Noname manuscript No.
(will be inserted by the editor)

Automata-Based Symbolic String Analysis for Vulnerability Detection

Fang Yu · Muath Alkhalaf · Tevfik Bultan · Oscar H. Ibarra

Received: date / Accepted: date

Abstract Verifying string manipulating programs is a crucial problem in computer security. String operations are used extensively within web applications to manipulate user input, and their erroneous use is the most common cause of security vulnerabilities in web applications. We present an automata-based approach for symbolic analysis of string manipulating programs. We use deterministic finite automata (DFAs) to represent possible values of string variables. Using forward reachability analysis we compute an over-approximation of all possible values that string variables can take at each program point. Intersecting these with a given attack pattern yields the potential attack strings if the program is vulnerable. Based on the presented techniques, we have implemented Stranger, an automata-based string analysis tool for detecting string-related security vulnerabilities in PHP applications. We evaluated Stranger on several open-source Web applications including one with 350,000+ lines of code. Stranger is able to detect known/unknown vulnerabilities, and, after inserting proper sanitization routines, prove the absence of vulnerabilities with respect to given attack patterns.

Keywords String Analysis · Automated Verification · Web Application Security · Vulnerability Analysis

F. Yu
Department of Management Information Systems, National Chengchi University, Taipei, Taiwan
Tel.: +886-2-82377453
Fax: +886-2-29393754
E-mail: yuf@nccu.edu.tw

M. Alkhalaf, T. Bultan, O. H. Ibarra
Department of Computer Science, University of California, Santa Barbara
E-mail: {muath, bultan, ibarra}@cs.ucsb.edu

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 Web applications makes this an extremely serious problem.

According to the Open Web Application Security Project (OWASP)'s top ten list that identifies the most serious web application vulnerabilities, the top three vulnerabilities in 2007 [24] were: 1) Cross Site Scripting (XSS), 2) Injection Flaws (such as SQL Injection) and 3) Malicious File Execution (MFE). Even after it has been widely reported that web applications suffer from these vulnerabilities, the top two of the vulnerabilities were still listed in the top three of the OWASP's top ten list in 2010 [25] and 2013 [26]. That is to say, in the past decade, even with the increased awareness about their importance due to OWAPS reports, these vulnerabilities continued to be widely spread in modern web applications, causing great damage.

A *XSS vulnerability* results from the application inserting part of the user's input in the next HTML page that it renders. Once the attacker convinces a victim to click on a URL that contains malicious HTML/JavaScript code, the user's browser will then display HTML and execute JavaScript that can result in stealing of browser cookies and other sensitive data. An *SQL Injection vulnerability*, on the other hand, results from the application's use of user input in constructing database statements. The attacker can invoke the application with a malicious input that is part of an SQL command that the application executes. This permits the attacker to damage or get unauthorized access to data stored in a database. Finally, *MFE vulnerabilities* occur if developers directly use or concatenate potentially hostile input with file or stream functions, or improperly trust input files.

One important observation is, all these vulnerabilities are caused by improper string manipulation. Programs that propagate and use malicious user inputs without sanitization or with improper sanitization are vulnerable to these well-known attacks. In this paper, we focus on vulnerabilities related to string manipulation, and we propose a string analysis technique that identifies if a web application is vulnerable to the types of attacks we discussed above.

Attacks that exploit the vulnerabilities related to string manipulation can be characterized as *attack patterns*, i.e., regular expressions that specify potential attack strings for sensitive operations (called *sinks*). Given an application and an attack pattern, our vulnerability analysis identifies if there are any input values that a user can provide to the application that could lead to an attack string to be passed to a sensitive operation. We use automata-based string analysis techniques for vulnerability analysis. Our tool takes an attack pattern specified as a regular expression and a PHP program as input and identifies if there is any vulnerability with respect to the given attack pattern.

Our string analysis framework uses deterministic finite automaton (DFA) to represent values that string expressions can take. At each program point,

each string variable is associated with a DFA. To determine if a program has any vulnerabilities, we use a symbolic forward reachability analysis that computes an over-approximation of all possible values that string variables can take at each program point. Intersecting the results of the forward analysis with the attack pattern gives us the potential attack strings if the program is vulnerable.

The string analysis technique we present is a symbolic forward reachability computation that uses DFAs as a symbolic representation. Furthermore, we use the symbolic DFA representation provided by the MONA DFA library [5], in which transition relations of the DFAs 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 investigate algorithms of next state computation for string operations such as concatenation and language-based replacement on DFAs. Particularly, the language-based replacement operation is defined as 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. We detail the DFA constructions (without using ϵ transitions) for three common cases on M_3: (1) M_3 accepts an empty string, (2) M_3 accepts single characters, and (3) M_3 accepts words with more than one single character. For general cases of M_3, the construction can be done using ϵ transitions.

Our language-based replacement operation is essential for modeling PHP replacement commands, such as, `preg_replace()` and `str_replace()`, and many PHP sanitization routines, such as,`htmlspecialchars()`, that are commonly used to perform input validation. These functions provide mechanisms for scanning a string for matches to a given pattern, expressed as a regular expression, and replacing 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 `$_GET["username"]` returns the string entered by the user, and the `ereg_replace` call replaces all matches of the search pattern `<script *>` with the empty string, and the result is assigned to the variable `$username`. This statement can be modeled by our language-based replacement operation, REPLACE(M_1, M_2, M_3), where M_1 accepts arbitrary strings (modeling the user input), 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.

To the best of our knowledge we are the first to extend the MONA automata package to analyze these complex string operations with loops on real programs. Tateishi, Pistoia, and Tripp [32] apply MONA to analyze strings and their index of programs without loops. In addition to computation of the language-based replacement operation, another difficulty is implementing the

string operations required for our analysis without using the standard constructions based on the ϵ-transitions. The MBDD-based automata representation used by MONA does not allow ϵ-transitions. We model nondeterminism 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 [4] 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 our approach in a tool called Stranger (STRing AutomatoN GEneratoR) that analyzes PHP programs. Stranger takes a PHP program as input and automatically analyzes it and outputs the possible XSS, SQL Injection, or MFE vulnerabilities in the program. For each input that leads to a vulnerability, it also outputs an automaton (in the dot format) that characterizes all possible string values for this sink which may exploit the vulnerability. Stranger uses the front-end of Pixy, a vulnerability analysis tool for PHP that is based on taint analysis [19]. Stranger also uses the automata package of MONA [10] to store the automata constructed during string analysis symbolically. We used Stranger to analyze public web applications downloaded from [31] including one with 350,000+ lines of code. Our results demonstrate that our tool not only detects vulnerabilities in vulnerable Web applications, but also proves the correctness of sanitization routines in secure Web applications.

