Symbolic String Verification: An Automata-based Approach*

Fang Yu, Tevfik Bultan, Marco Cova, and Oscar H. Ibarra

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

Abstract. We present an automata-based approach for the verification of string operations in PHP programs based on symbolic string analysis. String analysis is a static analysis technique that determines the values that a string expression can take during program execution at a given program point. This information can be used to verify that string values are sanitized properly and to detect programming errors and security vulnerabilities. In our string analysis approach, we encode the set of string values that string variables can take as automata. We implement all string functions using a symbolic automata representation (MBDD representation from the MONA automata package) and leverage efficient manipulations on MBDDs, e.g., determinization and minimization. Particularly, we propose a novel algorithm for language-based replacement. Our replacement function takes three DFAs as arguments and outputs a DFA. Finally, we apply a widening operator defined on automata to approximate fixpoint computations. If this conservative approximation does not include any *bad* patterns (specified as regular expressions), we conclude that the program does not contain any errors or vulnerabilities. Our experimental results demonstrate that our approach works quite well in checking the correctness of sanitization operations in real-world PHP applications.

1 Introduction

Unsanitized string variables are a common cause of security vulnerabilities in Web applications. In typical interactive Web applications, user-provided input strings are often used to query back-end databases. If the user input is not properly checked and filtered (i.e., sanitized), the input strings that contain hidden destructive commands can be sent to back-end databases and cause damage. Using the string analysis techniques proposed in this paper, it is possible to automatically verify that a string variable is properly sanitized at a program point, showing that such attacks are not possible.

We present a string analysis technique that computes an over approximation of possible values that a string expression can take at a given program point. We use a deterministic finite automaton (DFA) to represent the set of values string

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

expressions can take. At each program point, each string variable is associated with a DFA. The language accepted by the DFA corresponds to the values that the corresponding string variable can take at that program point.

The string analysis technique we present is a forward reachability computation that uses DFA as a symbolic representation. We use the symbolic DFA representation provided by the MONA DFA library [4], in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs). We iteratively compute an over approximation of the least fixpoint that corresponds to the reachable values of the string expressions. In each iteration, given the current state DFAs for all the variables, we compute the next state DFAs. We present algorithms for next state computation for string operations such as concatenation and language-based replacement. Particularly, we present an algorithm for the language-based replacement operation that computes the DFA for $\textsc{Replace}(M_1, M_2, M_3)$ where M_1, M_2, and M_3 are DFAs that accept the set of original strings, the set of match strings, and the set of replacement strings, respectively.

Our language-based replacement operation is essential to model various built-in functions of PHP language that can be used to perform input validation. These functions provide a general mechanism to scan a string for matches to a given pattern, expressed as a regular expression, and to replace the matched text with a replacement string. As an example of modeling these functions, consider the following statement:

```
$username = ereg\_replace("<script *>", "", $_GET[\"username\"]);
```

The expression _GET["username"] returns the string entered by the user, the ereg_replace call replaces all matches of the search pattern with the empty string, and the result is assigned to the variable username. This statement can be modeled by our language-based replacement operation, where M_1 accepts arbitrary strings, M_2 accepts the set of strings that start with <script followed by zero or more spaces and terminated by the character >, and M_3 accepts the empty string.

We believe that we are the first to extend the MONA automata package to analyze these complex string operations on real programs. In addition to computing the language-based replacement operation, another difficulty is implementing these string operations without using the standard constructions based on the ϵ-transitions, since the MBDD-based automata representation used by MONA does not allow ϵ-transitions. We model non-determinism by extending the alphabet with extra bits and then project them away using the on-the-fly subset construction algorithm provided by MONA. We apply the projection one bit at a time, and after projecting each bit away, we use the MBDD-based automata minimization to reduce the size of the resulting automaton.

Since DFAs can represent infinite sets of strings, the fixpoint computations are not guaranteed to converge. To alleviate this problem, we use the automata widening technique proposed by Bartzis and Bultan [3] to compute an over-approximation of the least fixpoint. Briefly, we merge those states belonging to the same equivalence class identified by certain conditions. This widening operator was originally proposed for automata representation of arithmetic constraints

but the intuition behind it is applicable to any symbolic fixpoint computation that uses automata.

We implemented the proposed string analysis technique for PHP programs. PHP is a scripting language which is widely used in implementing interactive Web applications. Our experiments show that the proposed symbolic analysis technique works quite well and can be used to prove the correctness of sanitization in real-world PHP applications.

An Example: Consider the PHP program fragment below which demonstrates a vulnerability from a guestbook application called `PBLguestbook-1.32`:

```
1:   foreach ($_POST as $name => $value) {
2:     if ($name != 'process' && $name != 'password2') {
3:       $count++;
4:       $result .= "`$name` = '$value'";
5:       if ($count <= $numofparts)
6:         $result .= ", ";
7:     }
8:   }
9:   $query = "UPDATE `pblguestbook_config` SET $result";
10:  mysql_query($query);
```

This program fragment traverses the input strings entered by the user (which are stored in the `_POST` array) in a loop (lines 1-8) and constructs a query string by accumulating them (by concatenating them to the `result` variable). This query is then sent to the back-end database (line 10).

This program shows an example of a SQL injection vulnerability. Input strings are concatenated in the loop at lines 1-8 to form the string used to query the application's database. Since no sanitization is performed, an attacker can modify the query, for example, by injecting a parameter with value `'; DROP DATABASE #`. In this case, the SQL string sent to the database will be `UPDATE `pblguestbook_config` SET `name` = ''; DROP DATABASE #'`. Note that the '`;`' character separates distinct queries and the '`#`' character starts a comment. Therefore, if the database allows the execution of multiple queries, it will execute the legitimate query intended by the developer and the injected query that drops the entire database. The vulnerability can be fixed by adding a sanitization step on the input parameters before the query string is formed.

A properly sanitized version of this program fragment would be:

```
1:    foreach ($_POST as $name => $value) {
1.1:    $name = preg_replace("/[^a-zA-Z0-9]/", "", $name);
1.2:    $value = preg_replace("/'/", "", $value);
2:      if ($name != 'process' && $name != 'password2') {
3:        $count++;
4:        $result .= "`$name` = '$value'";
5:        if ($count <= $numofparts)
6:          $result .= ", ";
7:      }
8:    }
9:    $query = "UPDATE `pblguestbook_config` SET $result";
10:   mysql_query($query);
```

The sanitization is achieved in lines 1.1 and 1.2 by deleting potentially problematic characters in the variables `$name` and `$value`, hence preventing the presented SQL command injection attack. We analyzed both the vulnerable and the sanitized versions of this program fragment using our string analysis tool. Our string analysis tool constructed a DFA that gives an over-approximation of the string values that the variable `query` can take at line 10. We wrote a regular expression characterizing strings that can be used for SQL command injection and converted it to a DFA. (Note that these types of attack DFAs can be constructed once and stored in a library. They do not have to be specified separately for each program that is being analyzed). Then, we checked if the intersection of the language recognized by the DFA for the `query` variable at line 10, and the DFA characterizing the SQL command injection attack is empty. When we applied our analysis to the vulnerable program fragment shown above, our string analysis tool reported that the intersection is not empty, i.e., the program fragment might be vulnerable. However, when we applied our analysis to the sanitized version, our tool reported that the intersection is empty, proving that the variables are properly sanitized.

