 28(208) / 56(801)	0.027+0.003	621	no	50(3551)	0.059+0.002	1294
MyEasyMarket-4.1, prod.php (94)	vul	2(20), 11(89) / 22(495)	0.013+0.004	555	no	21(604)	0.040+0.004	996
MyEasyMarket-4.1, prod.php (189)	vul	2(20), 2(20) / 5(113)	0.008+0.002	417	no	3(276)	0.018+0.001	465
php-fusion-6.01, db_backup.php (111)	vul	24(181), 2(8), 25(188) / 1201(25949)	0.226+0.025	9495	no	181(9893)	0.784+0.07	19322
php-fusion-6.01, forums_prune.php (28)	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. line: the line number of the checked point.

segments. Upon termination, we report that the file function is vulnerable 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. Using multi-track DFA analysis we are able to show that this type of vulnerability does not exist in these programs. However, when we use single-track automata based string analysis every instance generates a false positive since single-track DFA are not capable of representing relationships among variables which is necessary to verify these properties.

Discussion: We have shown that multi-track DFAs can handle problems that cannot be handled by multiple single-track DFAs. At the same, we also observed that string analysis based on multi-track DFAs uses more time and memory than the single-track DFA analysis. 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 the use of symbolic DFA representation (provided by the MONA DFA library [2]), in which transition relations of the DFAs are represented as Multi-terminal Binary Decision Diagrams (MBDDs). Using this 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.

9. Conclusion

Many security vulnerabilities are caused by inadequate manipulation of string variables. In this paper, we presented a formal characterization of the string verification problem and showed that it is undecidable. We proposed a conservative symbolic verification approach that computes an over-approximation of the reachable states. Our string analysis uses a single multi-track DFA to represent all possible values of string variables at a given program point. This enables us to check equality properties among string variables and improves the precision of the string analysis. We demonstrated the effectiveness of our approach on several examples.

References

[1] Constantinos Bartzis and Tevfik Bultan. Widening arithmetic automata. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 321–333, 2004.

[2] Morten Biehl, Nils Klarlund, and Theis Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, LNCS 1260*. Springer Verlag, 1997.

[3] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *12th International Conference on Computer Aided Verification*, pages 403–418, 2000.

[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] CVE. Common Vulnerabilities and Exposures. `http://www.cve.mitre.org`.

[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*, pages 87–96, Washington, DC, USA, 2007.

[7] Carl Gould, Zhendong Su, and Premkumar Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings of the 26th International Conference on Software Engineering*, pages 645–654, 2004.

[8] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference*, pages 432–441, 2005.

[9] 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.

[10] Open Web Application Security Project (OWASP). Top ten project. `http://www.owasp.org/`, May 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.

[12] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, 2007.

[13] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. Technical Report 2009-11, Computer Science Department, University of California, Santa Barbara, June 2009.

[14] 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 Software (SPIN 2008)*, pages 306–324, 2008.

[15] Fang Yu, Tevfik Bultan, and Oscar H. Ibarra. Symbolic string verification: Combining string analysis and size analysis. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, pages 322–336, 2009.