Contribution In this paper we extend the regular model checking techniques to verification of string manipulation operations in PHP programs. Our key contributions include:

1. an automatic automata-based approach for detecting or proving the absence of vulnerabilities in string manipulating programs,
2. a new algorithm to language-based replacement that can be used to model commonly used sanitization routines in Web applications,
3. a new forward reachability analysis with automata widening to accelerate/guarantee termination of fixpoint computation on strings
4. a new tool that implements our string analysis techniques, combined with a PHP front end and string manipulation library built on MONA.
5. an experimental study on XSS vulnerability checking against public and large-size Web applications.

Rest of the paper is organized as follows. In Section 2, we give an overview of our approach on some simple examples. In Section 3, we present our automata-

```
1 <?php
2    $www = $_GET["www"];
3    $l_otherinfo = "URL";
4    $www = preg_replace( "/[^A-Za-z0-9 .-@://]/", "", $www );
5    echo $l_otherinfo . ": " . $www ;
6 ?>
```

Fig. 1 A Small Example with Replacement

based string analysis. In Section 4, we describe our tool Stranger. In Section 5, we present the experiments we conducted using Stranger and discuss the results. In Section 6, we discuss the related work, and we conclude the paper in Section 7.

2 An Overview

Replacement In this section we give an overview of our analysis using some simple PHP scripts. The first script shown in Figure 1 is a simplified version of a vulnerability that exists in a web application called MyEasyMarket [2]. The script starts with assigning the user input provided in the `$_GET` array to the `$www` variable in line 2. Then, in line 3, it assigns a string constant to the `$l_otherinfo` variable. Next, in line 4, the user input is sanitized using the `preg_replace` command. This replace command gets three arguments: the match pattern, the replace string and the target string. It finds all the substrings of the target string that match the match pattern and replaces them with the replace string. In the replace command shown in line 4, the match pattern is the regular expression `[^A-Za-z0-9 .-@://]`, the replace string is the empty string (which corresponds to deleting all the substrings that match the match pattern), and the target string is the value of the variable `$www`. After the sanitization step, the PHP program outputs the concatenation of the variable `$l_otherinfo`, the string constant `": "`, and the variable `$www`.

The `echo` statement in line 5 is a sink statement since it can cause a Cross Site Scripting (XSS) vulnerability. For example, a malicious user may provide an input that contains the string constant `<script` and cause execution of a command leading to a XSS attack. The goal of the replace statement in line 4 is to remove any special characters from the input to prevent such attacks.

Using string replace operations to sanitize user input is common practice in web applications. However, this type of sanitization is error prone due to complex syntax and semantics of regular expressions. In fact, the replace operation in line 4 in Figure 1 contains an error that leads to a XSS vulnerability. The error is in the match pattern of the replace operation: `[^A-Za-z0-9 .-@://]`. The goal of the programmer was to eliminate all the characters that should not appear in a URL. The programmer implements this by deleting all the characters that do not match the characters in the regular expression `[A-Za-z0-9 .-@://]`, i.e., eliminate everything other than alpha-numeric characters, and the ASCII symbols `.`, `-`, `@`, `:`, and `/`. However, the regular expression is not correct. First,

there is a harmless error. The subexpression `//` can be replaced with `/` since repeating the symbol `/` twice is unnecessary. More serious error is the following: The expression `.-@` is the union of all the ASCII symbols that are between the symbol `.` and the symbol `@` in the ASCII ordering. The programmer intended to specify the union of the symbols `.`, `-`, and `@` but forgot that symbol `-` has a special meaning in regular expressions when it is enclosed with symbols `[` and `]`. The correct expression should have been `.\-@`. This error leads to a vulnerability because the symbol `<` (which can be used to start a script to launch a XSS attack) falls between the symbol `.` and the symbol `@` in the ASCII ordering. So, the sanitization operation fails to delete the `<` symbol from the input, leading to a XSS vulnerability.

Now, we explain how our approach automatically detects this vulnerability. First, the attack pattern for the XSS attacks can be specified as Σ^* `<script` Σ^*, i.e., any string that contains the substring `<script` matches the attack pattern. If, during the program execution, a string that matches the attack pattern reaches a sink statement, then we say that the program is vulnerable. For our small example, we simplify the attack pattern as Σ^* `<` Σ^*. Our analysis first generates the data dependency graph for the input PHP program and then conducts the forward reachability analysis on it. Figure 2 shows the dependency graph and the forward analysis result for the PHP script in Figure 1 (the program segment that corresponds to a node and the corresponding line number are shown inside the node). Nodes 1 and 2 correspond to the assignment statement in line 2, nodes 3 and 4, correspond to the assignment statement in line 3, nodes 5, 6, 7 and 8 correspond to the replace statement in line 4, and nodes 9, 10, 11, and 12 correspond to the concatenation operations and the echo statement in line 5. A circle node indicates a string operation. Specifically, we focus on two string operations: concatenation and replacement. A concatenation node, i.e., labelled with str_concat, has two input values: prefix and suffix taken from its predecessors (from left to right). A replacement node, e.g., labelled with preg_replace, str_replace, or ereg_replace, has three input values: match, replacement, and target strings taken from its predecessors (from left to tight). Under each node we show the result of the forward reachability analysis as a regular expression.

During forward analysis we characterize all the user input as Σ^*, i.e., the user can provide any string as input. Then, using our automata-based forward reachability analysis, we compute all the possible values that each string expression in the program can take. For example, during forward analysis, node 2, that corresponds to the value of the string variable `$www` after the execution of the assignment statement in line 2, is correctly identified as Σ^*. More interestingly, in node 8, the value of the string variable `$www` after the execution of the replace statement in line 4, is correctly identified as `[A-Za-z0-9 .-@:/]*` since any character that does not match the characters in the regular expression `[A-Za-z0-9 .-@://]` has been deleted.