It is worthwhile to note some of the challenges in analyzing the example given above. First, in order to prove that the variables are properly sanitized, we need to statically interpret the replacement function `preg_replace` with reasonable precision. Second, our fixpoint computation has to converge even though the above program fragment contains a loop. We are able to handle both of these challenges by 1) proposing and implementing a novel language-based replacement operation and 2) using an automata widening operator. Note that, for the sanitized program fragment, the fixpoint computation without widening will not converge. Moreover, a naive over-approximation, that sets the values of the variables that are updated in a loop to all possible strings, will not be a tight enough approximation to verify the sanitized program fragment.

The rest of the paper is organized as follows. In Section 2, we describe our symbolic string analysis algorithm. In Section 3, we describe the implementation of the closure, concatenation and replacement operations. In Section 4, we discuss the widening operation. In Section 5, we summarize our experiments. In Section 6, we discuss the related work, and, in Section 7, we conclude the paper.

2 Automata-based String Analysis

Most of the string manipulation operations performed in real-world applications can be reduced to the following four operations:

- *assignment*: assigns the current string value of a variable to another variable (the assignment operator in PHP is "`=`");
- *concatenation*: concatenates two string variables and/or constants (the concatenation operation in PHP is "`.`");
- *replacement*: replaces the parts of a string that match the given pattern with the given replacement string (there are several string replacement func-

tions in PHP such as `htmlspecialchars`, `tolower`, `toupper`, `str_replace`, `trim`, `preg_replace` and `ereg_replace`, and they can all be converted to this form).
- *restriction*: restricts the value of a string variable based on a branch condition.

The first step of string analysis is to construct a control flow graph (CFG) that only contains string variables and operations on string variables. We define a CFG as a tuple (V, S, E) where V is the set of string variables, S is the set of statements and $E \subseteq S \times S$ is the set of control flow edges. Each statement $s \in S$ could be one of the following operations: *null, assign, concat, replace, restrict, input.* The *null* operation represents the statements that do not influence the string variables and, hence, have been removed. We use *assign* to denote the assignment of a string constant or a variable to a string variable. We use *concat* to denote the assignment operations that assign the concatenation of two string constants and/or variables to a string variable. We use *replace* to denote the assignment of a string value computed by a replacement operation to a string variable. We use *restrict* to denote the restriction of a string value in order to model branch conditions. For instance, considering a branch condition $v = e$, where e is a regular expression, we add $restrict(v, e)$ at the beginning of the truth branch and $restrict(v, \bar{e})$ at the beginning of the false branch where $\bar{e}$ indicates to restrict the string values of v to the complement set of e. A similar idea has been discussed in [15]. Finally, we use *input* to denote a read operation, where a string variable is assigned a value provided by a user.

Automata Operations: In order to implement the automata-based string analysis, we implement the following operations:

- Construct(regexp e): Returns a DFA M, $L(M) = \{w \mid w \in L(e)\}$.
- Closure(DFA M_1): Returns a DFA M, $L(M) = \{w_1 w_2 \ldots w_k \mid k > 0, \forall i, 1 \leq i \leq k, w_i \in L(M_1)\}$.
- Concat(DFA M_1, DFA M_2): Returns a DFA M, $L(M) = \{w_1 w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.
- Replace(DFA M_1, DFA M_2, DFA M_3): Returns a DFA M, $L(M) = \{w_1 c_1 w_2 c_2 \ldots w_k c_k w_{k+1} \mid k > 0, w_1 x_1 w_2 x_2 \ldots w_k x_k w_{k+1} \in L(M_1), \forall_i,\ x_i \in L(M_2),\ w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.
- Union(DFA M_1, DFA M_2) : Returns a DFA M, $L(M) = L(M_1) \cup L(M2)$.
- Intersect(DFA M_1, DFA M_2): Returns a DFA M, $L(M) = L(M_1) \cap L(M_2)$.
- Widening(DFA M_1, DFA M_2): Returns a DFA M, $L(M) \supseteq L(M_1) \cup L(M_2)$.
- EquCheck(DFA M_1, DFA M_2): Checks whether $L(M_1) = L(M_2)$.
- EmpCheck(DFA M): Checks whether $L(M) = \emptyset$.
- Empty(): Returns a DFA which does not accept any string.
- Universal(): Returns a DFA which accepts all the strings.

String Analysis Algorithm: The string analysis algorithm, takes a CFG, a program point, a string variable and an attack pattern as input. It computes $|V| \times |S|$ DFAs, where the DFA (v, s) accepts the language that corresponds to all the string values that the variable v can take at the program point s during any

```
Input: (V, S, E), attackpattern, statement, variable
DFA attack := CONSTRUCT(attackpattern)
DFA old[1...|V|][1...|S|], new[1...|V|][1...|S|], temp[1...|V|]
for each v ∈ V, s ∈ S, old[v][s] := EMPTY(), new[v][s] := EMPTY()
repeat
    for each v ∈ V, s ∈ S, old[v][s] := new[v][s]
    for each s ∈ S
        for each v ∈ V, temp[v] := EMPTY()
            for each (s', s) ∈ E, temp[v] := UNION(temp[v], old[v][s'])
        for each v ∈ V, new[v][s] := temp[v]
        switch s.type
            case null skip
            case read // v := get input
                new[v][s] := UNIVERSAL()
            case assign // v := v1
                new[v][s] := temp[v1]
            case concat // v := concat(v1, v2)
                new[v][s] := CONCAT(temp[v1],temp[v2])
            case replace // v := replace(v1, e, c)
                where e is a regular expression and c is a string.
                DFA t1 := CONSTRUCT(e), DFA t2 := CONSTRUCT(c)
                new[v][s] := REPLACE(temp[v1], t1, t2)
            case restrict // restrict(v,e)
                DFA t1 := CONSTRUCT(e)
                new[v][s] := INTERSECT(old[v][s],t1)
    for each v ∈ V, s ∈ S, old[v][s] := WIDENING(old[v][s],new[v][s])
until (for all v, s, EQUCHECK(old[v][s], new[v][s]))
if (EMPCHECK(INTERSECT(new[variable][statement], attack))) then VER else ERR
```

Fig. 1. String analysis algorithm

program execution. We compute these DFA using a least fixpoint computation as shown in Figure 1. Since the lattice is infinite, it might not be possible to reach the least fixpoint using an iterative algorithm. To tackle this problem, we apply the automata widening operator in [3] to our analysis. Following Bartzis and Bultan's results, we characterize a set of languages that this widening operator can result in the precise fixed point. Our string analysis algorithm returns VER if it is not possible for the input variable to have a string value that matches the attack pattern at the given program point; however, it may yield a false alarm while it returns ERR.