Node 12 is the sink node. The result of the forward analysis identifies the value of the sink node as `URL:[A-Za-z0-9 .-@:/]*`. Next, we take the intersection of the result of the forward analysis with the attack pattern to identify if the

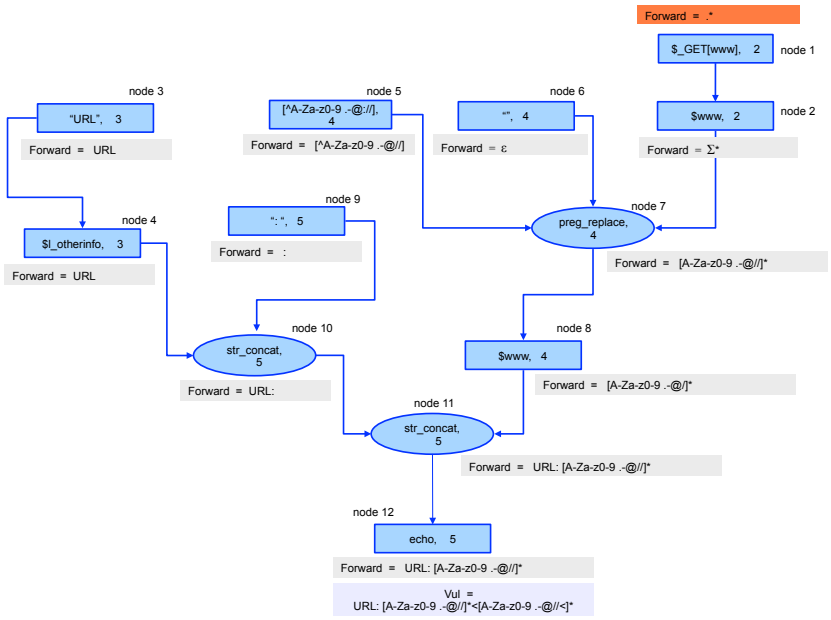


Fig. 2 Results of Forward Analysis

program contains a vulnerability. If the intersection is empty then the program is not vulnerable with respect to the given attack pattern. Since our analysis is sound, this means that there is no user input that can generate a string that matches the attack pattern at the sink node. However, in our example, the intersection of the attack pattern and the result of the forward analysis for the sink node is not empty and is characterized by the following regular expression: `URL:[A-Za-z0-9 .-;=-@:/]*<[A-Za-z0-9 .-@:/]*.`

Loop statement The second script shown in Figure 3 presents a simplified routine that iteratively appends a query result to a variable (later used to yield dynamic web pages). Within the loop (line 3 to line 6), while the result of the query has more rows, the value of the first element, i.e., `$reg[0]`, is sanitized in line 4 and later appended to `$www` in line 5. The value is sanitized in line 4 by applying the replacement statement (mentioned in the previous example) to replace the characters that are not in the regular expression `[A-Za-z0-9 .\-@:/]` with an empty string.

The loop yields a cycle in the dependency graph of this example. We add reachable states iteratively until a fix point has been reached. We also apply automata widening techniques (discussed in Section 3.6) to accelerate the computation. Upon termination, the result of the forward analysis identifies the value of the sink node (the echo statement in line 7) as `<tr [A-Za-z0-9 .\-@:/]*.` The intersection of the previous attack pattern and this result is empty, i.e.,

```
1 <?php
2    $www = "<tr ";
3    while($reg = mysql_fetch_row($query)){
4     $tmp = preg_replace( "/[^A-Za-z0-9 .\-@:/]/", "", $reg[0] );
5     $www = $www.$tmp;
6    }
7    echo $www;
8 ?>
```

Fig. 3 A Small Example with Loop

given any values from the database, the echo statement in line 7 will not take any input that matches the attack pattern. We conclude the segment is not vulnerable with respect to the attack pattern. For cases that there exist cycles in a dependency graph that may include complex string operations, e.g., replacement statements, our approach offers a rather precise analysis based on the fixpoint computation on automata, iteratively computing and adding the post images of (complex) string operations on reachable states.

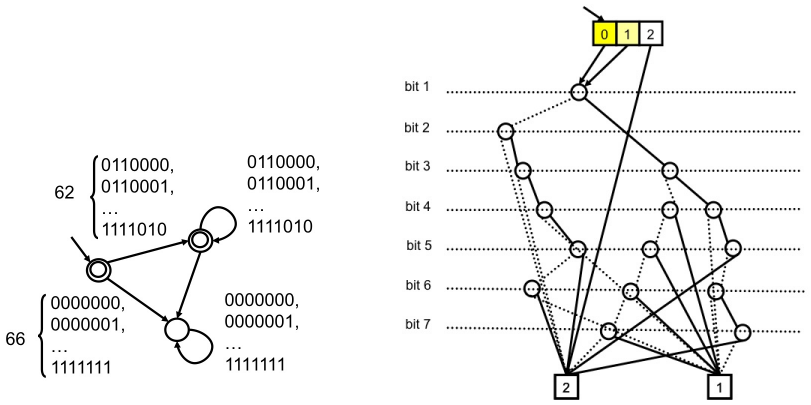
Finally, the work by Christensen et al. [12] can handle loops but not replacement statements, and, hence, cannot be directly applied to analyze applications with sanitization statements. Minamide et al. [23,27] propose using external transducers to model string replacement and matching statements. Their approach can precisely model string replacements similar to ours; however, for cases where replacement statements are within a loop (the example shown in Figure 3), their approach may lose precision due to a pre-determined number of external transducers used with grammars. In fact, most previous work [2, 32] overlooks replacement statements within a loop and adopts a coarse approximation, e.g., returning A^* where A is the set of possible characters, to estimate potential reachable strings. We provide a rather precise analysis with respect to handling of replace statements.

3 Automata-Based String Analysis

In this section, we first give basic definitions of automata and string operations. We then formally describe data dependency graph and our vulnerability analysis. We detail the automata construction for the post-image computation on concatenation and replacement operations, and at the end discuss how to incorporate automata widening techniques to reach a least fixpoint in the forward reachability analysis.