Symbolic Automata Representation: We use the DFA library of MONA [4] to implement the string operations listed above. In MONA, transition relations of DFA are symbolically represented using Multi-terminal Binary Decision Diagrams (MBDDs). A MBDD is a BDD with multiple roots and multiple leaves. In MONA's DFA representation, each state of the DFA is a root and points to a BDD node, and each leaf value is a state of the DFA. Given the current state and a symbol $a \in B^k$, where B^k is alphabet of bit vectors of length k, one can

find the next state by following the BDD nodes according to the bit vector of a from the BDD node pointed by the current state. We use a 7-bit vector, i.e., B^7, as our alphabet representing the binary value of ASCII symbols, e.g., for the ASCII symbol 'a', the ASCII code is 97 which is represented as '1100001' in our encoding.

The MONA DFA library provides efficient implementations of standard automata operations. These operations include product, project and determinize, and minimize [4]. The product operation takes the Cartesian product of the states of the two input automata. We use the product operation to implement the intersection and union operations. The project and determinize operation, denoted as $\textsc{Project}(M, i)$, where $1 \leq i \leq k$, converts a DFA M recognizing a language L over the alphabet B^k, to a DFA M' recognizing a language L' over the alphabet B^{k-1}, where L' is the language that results from applying the tuple projection on the ith bit to each symbol of the alphabet. The process consists of removing the ith track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction.

3 String Operations on Automata

In this section, we describe how to implement the closure, concatenate and replace operations. Since we use MBDD representation for DFA, we are not able to introduce ϵ-transitions. Instead, to avoid the non-determinism introduced by these operations, we extend the alphabet by adding extra bits, and then use projection to map the resulting DFA to the original alphabet.

A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F \rangle$ where Q is a finite set of states, q_0 is the initial state, $\Sigma \subseteq B^k$ is the alphabet, where each symbol is encoded as a k-bit string. $F : Q \rightarrow \{-, +\}$ is a mapping function from a state to its status. Given a state $q \in Q$, q is an accepting state if $F(q) = +$. $\delta : Q \times \Sigma \rightarrow Q$ is the transition relation. A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $F(q) = -$. In the following sections, we assume that for all unspecified pairs (q, α), $\delta(q, \alpha)$ goes to a *sink* state. In the constructions below, we also ignore the transitions that lead to a sink state.

Given $\alpha \in B^k$, we use $\alpha 0$ or $\alpha 1 \in B^{k+1}$ to denote the bit string that is α appended with '0' or '1'. For instance, if α is '110011' then $\alpha 0$ is '1100110'.

Closure: The DFA M is a closure-DFA of the DFA M_1, if $L(M) = \{\, w_1 w_2 \ldots w_k \mid \exists k > 0, \forall 1 \leq i \leq k, w_i \in L(M_1)\}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, its closure M can be constructed by first constructing an intermediate DFA $M^{'} = \langle Q_1, q_{10}, \Sigma^{'}, \delta^{'}, F_1 \rangle$ as:

- $\Sigma^{'} = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta^{'}(q, \alpha 0) = q'$, if $\delta_1(q, \alpha) = q'$.
- $\forall q \in Q_1, \delta^{'}(q, \alpha 1) = q'$, if $F_1(q) = +$ and $\delta_1(q_{10}, \alpha) = q'$.

Then, $M = \textsc{Project}(M^{'}, k+1)$ is the closure of M_1.

Since M_1 is a DFA, the project operation requires the subset construction only when there exists $q \in Q_1, F_1(q) = +$, and $\exists \alpha, q', q^{''}, \alpha \in \Sigma, q', q^{''} \in Q_1, q' \neq q^{''}, \delta_1(q, \alpha) = q', \delta_1(q_{10}, \alpha) = q^{''}$.

Concatenation: The DFA M is a concatenation-DFA of the DFA M_1 and M_2, if $L(M) = \{w_1 w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.

Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$ and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, the concatenation-DFA M can be constructed as follows. Without loss of generality, we assume that $Q_1 \cap Q_2$ is empty. We first construct an intermediate DFA $M^{'} = \langle Q^{'}, q_{10}, \Sigma^{'}, \delta^{'}, F^{'} \rangle$, where

- $Q^{'} = Q_1 \cup Q_2$
- $\Sigma^{'} = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\}$
- $\forall q, q' \in Q_1, \delta^{'}(q, \alpha 0) = q'$, if $\delta_1(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta^{'}(q, \alpha 0) = q'$, if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_1, \delta^{'}(q, \alpha 1) = q'$, if $F_1(q) = +$ and $\exists q' \in Q_2, \delta_2(q_{20}, \alpha) = q'$
- $\forall q \in Q_1, F^{'}(q) = +$, if $F_1(q) = +$ and $F_2(q_{20}) = +$; $F^{'}(q) = -$, o.w.
- $\forall q \in Q_2, F^{'}(q) = F_2(q)$.

Then, $M = \textsc{Project}(M^{'}, k+1)$. Again, since both M_1 and M_2 are DFA, the subset construction happens only when there exists $q \in Q_1, F_1(q) = +$ such that $\exists \alpha, q', q^{''}, \alpha \in \Sigma, q' \in Q_1, q^{''} \in Q_2, \delta_1(q, \alpha) = q', \delta_2(q_{20}, \alpha) = q^{''}$.

Replacement: A DFA M is a replaced-DFA of a DFA tuple (M_1, M_2, M_3), if and only if $L(M) = \{w \mid k > 0, w_1 x_1 w_2 \dots w_k x_k w_{k+1} \in L(M_1), w = w_1 c_1 w_2 ... w_k c_k w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k+1, w_i \notin \{w_1' x' w_2' \mid x' \in L(M_2), w_1', w_2' \in \Sigma^*\}\}$.

This definition requires that all occurrences of matching sub-strings in a word are replaced. The intuition of the implementation of this language-based replacement is that we first insert marks into automata, then identify matching sub-strings by intersection of automata, and finally construct the final automaton by replacing these matching sub-strings.

We consider a new alphabet $\bar{\Sigma} = \{\bar{\alpha} | \alpha \in \Sigma\}$, and let $\bar{x}$ denote a new string in which we add bar to each character in x. Assume that M_1, M_2, M_3 have the same alphabet Σ, where $\sharp_1, \sharp_2 \notin \Sigma$, and $\forall \alpha \in \Sigma, \bar{\alpha} \notin \Sigma$. We define $M_1^{'}$, M_2' and M as follows, and claim that M accepts the same language as the replaced-DFA of the tuple (M_1, M_2, M_3).