3.1 Preliminaries

Deterministic Finite Automata A DFA M is a tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$ where Q is a finite set of states, q_0 is the initial state, Σ is the alphabet, and $F \subseteq Q$ is the set of accepting states. $\delta : Q \times \Sigma \to Q$ is the transition relation. $\delta^* : Q \times \Sigma* \to Q$ extends δ to a word w. I.e., $q_n = \delta^*(q_0, w_0 w_1 \dots w_{n-1})$ if there



(a) explicit automata encoding (b) symbolic automata encoding

Fig. 4 An example of symbolic automata encoding with MBDDs

exists a sequence $q_0, ..., q_n$, such that $\forall 0 \leq i < n, \delta(q_i, w_i) = q_{i+1}$. A word w is accepted by M if $\delta^*(q_0, w) \in F$.

A state q of M is a *sink* state if $\forall \alpha \in \Sigma, \delta(q, \alpha) = q$ and $q \notin F$. 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.

The alphabet $\Sigma \subseteq B^k$ is encoded using k binary bits. For $\alpha \in \Sigma \subseteq B^k$, we also use $\alpha 0$ (or $\alpha 1$) $\in B^{k+1}$ to denote α appended with 0 (or 1).

Symbolic Automata Representation Our string analysis is an automata-based analysis. The set of string values is approximated as a regular language and represented as an automaton that accepts the language. It is hence essential to be able to encode automata and perform basic automata manipulations effectively. In addition, to analyze real-world web applications, it is also needed to deal with a large set of alphabet, e.g., ASCII or UNI code encodings. With this regard, symbolic encoding of automata poses an attractive solution to offer the ability to handle a large set of alphabet and a compact representation of automata. To achieve this goal, we leverage the MONA DFA library [10] for automata construction and manipulation. In MONA, a DFA is symbolically represented as a multi-terminal binary decision diagram (MBDD), where the transition relation of a DFA is encoded as a binary decision diagram (bdd) with multiple terminal nodes. Figure 4 shows an example of symbolic automata encoding using 7 bits (alphabet). The number of states is 3 and the number of bdd nodes is 15. In the symbolic encoding with MBDDs, a direct path from q to q' without any bdd node, e.g., state 2 to state 2 in Figure 4, indicates that $\delta(q, x) = q'$ for all $x \in \Sigma$. An ϵ-transition can not be specified with MBDDs.

In addition to a compact representation on transitions of DFAs, the MONA DFA library provides efficient implementations of standard automata operations. These operations include product, project and determinize, and min-

imize [22]. 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 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 i^{th} bit to each symbol of the alphabet. The process consists of removing the i^{th} track of the MBDD and determinizing the resulting MBDD via on-the-fly subset construction.

To deal with non-determinism, we extend the alphabet by adding extra bits, and then apply projection on the added bit(s) to map the resulting DFA to the original alphabet. Compared to Non-deterministic Finite Automata (NFA), using DFA suffers the inability of specifying ϵ-transitions and non-determinism, but provides efficient basic automata manipulation such as complement and inclusion checking; both are frequently used in our string analysis.

String Operations We focus on two common string operations: *concatenation* and *replacement* in this work. Both are widely used to manipulate strings in web applications. The concatenation of two strings w_1 and w_2 is defined as the string w_1w_2. The concatenation of two languages L_1 and L_2 is defined as the string set $\{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. We say a DFA M is the concatenation-DFA of M_1 and M_2 if and only if M accepts the concatenation of $L(M_1)$ and $L(M_2)$, i.e., $L(M) = \{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.

We define the replacement on languages as follows. Given M_1, M_2, and M_3 that accept the original strings, the match strings, and the replacement strings, respectively, the replacement language of the DFA tuple (M_1, M_2, M_3) is defined as the set $\{w \mid k > 0, w_1x_1w_2 \ldots w_kx_kw_{k+1} \in L(M_1), w = w_1c_1w_2...w_kc_kw_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), c_i \in L(M_3), \forall 1 \leq i \leq k+1, w_i \notin \{w'_1x'w'_2 \mid x' \in L(M_2), w'_1, w'_2 \in \Sigma^*\}\}$. We say a DFA M is the replaced-DFA of a DFA tuple (M_1, M_2, M_3) if and only if M accepts the replacement language of the DFA tuple (M_1, M_2, M_3). That is, M accepts the set of strings that can be yielded from a string s accepted by M_1 whose substrings that are accepted by M_2 are all replaced with any string r accepted by M_3.

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 example, consider $L(M_1) = \{baab\}$, $L(M_2) = a^+$ (M_2 accepts the language $\{a, aa, aaa, \ldots\}$) and $L(M_3) = \{c\}$. According to 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"]);`, which are precisely modeled by our language-based replacement operation. On the other hand, Sakuma et al. [27] proposed precise matching against different matching strategies using transducers. The presented replacement can be extended to different matching strategies with their approach.

3.2 Data Dependency Graph

A data dependency graph specifies the data flow in the program. We adopt the data flow analysis proposed in [19] to generate dependency graphs of PHP programs. It has been shown the data flow analysis by combining flow-sensitive analysis, inter procedural analysis, alias analysis and literal analysis, is able to effectively discover sensitive functions (sinks) that may take an input value that depends on the values of user input(s), as well as to construct the corresponding dependency graph to specify how the values of user inputs flow into a sink with string operations. As we have shown in the previous example (Figure 3), with proper string operations, these sinks may not raise vulnerabilities. In the following analysis, we assume that a vulnerability only comes from these sinks.

We formally define a dependency graph $G = \langle N, E, I \rangle$ as 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. A labeling function $l : N \rightarrow \{$ `input`, `constant`, `sink`, `variable`, `internal`, `concat`, `replace`, `un-modelled`$\}$ is defined to specify the type of a node. When $l(n)$ is `input`, n is an input node that identifies the data from untrusted parties, e.g., an input from web forms (node 1 in Figure 2). When $l(n)$ is `constant`, n is a constant node that is associated with a constant value (node 3, 6 and 9) or a constant regular expression (node 5). When $l(n)$ is `concat`, n is a concat node that has two predecessors labeled as the prefix node ($n.p$) and the suffix node ($n.s$) (node 10 and11). The node indicates a concatenation operation where $n.p$ keeps the prefix values and $n.s$. keeps the suffix values. (node 4 and 9 for node 10) When $l(n)$ is `replace`, n is a replace node that has three predecessors labeled as the target node ($n.t$), the match node ($n.m$), and the replacement node ($n.r$). The node indicates a replacement function in the program which can be modeled by our language-based replacement operation. (node 7) Specifically, $n.t$ keeps the values of the target strings, $n.m$ keeps the values of the match strings, and $n.r$ keeps the values of replacement strings, respectively. (node 2, 5 and 6 for node 7) When $l(n)$ is `sink`, n is a sink node that identifies a sensitive function in the program, which may take an input value that depends on the values of user inputs. (node 12) A sink node has no successors. When $l(n)$ is `internal`, n is an internal node. These nodes are used for transitions of the data flow, e.g., a temp variable in the program. (node 2 and 8) When $l(n)$ is `un-modeled`, n is a node that denotes an un-modeled built-in function in the original PHP program. We adopt a sound manner and assume the return values of these functions are arbitrary

strings. For $n \in N$, $Succ(n) = \{n' \mid (n, n') \in E\}$ is the set of successors of n. $Pred(n) = \{n' \mid (n', n) \in E\}$ is the set of predecessors of n. If $l(n)$ is `concat`, then $Pred(n) = \{n.p, n.s\}$. If $l(n)$ is `replace`, then $Pred(n) = \{n.t, n.m, n.r\}$. If $l(n)$ is `input` or `constant`, then $Pred(n) = \emptyset$. If $l(n)$ is `sink`, then $Succ(n) = \emptyset$. For a dependency graph G, we define $Root(G) = \{n \mid Pred(n) = \emptyset\}$ and $Leaf(G) = \{n \mid Succ(n) = \emptyset\}$.

Since a dependency graph specifies a sensitive function which may take an input value that depends on the value of user input(s), the following hypothesis holds:

Hypothesis 1 For a dependency graph $G = (N, E)$, there exists at least one node $n \in N$ such that $l(n)$ is `sink` and for each such n, there exists at least one node $n' \in N$ such that $l(n')$ is `input` and n is reachable from n' in G.

3.3 Vulnerability Analysis

Our vulnerability analysis is shown in Algorithm 1. The analysis takes two inputs: a dependency graph (denoted as G) and an attack pattern (denoted as M_{attack}). M_{attack} is a DFA that accepts a given set of attack strings (specified as a regular expression). We say G is vulnerable if there exists at least one sink node that can take value accepted by M_{attack}.

To associate each node with an automaton, we create an automata vector $POST$ with size $|N|$. Initially, all these automata accept nothing, i.e., their languages are empty. Vul is the set of nodes that are identified as vulnerable program points. Initially Vul is an empty set. The main computation is done by the process fwdAnalysis called in line 3. Upon its termination, $POST[n]$ is the DFA accepting all possible values that node n can take. In line 4, for each node n where $l(n)$ is `sink`, we generate a DFA M_{reach_attack} by intersecting the attack pattern M_attack and the automaton that accepts all possible values of node n (recorded in $POST[n]$). M_{reach_attack} accepts the set of reachable attack strings at node n that can be used to exploit the vulnerability. As checked in line 6, if M_{reach_attack} accepts a non-empty language, then n is identified as a vulnerable program point. We add n to Vul in line 7.

In line 10 to 15, if there exists at least one node in Vul, we report G is vulnerable, as well as report the vulnerability for each $n \in Vul$. Otherwise, we report G is not vulnerable with respect to the attack pattern.

3.4 Forward Analysis

The forward reachability analysis is based on a standard working queue algorithm as shown in Algorithm 2. We iteratively update the automata vector $POST$ until a least fixpoint is reached. In line 6, construct(n) returns a DFA that: (1) accepts arbitrary strings if $l(n)$ is `input` or `un-modeled`, (2) accepts the constant value if $l(n)$ is `constant`, or (3) accepts an empty string if $l(n)$ is

Algorithm 1 VulAnalysis(G, M_{attack})

```
1: Init(POST);
2: set Vul := ∅;
3: fwdAnalysis(G, POST);
4: for each n, where l(n) is sink do
5:    M_reach_attack: = POST[n] ∩ M_attack;
6:    if L(M_reach_attack) ≠ ∅ then
7:       Vul := Vul ∪ {n};
8:    end if
9: end for
10: if Vul ≠ ∅ then
11:    Report each vulnerability and its path;
12:    return "Vulnerable";
13: else
14:    return "Secure w.r.t. M_attack";
15: end if
```

`variable` (We assume (uninitialized) string variables are empty string by default.). In line 8 and line 10, we incorporate two automata-based string manipulating functions: $concat(M_1, M_2)$ and $replace(M_1, M_2, M_3)$ that return the concatenation-DFA of M_1 and M_2, and the replaced-DFA of (M_1, M_2, M_3), respectively. We discuss how to implement these functions in Sec. 3.5. In line 14, we incorporate the automata widening operator ∇(discussed in Sec. 3.6) to accelerate the fixpoint computation. Note that when there are cycles (cyclic dependency relations) in G, a least fixpoint (line 15) may never be reached given strings in infinite domain. Hence the computation does not terminate. We incorporate two kinds of widening operators to compute a least fixed point that over approximates all reachable states, but guarantees the termination.

Upon termination, $POST[n]$ stores the DFA whose language includes all possible values that n can take. This information is then passed to check whether the program is vulnerable.

3.5 Post-image Computation

3.5.1 $concat(M_1, M_2)$

We present the construction of the concatenation-DFA of M_1 and 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 $q \in F_1$ and $\exists q' \in Q_2, \delta_2(q_{20}, \alpha) = q'$
- $F^{'} = F_1 \cup F_2$, if $q_{20} \in F_2$; F_2, otherwise.