- $M_1^{'}$, where $L(M_1^{'}) = \{w' \mid k > 0, w = w_1 x_1 w_2 \dots w_k x_k w_{k+1} \in L(M_1), w' = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}\}$.
- $M_2^{'}$, where $L(M_2^{'}) = \{w' \mid k > 0, w' = w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), \forall 1 \leq i \leq k+1, w_i \in L(M_h)\}$, where $L(M_h)$ is the set of strings which do not contain any substring in $L(M_2)$. The language $L(M_h)$ is defined as the complement set of $\{w_1 x w_2 \mid x \in L(M_2), w_1, w_2 \in \Sigma^*\}$.
- M, where $L(M) = \{w \mid k > 0, w_1 \sharp_1 \bar{x_1} \sharp_2 w_2 \dots w_k \sharp_1 \bar{x_k} \sharp_2 w_{k+1} \in L(M_1^{'}) \cap L(M_2^{'}), w = w_1 c_1 w_2 \dots w_k c_k w_{k+1}, \forall 1 \leq i \leq k, c_i \in L(M_3)\}$.

To distinguish the original and bar alphabets, we append an extra bit to α so that α is $\alpha 0$ and $\bar{\alpha}$ is $\alpha 1$. Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$, $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$, and $M_3 = \langle Q_3, q_{30}, \Sigma, \delta_3, F_3 \rangle$, the process to construct a replaced-DFA M can be decoupled into the following steps:

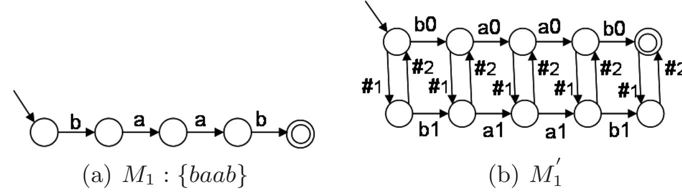
1. Construct $M_1{}'$ from M_1,
2. Construct $M_2^{'}$ from M_2,
3. Generate $M^{'}$ as the intersection of $M_1^{'}$ and $M_2^{'}$,
4. Construct $M^{''}$ from $M^{'}$ where the strings that appear between $\sharp_1$ and $\sharp_2$ are replaced by words in $L(M_3)$, and
5. Generate M from $M^{''}$ by projection.

We formally describe the implementation of these steps below. As a running example, we use $L(M_1) = \{baab\}$, $L(M_2) = a^+$ (M_2 accepts the language $\{a, aa, aaa, \ldots\}$) and $L(M_3) = \{c\}$ or $L(M_3) = \{\epsilon\}$. Let $|M|$ denote the number of states of M. An upper bound for each intermediate automaton before projection and minimization is also described.

Step 1: $M_1^{'} = \langle Q_1^{'}, q_{10}, \Sigma^{'}, \delta_1^{'}, F_1^{'} \rangle$ is constructed from M_1, where

- $Q_1^{'} = Q_1 \cup Q_{1'}$, $Q_{1'}$ is the duplicate of Q_1. For all $q \in Q_1$, there is a one to one mapping $q' \in Q_{1'}$.
- $\Sigma^{'} = \{\alpha 0 \mid \alpha \in \Sigma\} \cup \{\alpha 1 \mid \alpha \in \Sigma\} \cup \{\sharp_1, \sharp_2\}$
- $\delta_1^{'}(q_1, \alpha 0) = q_2$ and $\delta_1^{'}(q_{1'}, \alpha 1) = q_{2'}$, if $\delta_1(q_1, \alpha) = q_2$
- $\forall q_1 \in Q_1, \delta_1^{'}(q_1, \sharp_1) = q_{1'}$ and $\delta_1^{'}(q_{1'}, \sharp_2) = q_1$
- $\forall q \in Q_1, F_1^{'}(q) = F_1(q)$ and $\forall q \in Q_{1'}, F_1^{'}(q) = 0$.

An example for constructing $M_1^{'}$ from M_1, where $L(M_1) = \{baab\}$, is given in Fig 2. $|M_1^{'}|$ is bounded by $2|M_1|$.



(a) $M_1 : \{baab\}$ (b) $M_1^{'}$

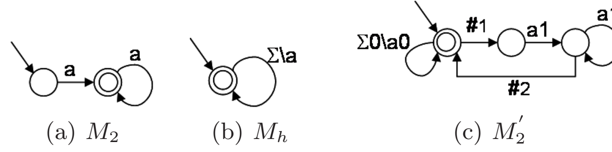
Fig. 2. Constructing $M_1^{'}$ from M_1

Step 2: To construct $M_2^{'}$, we first construct M_h which accepts the complement set of $\{w_1 x w_2 \mid w_1, w_2 \in \Sigma^*, x \in L(M_2)\}$. For instance, as shown in Fig 3(b), for $L(M_2) = a^+$, M_h is the DFA that accepts $(\Sigma \setminus \{a\})^*$. Let M_* be the DFA accepting Σ^*. M_h can be constructed by NEGATE(CONCAT(CONCAT(M_*, M_2), M_*)). We obtain the DFA in Fig 3(b) by applying this construction with minimization.

Assume $M_h = \langle Q_h, q_{h0}, \Sigma, \delta_h, F_h \rangle$, and $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$. $M_2^{'} = \langle Q_2^{'}, q_{h0}, \Sigma^{'}, \delta_2^{'}, F_2^{'} \rangle$ can then be constructed as:

- $Q_2^{'} = Q_h \cup Q_2$
- $\Sigma^{'} = \{\alpha 0 \mid \forall \alpha \in \Sigma\} \cup \{\alpha 1 \mid \forall \alpha \in \Sigma\} \cup \{\sharp_1, \sharp_2\}$
- $\forall q, q' \in Q_h, \delta_2^{'}(q, \alpha 0) = q'$, if $\delta_h(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta_2^{'}(q, \alpha 1) = q'$, if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_h, \delta_2^{'}(q, \sharp_1) = q_{20}$ if $F_h(q) = +$
- $\forall q \in Q_2, \delta_2^{'}(q, \sharp_2) = q_{h0}$ if $F_2(q) = +$
- $\forall q \in Q_h, F_2^{'}(q) = F_h(q)$ and $\forall q \in Q_2, F_2^{'}(q) = -$.

The corresponding $M_2^{'}$ for our example is shown in Fig 3(c). $|M_2^{'}|$ is bounded by $|M_h| + |M_2|$, where $|M_h|$ is bounded by $|M_2| + 2$.



(a) M_2 (b) M_h (c) $M_2^{'}$

Fig. 3. Constructing $M_2^{'}$ from M_2 and M_h

Step 3: $M^{'} = \langle Q^{'}, q_0^{'}, \Sigma^{'}, \delta^{'}, F^{'} \rangle$ is generated as the intersection of $M_1^{'}$ and $M_2^{'}$ based on production. The example $M^{'}$ is shown in Fig 4 (a). $|M^{'}|$ is bounded by $|M_1^{'}| \times |M_2^{'}|$.