Algorithm 2 FwdAnalysis($G, POST$)

```
1: queue WQ;
2: WQ.enqueue(Root(G));
3: while WQ not empty do
4:    n := WQ.dequeue();
5:    if l(n) is input, constant, variable, or un-modeled then
6:       tmp := CONSTRUCT(n);
7:    else if l(n) is concat then
8:       tmp : = CONCAT(POST[n.p], POST[n.s]);
9:    else if l(n) is replace then
10:      tmp : = REPLACE(POST[n.t], POST[n.m], POST[n.r]);
11:   else
12:      tmp : = ⋃_{n'∈Pred(n)} POST[n'];
13:   end if
14:   tmp := (tmp ∪ POST[n])∇POST[n];
15:   if tmp ⊈ POST[n] then
16:      POST[n] := tmp;
17:      WQ.enqueue(Succ(n));
18:   end if
19: end while
```

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 $q \in F_1$ such that $\exists \alpha, q^{'}, q^{''}, \alpha \in \Sigma, q^{'} \in Q_1, q^{''} \in Q_2, \delta_1(q, \alpha) = q^{'}, \delta_2(q_{20}, \alpha) = q^{''}$.

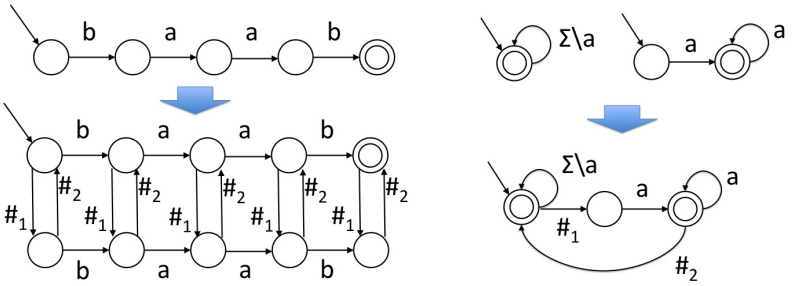
3.5.2 replace(M_1, M_2, M_3)

We present the construction of the replaced-DFA of (M_1, M_2, M_3). Without loss of generality, we assume that M_1, M_2, M_3 have the same alphabet Σ, and $\sharp_1, \sharp_2 \notin \Sigma$ are two auxiliary symbols. 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).

- $M_1^{'}$, s.t. $L(M_1^{'}) = \{w^{'} \mid k > 0, w = w_1 x_1 w_2 \dots w_k x_k w_{k+1} \in L(M_1), w^{'} = w_1 \sharp_1 x_1 \sharp_2 w_2 \dots w_k \sharp_1 x_k \sharp_2 w_{k+1}\}$.
- $M_2^{'}$, s.t. $L(M_2^{'}) = \{w^{'} \mid k > 0, w^{'} = w_1 \sharp_1 x_1 \sharp_2 w_2 \dots\ w_k \sharp_1 x_k \sharp_2 w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), \forall 1 \leq i \leq k+1, w_i \in L(M_h)\}$, 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, s.t. $L(M) = \{w \mid k > 0, w_1 \sharp_1 x_1 \sharp_2 w_2 \dots w_k \sharp_1 x_k \sharp_2 w_{k+1} \in L(M_1^{'}) \cap L(M_2^{'}), w = w_1 c_1 w_2 \dots w_k c_k w_{k+1}, \forall 1 \leq i \leq k, c_i \in L(M_3)\}$.

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 $M_2^{'}$,



(a) $M_1 : \{baab\}$ to $M_1^{'}$ (b) M_h, M_2 to $M_2^{'}$

Fig. 5 Construct $M_1^{'}$ 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.

The intuition is that we insert marks into automata, identify matching sub-strings by intersection of automata, and then construct the final automaton by replacing these matching sub-strings with replacement. We formally describe the construction of each step as 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 $L(M_3) = \{\epsilon\}$ to illustrate the construction step by step. 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 $q' \in Q_{1'}$.
- $\Sigma^{'} = \Sigma \cup \{\sharp_1, \sharp_2\}$
- $\delta_1^{'}(q_1, \alpha) = q_2$ and $\delta_1^{'}(q_{1'}, \alpha) = q_{2'}$, if $\delta_1(q_1, \alpha) = q_2$
- $\forall q_1 \in Q_1, \delta_1^{'}(q_1, \sharp_1) = q_{1'}$ and $\delta_1^{'}(q_{1'}, \sharp_2) = q_1$

An example for constructing $M_1^{'}$ from M_1, where $L(M_1) = \{baab\}$, is given in Figure 5(a). $|M_1^{'}|$ is bounded by $2|M_1|$.

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 Figure 5(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 taking the complement of (Concat(Concat(M_*, M_2), M_*)). We obtain the DFA in Figure 5(b) by applying this construction with minimization.

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

- $Q_2^{'} = Q_h \cup Q_2$
- $\Sigma^{'} = \Sigma \cup \{\sharp_1, \sharp_2\}$

- $\forall q, q' \in Q_h, \delta_2^{'}(q, \alpha) = q'$, if $\delta_h(q, \alpha) = q'$
- $\forall q, q' \in Q_2, \delta_2^{'}(q, \alpha) = q'$, if $\delta_2(q, \alpha) = q'$
- $\forall q \in Q_h, \delta_2^{'}(q, \sharp_1) = q_{20}$ if $q \in F_h$
- $\forall q \in Q_2, \delta_2^{'}(q, \sharp_2) = q_{h0}$ if $q \in F_2$

The corresponding $M_2^{'}$ for our example is shown in Figure 5(b). $|M_2^{'}|$ is bounded by $|M_h| + |M_2|$.

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 Figure 6. $|M^{'}|$ 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 w, $q' = \delta^*(q, \sharp_1 w \sharp_2)$. For instance, in Figure 6, 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 $L(M_2)$.

$M^{''}$ is constructed from $M^{'}$ by, for each q and $q' \in reach(q)$, insert paths between q and q' to recognize $L(M_3)$. Let $M_3 = \langle Q_3, q_{30}, \Sigma, \delta_3, F_3 \rangle$. This step can be done by adding an ϵ-transition from q to q_{30}, and for each $q'' \in F_3$, adding an ϵ-transition from q'' to q'. However, it is needed to prevent using ϵ-transitions with MBDDs, which can be done by nondeterminism, similar to previous techniques in the construction of ϵ-free NFA from regular expressions, e.g., [7]. To deal with nondeterminism with MBDDs, we add extra bits to the alphabet as we did in the construction of concatenation. Extra bits are added to the alphabet to make transitions deterministic and later be projected away to yield the DFA with the original alphabet.