Step 4: Before we construct $M^{''}$ from $M^{'}$, we first introduce a function $reach : Q^{'} \rightarrow 2^{Q^{'}}$, which maps a state to all its $\sharp$-reachable states in $M^{'}$. We say q' is $\sharp$-reachable from q if there exists a sequence $q, q_1, \ldots, q_n, q'$ so that (1) $n \geq 1$, (2) $\delta^{'}(q, \sharp_1) = q_1$, (3) $\delta^{'}(q_n, \sharp_2) = q'$, and (4) $\forall 0 < i < n, \delta^{'}(q_i, x) = q_{i+1}$, where $x \in \{\alpha 1 \mid \forall \alpha \in \Sigma\}$. For instance, in Fig 4 (a), one can find that $reach(i) = \{j, k\}$ and $reach(j) = \{k\}$. Intuitively, one can think that each pair (q, q'), where $q' \in reach(q)$, identifies a word in $L(M_2)$.

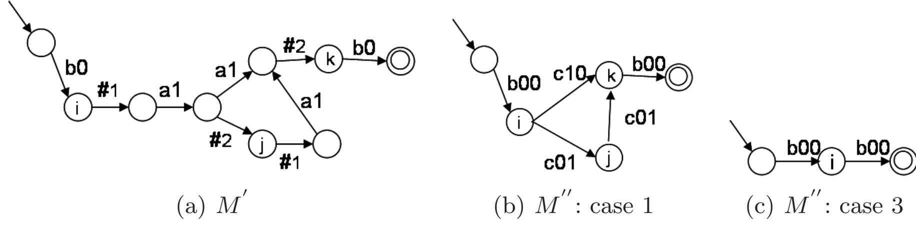
Our goal is, for each $q' \in reach(q)$, inserting paths between q and q' that recognize all words in $L(M_3)$. If there exist $q', q^{''} \in reach(q)$ and $q' \neq q^{''}$, this insertion will cause nondeterminism. To tackle this problem, as we did in the construction of closure and concatenation, we add extra bits to the alphabet and later project them away. Assume n is the maximum size of $reach(q)$ for all $q \in Q^{'}$. We need at most $\lceil log(n+1) \rceil$ bits to be added to the alphabet so that the construction can result in a DFA. Let $P = \{q \mid q \in Q^{'}, reach(q) > 0\}$. Let $m = \lceil log(n+1) \rceil$, where n is the maximum size of $reach(q)$ for all $q \in P$. Let m_q be an m-bit string. For $\alpha \in B^k$, $\alpha m_q \in B^{k+m}$ is a string in which m_q is appended to α. Let m_0 be an m-bit string of 0s. We assume $\forall q, m_q \neq m_0$, and for any $q \in P$, $m'_q \neq m''_q$ if $q', q'' \in reach(q)$.

The construction of $M^{''}$ depends on $L(M_3)$. We consider the following three cases: (1) M_3 only accepts single characters, i.e., $L(M_3) \subseteq \Sigma$, (2) M_3 only accepts words with more than one character, i.e., $L(M_3) \subseteq \Sigma^+ \setminus \Sigma$, (3) M_3 only accepts the empty string, i.e., $L(M_3) = \{\epsilon\}$.

Case 1: $\forall w \in L(M_3), |w| = 1$. $M^{''} = \langle Q^{'}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{'} \rangle$ is constructed as:

– $\Sigma^{''} \subseteq B^{k+m}$
– $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha 0) = q'$
– $\forall q \in P, \forall q' \in reach(p), \forall \alpha \in L(M_3), \delta^{''}(q, \alpha m_{q'}) = q'$.

In Fig 4(a), $P = \{i, j\}$, $reach(i) = \{j, k\}$ and $reach(j) = k$. Let $L(M_3) = \{c\}$. $M^{''}$ of our example is shown in Fig 4(b). Each symbol is appended with two extra bits, e.g., $\delta(i, c01) = j$ and $\delta(i, c10) = k$. $|M^{''}|$ is bounded by $|M^{'}|$.



(a) $M^{'}$ (b) $M^{''}$: case 1 (c) $M^{''}$: case 3

Fig. 4. Constructing $M^{''}$ from $M^{'}$. $M^{'}$ is the intersection of $M_1{}'$ and $M_2^{'}$

Case 2: $\forall w \in L(M_3), |w| \geq 2$. For each $p \in P$, we construct a copy of M_3 as $M_p = \langle Q_p, q_{p0}, \Sigma, \delta_p, F_p \rangle$. $M^{''}$ is constructed by inserting M_p between p and $reach(p)$.

$M^{''} = \langle Q^{''}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{''} \rangle$, where

– $Q^{''} = Q^{'} \bigcup_{p \in P} Q_p$
– $\Sigma^{''} \subseteq B^{k+m}$
– $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha 0) = q'$
– $\forall p \in P, \forall q \in Q_p, \delta^{''}(q, \alpha m_0) = q'$, if $\delta_p(q, \alpha) = q'$.
– $\forall p \in P, \delta^{''}(p, \alpha m_q) = q$, if $\delta_p(q_{p0}, \alpha) = q$.
– $\forall p \in P, \forall q \in reach(p), \delta^{''}(q', \alpha m_0) = q$, if $\delta_p(q', \alpha) = q^{''}$ and $F_p(q^{''}) = +$.
– $\forall q \in Q^{'}, F^{''}(q) = F^{'}(q)$
– $\forall p \in P, q \in Q_p, F^{''}(q) = -$.

In this case, $|M^{''}|$ is bounded by $|M^{'}| + |M^{'}| \times |M^{'}| \times |M_3|$.

Case 3: $\forall w \in L(M_3), |w| = 0$. We consider this case as *deletion*. Before we start the construction, it is worth to know that for deletion, one may change the argument M_2 to N, where $L(N) = L(M_2)+$ (Kleene plus closure) , and get the same result. We specify this property as follows.

Property 1 Let M=Replace(M_1, M_2, M_3), and $M^{'}$=Replace(M_1, N, M_3), where $L(N) = L(M_2)+$. $L(M) = L(M^{'})$ if $L(M_3) = \{\epsilon\}$.

The correctness comes from the fact that, by construction, if there exists $w \in L(N)$, then there exists $k > 0$, $w = w_1 w_2 \dots w_k$, where $\forall 1 \leq i \leq k, w_i \in L(M_2)$. Since w or any w_i will be deleted after the replacement, using N instead of M_2 yields the same result.

Note that the $\sharp$-reachable states of M' using N is actually the set of reachable closure of the $\sharp$-reachable states of M' using M_2. This facilitates our construction

by taking all deleted pairs into account in one step. In the following construction, without loss of the generality, we assume that the matching strings are accepted by N. N can be constructed from the original M_2 by our closure operation.