For example, when there exist $q', q^{''} \in reach(q)$ and $q' \neq q^{''}$, the insertion will cause nondeterminism. 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 be under the fashion of DFA.

Though our replacement operation is defined in a general case in terms of M_3, we have observed that (actually, for all cases in our experiment) the replacement statements in PHP programs, such as `str_replace`, `preg_replace`, and `ereg_replace`, have L(M_3) in 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., $L(M_3) \subseteq \Sigma^+ \backslash \Sigma$, and
3. M_3 only accepts the empty string, i.e., $L(M_3) = \{\epsilon\}$.

We detail the direct DFA construction of $M^{''}$ for each case as below. One main part of the construction is introducing sufficient bits to encode nondeterminism. Let $P = \{q \mid q \in Q^{'}, \mid reach(q) \mid > 0\}$ be the set of initial states to insert M_3. 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 $q', q'' \in reach(q)$.

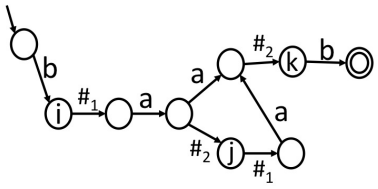
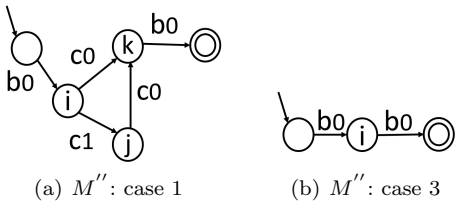


Fig. 6 Construct $M^{'}$ as the intersection of $M_1{'}$ and $M_2^{'}$



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

Fig. 7 Construct $M^{''}$ from $M^{'}$

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

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

In Figure 6, $P = \{i, j\}$, $reach(i) = \{j, k\}$ and $reach(j) = \{k\}$. Let $L(M_3) = \{c\}$. $M^{''}$ of the example is shown in Figure 7(a). Each symbol is appended with one extra bit, e.g., $\delta(i, c0) = j$ and $\delta(i, c1) = k$. $|M^{''}|$ is bounded by $|M^{'}|$.

Case 2: $L(M_3) \subseteq \Sigma^+ \setminus \Sigma$. 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

- $Q^{''} = Q^{'} \cup \bigcup_{p \in P} Q_p$
- $\Sigma^{''} \subseteq B^{k+m}$
- $\forall q \in Q^{'}, \delta^{''}(q, \alpha m_0) = q'$, if $\delta^{'}(q, \alpha) = q'$
- $\forall p \in P, \forall q \in Q_p, \delta^{''}(q, \alpha m_0) = q'$, 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 $q^{''} \in F_p$.

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

Case 3: $L(M_3) = \{\epsilon\}$. We consider this case as *deletion*. Let M_2^+ accept the kleene plus closure of $L(M_2)$. We have the following property.

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

The correctness comes from the fact that, by construction, if there exists $w \in L(M_2^+)$, 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 M_2^+ instead of M_2 yields the same result.

Note that the $\sharp$-reachable states of M' using M_2^+ 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, we use M_2^+ instead of M_2 as the match automaton.

$M^{''}$ 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 $\delta^{'}(q, \alpha) = q'$
- $\forall p \in P, \forall q \in reach(p), \delta^{''}(p, \alpha m_{q'}) = q'$, if $\delta^{'}(q, \alpha) = q'$.
- $F^{''} = \{p \mid p \in P, \exists q \in reach(p) \text{ and } q \in F^{'}\} \cup F^{'}$.

When $L(M_3) = \{\epsilon\}$, the result of $M^{''}$ is shown in Figure 7(b). Note that if $M_2 = \{a\}$, we would get the same result. $|M^{''}|$ is bounded by $|M^{'}|$.

Step 5: For the three cases, the final replaced-DFA M can then be constructed by iteratively projecting away the extra bits (over Σ) in $M^{''}$. The subset construction is only applied when needed.

The replaced-DFA of (M_1, M_2, M_3), where $L(M_1) = \{baab\}$, $L(M_2) = a^+$, and $L(M_3) = \{c\}$, is M that accepts $\{bcb, bccb\}$.

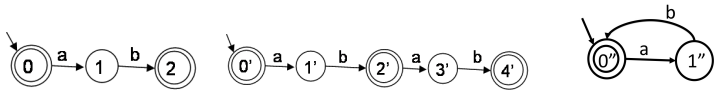
Exponential Blow-up Lemma 1 shows that a potential exponential blow-up of the number of states of the final DFA is inevitable in a replacement operation.

Lemma 1 *For every $n \geq 1$, there exists a DFA M with $O(n)$ states accepting a language $L \subseteq \{0,1\}^*\#\{0,1\}^*$ such that any DFA accepting the language L' obtained from L by replacing # with 1 requires $O(2^n)$ states.*

Proof Let $n \geq 1$. Let $L = \{x\#y \mid x, y \in \{0,1\}^*, |y| = n-1\}$. Clearly, L can be accepted by a DFA M with O(n) states. Now $L' = \{x1y \mid x, y \in \{0,1\}^*, |y| = n-1\}$. Below we show that any DFA accepting L' requires O(2^n) states. Assume a DFA A accepting L'. Let w be any binary string of length n, i.e., $|w| = n$. Let $s(w)$ denote the state that A enters after processing w. The proof is based on the fact that for any w and w' s.t. $|w| = |w'| = n$ and $w \neq w'$, $s(w) \neq s(w')$. Since there are 2^n distinct strings of length n, there are 2^n distinct $s(w)'s$. Hence, A has at least 2^n states.

3.6 Widening Automata

Finally, we describe the widening operator we use, which was originally proposed for arithmetic automata by Bartzis and Bultan [4].