$M^{''}$ can then be constructed as $\langle Q^{'}, q_0^{'}, \Sigma^{''}, \delta^{''}, F^{''}\rangle$, where

- $\Sigma^{''} \subseteq B^{k+m}$
- $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha 0) = q'$
- $\forall p \in P, \forall q \in reach(p), \delta^{''}(p, \alpha m_{q'}) = q'$, if $\delta^{'}(q, \alpha 0) = q'$.
- $\forall p \in P, F^{''}(p) = +$, if $\exists q \in reach(p), F^{'}(q) = +$.
- $F^{''}(q) = F^{'}(q)$, o.w.

Let $L(M_3) = \{\epsilon\}$. The result of $M^{''}$ is shown in Fig 4(c). Note that if $M_2 = \{a\}$, we would get the same result. $|M^{''}|$ is bounded by $|M^{'}|$.

Finally, consider M_3 as a general DFA. Replace($M1$, M_2, M_3) can be constructed as the union of the results of the following three operations:

- Replace($M1$, M_2, M_{3_1}), where $L(M_{3_1}) = L(M_3) \cap \Sigma$
- Replace($M1$, M_2, M_{3_2}), where $L(M_{3_2}) = L(M_3) \cap \Sigma^+ \setminus \Sigma$
- Replace($M1$, M_2, M_{3_3}), where $L(M_{3_3}) = L(M_3) \cap \{\epsilon\}$

Our replacement operation is defined in a general case in terms of M_3. For all replacement statements in PHP programs, such as `str_replace`, `preg_replace`, and `ereg_replace`, $L(M_3)$ is a constant string. In our implementation, we determine which type of construction to apply based on the length of this string.

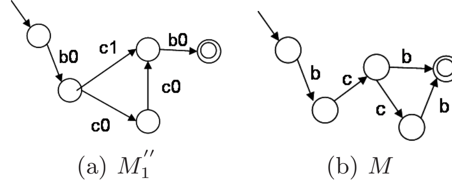
Step 5: Finally, we get M over Σ by iteratively projecting away the extra bits. The subset construction is only applied when needed.

The final DFA M =Replace(M_1, M_2, M_3), where $L(M_1) = \{baab\}$, $L(M_2) = a^+$, and $L(M_3) = \{c\}$, is shown in Fig 5. M accepts $\{bcb, bccb\}$.

In PHP programs, replacement operations such as `ereg_replace` can use different replacement semantics such as *longest match* or *first match*. Our replacement operation provides an over approximation of such more restricted replace semantics. For the example above, in the longest match semantics, M only accepts *bcb*, in which the longest match *aa* is replaced by *c*. In the first match semantics, M only accepts *bccb*, in which two matches *a* and *a* are replaced with *c*. Both of these are included in the result obtained by our replacement operation. This over approximation works well for our benchmarks, and does not raise false alarms. Indeed, we have observed that most statements we encountered yield the same result in the first and longest match semantics, e.g.,`ereg_replace("<script *>","",$_GET["username"]);`, and are precisely modelled by our language-based replacement operation.

4 Widening Automata

In this section, we describe the widening operator we use, which was originally proposed for arithmetic automata by Bartzis and Bultan [3].



(a) $M_1^{''}$ (b) M

Fig. 5. $M_1^{''}$ is PROJECT$(M^{''}, k+2)$, M is PROJECT$(M_1^{''}, k+1)$

Given two finite automata $M = \langle Q, q_0, \Sigma, \delta, F\rangle$ and $M' = \langle Q', q_0', \Sigma, \delta', F'\rangle$, we first define the binary 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

$$\forall w \in \Sigma^*.\ F(\delta^*(q, w)) = + \Leftrightarrow\ F(\delta'^*(q', w)) = +. \tag{1}$$

$$\text{or} \quad q, q' \neq sink \wedge \exists w \in \Sigma^*.\ \delta^*(q_0, w) = q \wedge \delta'^*(q_0', w) = q', \tag{2}$$

where $\delta^*(q, w)$ is defined as the state that M reaches after consuming w starting from state q. In other words, condition 1 states that $q \equiv_\nabla q'$ if $\forall w \in \Sigma^*$, w is accepted by M from q then w is accepted by M' from q', and vice versa. Condition 2 states that $q \equiv_\nabla q'$ if $\exists w \in \sigma$, M reaches state q and M' reaches state q' after consuming w from its initial state. For $q_1 \in Q$ and $q_2 \in Q$ we say that $q_1 \equiv_\nabla q_2$ if and only if

$$\exists q' \in Q'.\ q_1 \equiv_\nabla q' \wedge q_2 \equiv_\nabla q' \quad \vee \quad \exists q \in Q.\ q_1 \equiv_\nabla q \wedge q_2 \equiv_\nabla q \tag{3}$$

Similarly we can define $q_1' \equiv_\nabla q_2'$ for $q_1' \in Q'$ and $q_2' \in Q'$.

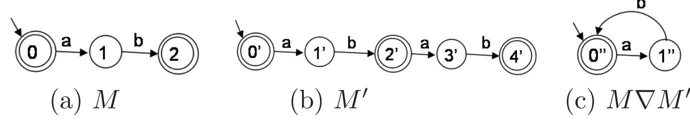
It can be seen that $\equiv_\nabla$ is an equivalence relation. Let C be the set of equivalence classes of $\equiv_\nabla$. We define $M \nabla M' = \langle Q'', q_0'', \Sigma, \delta'', F''\rangle$ by:

$$\begin{aligned}
Q'' &= C \\
q_0'' &= c \ \text{ s.t. } \ q_0 \in c \wedge q_0' \in c \\
\delta''(c_i, \sigma) &= c_j \ \text{ s.t. } \ (\forall q \in c_i \cap Q.\ \delta(q, \sigma) \in c_j \vee \delta(q, \sigma) = sink) \wedge \\
&\qquad\qquad (\forall q' \in c_i \cap Q'.\ \delta'(q', \sigma) \in c_j \vee \delta'(q', \sigma) = sink) \\
F''(c) &= + \ \text{ s.t. } \ \exists q \in F \cup F'.\ q \in c.\ \ F''(c) = - \ \text{ o.w.}
\end{aligned}$$

In other words, the set of states of $M \nabla M'$ is the set C of equivalence classes of $\equiv_\nabla$. Transitions are defined from the transitions of M and M'. The initial state is the class containing the initial states q_0 and q_0'. 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', $L(M) \cup L(M') \subseteq L(M \nabla M')$ [3].

In Fig 6, we give an example for the widening operation. $L(M) = \{\epsilon, ab\}$ and $L(M') = \{\epsilon, ab, abab\}$. The set of equivalence classes is $C = \{q_0'', q_1''\}$, where $q_0'' = \{q_0, q_0', q_2, q_2', q_4'\}$ and $q_1'' = \{q_1, q_1', q_3'\}$. $L(M \nabla M^{'}) = (ab)^*$.

As shown in Fig 1, we use this widening operator iteratively to compute an over-approximation of the least fixpoint that corresponds to the reachable values of string expressions. To simplify the discussion, let us assume a program with



(a) M (b) M' (c) $M \nabla M'$

Fig. 6. Widening automata

a single string variable represented with one automaton M. Let M_i represent the automaton computed at the ith iteration and let I denote the initial value of the string variable. The fixpoint computation will compute a sequence M_0, M_1, ..., M_i, ..., where $M_0 = I$ and $M_i = M_{i-1} \cup post(M_{i-1})$ where the post-condition for different statements is computed as described in Fig 1. We reach the least fixpoint M_j if at some iteration, $M_j = M_{j-1}$. Since we are dealing with an infinite state system, the computation may not converge. In the following, we use M_∞ to denote the least fixpoint.

Given the widening operator, we actually compute an sequence M'_0, M'_1, ..., M'_i, ..., that over-approximates the fixpoint computation where M'_i is defined as: $M'_0 = M_0$, and for $i > 0$, $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}))$. Let M'_∞ denote the least fixpoint of this approximate sequence. Then we have the following result [3]:

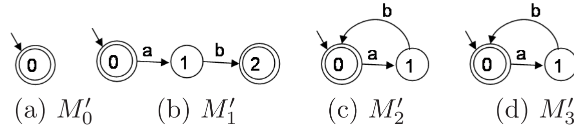
Definition 1. *$M_1 = \langle Q_1, q_{01}, \Sigma, \delta_1, F_1 \rangle$ is simulated by $M_2 = \langle Q_2, q_{02}, \Sigma, \delta_2, F_2 \rangle$ iff there exists a total function $f : Q_1 \setminus \{sink\} \rightarrow Q_2$ such that $\delta_1(q, \sigma) = sink$ or $f(\delta_1(q, \sigma)) = \delta_2(f(k), \sigma)$ for all $q \in Q_1 \setminus \{sink\}$ and $\sigma \in \Sigma$. Furthermore, $f(q_{01}) = q_{02}$ and for all $q \in F_1$, $f(q) \in F_2$.*

Definition 2. *$M = \langle Q, q0, \Sigma, \delta, F \rangle$ is state-disjoint iff there is no state $q \in Q$ such that there exist $\alpha \in \Sigma$ and $q', q'' \in Q$, $q' \neq q''$, and $\delta(q', \alpha) = q$ and $\delta(q'', \alpha) = q$.*

Theorem 1. *If (1) M_∞ exists, (2) M_∞ is a state-disjoint automaton, and (3) M_0 is simulated by M_∞, then (1) M'_∞ exists and (2) $M'_\infty = M_\infty$.*

Consider a simple example where we start from an empty string and simply concatenate a substring ab at each iteration. The exact sequence M_0, M_1, ..., M_i, ... will never converge to the least fixpoint, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 1 \leq k \leq i\} \cup \{\epsilon\}$. However, M_∞ exists and $L(M_\infty) = (ab)^*$. In addition, M_∞ is a state-disjoint automaton, and M_0 is simulated by M_∞. Based on Theorem 1, these conditions imply that once the computation of the approximate sequence reaches the fixpoint, the fixpoint is equal to M_∞ and the analysis is precise. Computation of the approximate sequence is shown in Fig 7. $M'_i = M'_{i-1} \nabla (M'_{i-1} \cup post(M'_{i-1}, R))$, where $post(M)$ returns an automaton that accepts $\{wab \mid w \in L(M)\}$. In this case, we reach the fixpoint at the 3rd iteration and $M'_\infty = M_\infty = M'_3$.

A more general case that we commonly encounter in real programs is that we start from a set of initial strings (accepted by M_{init}), and concatenate an arbitrary but fixed set of strings (accepted by M_{tail}) at each iteration. Based on Theorem 1 one can conclude that if the DFA M that accepts $L(M_{init})L(M_{tail})^*$ is state-disjoint, then our analysis via widening will reach the precise least fixpoint when it terminates.



(a) M'_0 (b) M'_1 (c) M'_2 (d) M'_3

Fig. 7. An approximate sequence

5 Experiments

We experimented with our string analysis tool on a number of test cases extracted from a set of real-world, open source applications: `MyEasyMarket-4.1` (a shopping cart program), `PBLguestbook-1.32` (a guestbook application), `Aphpkb-0.71` (a knowledge base management system), `BloggIT-1.0` (a blog engine), and `proManager-0.72` (a project management system). We believe that these programs are representative of how web applications use regular expression based replacement functions to modify their input (in particular, in a security context, to perform input sanitization), and, thus, are good test cases for our technique. These vulnerable functions were identified and sanitized by Balzarotti et al. in [1, 2].

Table 5 shows the results of applying our string analysis tool to these programs. The first column of Table 5 identifies the application, the function that was analyzed and the line number for the vulnerable operation. For each test case we analyzed the original version of the program (that contained the vulnerability) and a modified version which was modified with the intention of fixing the vulnerability. Our analysis is quite efficient and takes a couple of seconds. Since our string analysis tool is sound, it identifies the existing vulnerabilities correctly in each case. However, since our conservative approximations can lead to false positives, the fact that our tool identifies a possible vulnerability does not mean that it is guaranteed to be a vulnerability.

The impressive part of our results is that for all the modified program segments our approach is able to prove that the sanitization is correct. This indicates that the approximations we use work quite well in real-world applications.

We also experimented with Saner [1] to check these benchmarks. We discuss this tool in related work. The results are shown in table 5. Our tool performs slightly better than Saner in terms of time. It is interesting to note that there are some conflicts on the verification results. Saner performs bounded verification and approximates the value of out of bound computation as arbitrary strings. This rough approximation raises a false alarm while checking the sanitized version of `PBLguestbook-1.32`(1210). While checking `BloggIT-1.0`, Saner, in the default configuration, assumes that data from the database are sanitized; while we assume that these data may be tainted and model them the same as data from users. Saner raises an error for the sanitization routine in `PBLguestbook-1.32`(182) since it does not support the syntax of the replace operator used in that routine.

6 Related Work

Due to its importance in security, string analysis has been widely studied. Christensen, Møller and Schwartzbach [7] proposed a grammar-based string analysis

Application File(line)	Ver.	Res.	Final DFA state(bdd)	Peak DFA state(bdd)	Time user+sys(sec)	Mem (kb)	Saner n(type)	Saner Time(sec)
MyEasyMarket-4.1	o	y	17(133)	17(148)	0.010+0.002	444	1(xss)	1.173
trans.php(218)	m	n	17(132)	17(147)	0.009+0.001	451	0	1.139
PBLguestbook-1.32	o	y	42(329)	42(376)	0.019+0.001	490	1(sql)	1.264
pblguestbook.php(1210)	m	n	49(329)	42(376)	0.016+0.002	626	1(sql)	1.665
PBLguestbook-1.32	o	y	842(6749)	842(7589)	2.57+0.061	13310	1(reg)	4.618
pblguestbook.php(182)	m	n	774(6192)	740(6674)	1.221+0.007	8184	1(reg)	4.331
Aphpkb-0.71	o	y	27(219)	289(2637)	0.045+0.003	2436	1(xss)	1.220
saa.php(87)	m	n	18(157)	1324(15435)	0.177+0.009	11388	0	1.622
BloggIT 1.0	o	y	79(633)	79(710)	0.499+0.002	3569	0	0.558
admin.php(23,25,27)	o	y	126(999)	126(1123)				
	o	y	138(1095)	138(1231)				
	m	n	79(637)	93(1026)	0.391+0.006	5820	0	0.559
	m	n	115(919)	127(1140)				
	m	n	127(1015)	220(2000)				
proManager-0.72	o	y	387(3166)	2697(29907)	1.771+0.042	13900	1(xss)	6.980
message.php(91)	m	n	423(3470)	2697(29907)	2.091+0.051	19353	0	7.201

Table 1. Experimental results. Application: name of the application and the checked program point. Version: o-original, m-modified. Res.: y-the intersection of attack strings is not empty (vulnerable), n-the intersection of attack strings is empty (secure). Final DFA is the minimized DFA at the checked program point, and Peak DFA is the largest DFA observed during the fixpoint iteration. state: number of states. bdd: number of bdd nodes. n: number of warnings raised by Saner. type:(1) xss - cross site scripting vulnerablity, (2) sql - SQL injection vulnerability, (3) reg - regular expression error.

(implemented in a tool called JSA) to statically determine the values of string expressions in Java programs. They convert the flow graph into a context free grammar where each string variable corresponds to a nonterminal, and each string operation corresponds to a production rule. Then, they convert this grammar to a regular language by computing an over-approximation. Gould et al. [11] use this grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications [7]. Christodorescu et al. [8] present an implementation of the grammar-based string analysis technique for executable programs for the x86 architecture. Minamide [13] supports string-based replacement operations by escaping replace operations to finite-state transducers, and describes a string analysis similar to JSA to statically detect cross-site scripting vulnerabilities and to validate pages generated by web applications written in the PHP language. Wassermann et al. [18] proposed a static analysis to detect SQL injections following Minamide [13]. There are some other tools for string analysis [6, 9, 15, 19]. Shannon et al. [15] propose forward bounded symbolic execution to perform string analysis on Java programs. Similar to our approach, automata are used to trace path constraints and encode the values of string variables. They support trim and substring operations. Xie and Aiken [19] support string assignment and validation operations. Fu et al. [9] and Choi et al. [6] support string-based replacement (as opposed to language-based replacement). None of the tools mentioned above addresses language-based replacement operations. This defect causes the approximations computed by these tools to be too coarse for some input sanitization routines.

Language-based replacement has been discussed in computational linguistics [10, 12, 14, 17]. These algorithms are based on the composition of finite state transducers. By composing specific transducers, constraints like longest match

and first match can be precisely modeled. However, each composition may result in a quadratic size of non-deterministic automaton, and is more likely to blow-up compared to our construction. The transducer-based replacement function [14] has been implemented in Finite State Automata utilities (FSA) [16], where automata are stored and manipulated using an explicit representation. We use a symbolic DFA representation based on MBDDs. This symbolic encoding enables us to perform complex automata operations, such as closure, concatenation, replace, and widening, efficiently using the MBDDs.

Balzarotti et al. [1] combine both dynamic and static techniques to verify PHP programs. They support language-based replacement by incorporating FSA [16], but they only support bounded computation for loops and approximate variables updated in a loop as arbitrary strings once the computation does not converge within a fixed bound. We incorporate the widening operator in [3] to tackle this problem and obtain a tighter approximation that enables us to verify a larger set of programs.

Choi et al. [6] also investigates a widening method to analyze strings. The widening operator is defined on strings and the widening of a set of strings is achieved by applying the widening operator pairwise to each string pair. The widening operator we use is defined on automata, and was originally proposed for arithmetic constraints [3]. The intuition behind this widening operator is applicable to any symbolic fixpoint computation that uses automata. In [3] it is proved that for a restricted class of systems the widening operator computes the precise fixpoint and we extend this result to our analysis. Moreover, in our experiments, the over-approximation computed by this widening operator works well to prove the properties we were interested in.

Finally, the use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [5]). In this paper we focus on verification of string manipulation operations in PHP programs.

7 Conclusion

We proposed a symbolic approach for string verification on PHP programs. Our approach computes a conservative approximation of the set of values that a string variable can take at a given program point. We use a symbolic automata representation based on MBDDs and implement the string operations such as concatenation and replacement on this symbolic representation. Our experiments demonstrate that the proposed string analysis technique is capable of verifying the correctness of string sanitization operations in real-world PHP programs.

References

1. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proc. Symposium on Security and Privacy*, 2008.

2. D. Balzarotti, M. Cova, V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proc. 14th ACM conference on Computer and communications security*, pages 25–35, New York, NY, USA, 2007. ACM.
3. C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proc. 16th International Conference on Computer Aided Verification*, pages 321–333, 2004.
4. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *Proc. First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, LNCS 1260*. Springer Verlag, 1997.
5. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. 12th International Conference on Computer Aided Verification*, pages 403–418, 2000.
6. T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A practical string analyzer by the widening approach. In *APLAS*, pages 374–388, 2006.
7. A. S. Christensen, A. Møller, and M. 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.
8. M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *Proc. 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. ACM Press, September 2005.
9. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *Proc. 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*, pages 87–96, Washington, DC, USA, 2007. IEEE Computer Society.
10. D. Gerdemann and G. van Noord. Transducers from rewrite rules with backreferences. In *Proc. 9th Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, 1999.
11. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proc. 26th International Conference on Software Engineering*, pages 645–654, 2004.
12. L. Karttunen. The replace operator. In *Proc. 33rd annual meeting on Association for Computational Linguistics*, pages 16–23, 1995.
13. Y. Minamide. Static approximation of dynamically generated web pages. In *Proc. 14th International World Wide Web Conference*, pages 432–441, 2005.
14. M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Proc. 34th annual meeting on Association for Computational Linguistics*, pages 231–238. Association for Computational Linguistics, 1996.
15. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *Proc. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 13–22, Washington, DC, USA, 2007. IEEE Computer Society.
16. G. van Noord. FSA utilities toolbox. http://odur.let.rug.nl/ vannoord/Fsa/.
17. G. van Noord and D. Gerdemann. An extendible regular expression compiler for finite-state approaches in natural language processing. In *Proc. of the 4th International Workshop on Implementing Automata (WIA)*, pages 122–139. Springer-Verlag, July 1999.
18. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 32–41, 2007.
19. Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proc. 15th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2006. USENIX Association.