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

Fig. 8 Widening automata

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

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

$$\text{or} \quad q, q' \neq sink \wedge \exists w \in \Sigma^*.\ \delta^*(q_0, w) = q \wedge \delta'^*(q'_0, w) = q', \tag{2}$$

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 $q_1, q_2 \in Q$ and $q_1 \neq q_2$ we say that $q_1 \equiv_\nabla q_2$ if and only if

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

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:

$$\begin{aligned}
Q'' &= C \\
q''_0 &= c \ \text{ s.t. } \ q_0 \in c \wedge q'_0 \in c \\
\delta''(c_i, \sigma) &= c_j \ \text{ s.t.} \\
&\quad (\forall q \in c_i \cap Q.\ \delta(q, \sigma) \in c_j \vee \delta(q, \sigma) = sink) \wedge \\
&\quad (\forall q' \in c_i \cap Q'.\ \delta'(q', \sigma) \in c_j \vee \delta'(q', \sigma) = sink) \\
F'' &= \{\ c \mid \exists q \in F \cup F'.\ q \in c\ \}
\end{aligned}$$

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', $L(M) \cup L(M') \subseteq L(M \nabla M')$ [4].

In Figure 8, 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 Algorithms 2, 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 single string variable represented with one automaton M. Let M_i represent the automaton computed at the i^{th} iteration and let I denote the set of initial values of the string variable. The fixpoint computation will compute a sequence M_0, M_1, ..., M_i, ..., where $L(M_0) = I$ and $L(M_i) = L(M_{i-1}) \cup L(post(M_{i-1}))$ where the post-image for different statements is computed as described in Section 3.5.

Definition 1 $L(M)$ is a fixpoint if $L(M) = L(M) \cup L(post(M))$. $L(M_\infty)$ is the least fixpoint if $L(M_\infty)$ is a fixpoint and for all other fixpoint $L(M)$, $L(M_\infty) \subseteq L(M)$.

We reach the least fixpoint $L(M_\infty) = L(M_i)$ at the i^{th} iteration when $L(M_i) = L(M_{i-1})$. Since we are dealing with an infinite state system, the fixpoint computation may not converge.

Given the widening operator, we actually compute a sequence M'_0, M'_1, ..., M'_i, ..., that over-approximates the fixpoint computation where M'_i is defined as: $L(M'_0) = L(M_0)$, and for $i > 0$, $L(M'_i) = L(M'_{i-1} \nabla M_i)$, where $L(M_i) = L(M'_{i-1}) \cup L(post(M'_{i-1}))$. Let M'_∞ denote the limit of this approximate sequence where there exists a j, $L(M'_\infty) = L(M'_j) = L(M'_{j-1})$. Then we have the following result from [4]:

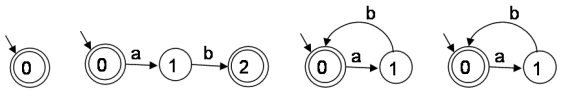
Definition 2 $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$ is simulated by $M_2 = \langle Q_2, q_{20}, \Sigma, \delta_2, F_2 \rangle$ if and only if there exists a total function $f : Q_1 \setminus \{sink\} \to Q_2$ such that $\delta_1(q, \sigma) = sink$ or $f(\delta_1(q, \sigma)) = \delta_2(f(q), \sigma)$ for all $q \in Q_1 \setminus \{sink\}$ and $\sigma \in \Sigma$. Furthermore, $f(q_{10}) = q_{20}$ and for all $q \in F_1$, $f(q) \in F_2$.

Definition 3 $M = \langle Q, q_0, \Sigma, \delta, F \rangle$ is state-disjoint if and only if for all $q \neq q' \in Q$, $L(q) \neq L(q')$, where $L(k) = \{w \mid \delta(k, w) \in F\}$, for $k \in Q$.

Theorem 1 *If (1) M_∞ exists, (2) M_∞ is a state-disjoint automaton, and (3) M_0 is simulated by M_∞, then if M'_∞ exists (i.e., if the approximate sequence converges) then $L(M'_\infty) = L(M_\infty)$.*

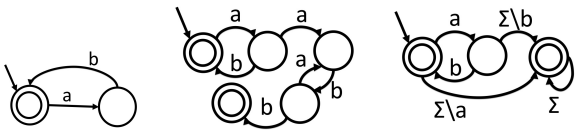
Recall the simple example where we start from an empty string and simply concatenate a substring ab at each iteration. The exact sequence M_0, M_1, ..., M_i, ... will never converge to the least fixpoint, where $L(M_0) = \{\epsilon\}$ and $L(M_i) = \{(ab)^k \mid 0 \leq k \leq i\}$. However, M_∞ exists 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 Figure 9. $L(M'_i) = L(M'_{i-1} \nabla M_i)$, where $L(M_i) = L(M'_{i-1}) \cup L(post(M'_{i-1}))$ and $post(M)$ returns an automaton that accepts $\{wab \mid w \in L(M)\}$. In this case, we reach the fixpoint at the 2^{nd} iteration and $M'_\infty = M_\infty = M'_2$.

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



(a) M'_0 (b) M_1 (c) M'_1 (d) M'_2

Fig. 9 The approximate sequence of a convergency example



(a) M'_1 (b)M'_2 (c) M''_2

Fig. 10 The approximate sequence of a non-regular example

$L(M_{init})L(M_{tail})^*$ is state-disjoint, then our analysis via widening will reach the precise least fixpoint when it terminates.

The automata widening operator defined in [4] has two variations and only the coarser version guarantees convergence. The coarser widening operator ∇_c is defined the same as ∇ except that we discard the condition $q, q' \neq sink$ in Eqt. (2). In our implementation we start with the more precise version (∇) and after a constant number of steps switch to the coarser version (∇_c) to guarantee convergence.

In general, the least fixpoint $L(M_\infty)$ may not exist, or even if it exists, the language may not be regular. For instance, if we change the previous example to that $post(M)$ returns an automaton that accepts $\{awb \mid w \in L(M)\}$ instead, we will have $L(M_i) = \{a^k b^k \mid 0 \leq k \leq i\}$. Though the least fixpoint $L(M_\infty) = \{a^n b^n \mid 0 \leq n\}$ exists, it is not a regular language. Let $L(M_0) = L(M'_0) = \{\epsilon\}$. For the above example, $M'_1 = M'_0 \nabla (M'_0 \cup post(M'_0))$ and $M'_2 = M'_1 \nabla (M'_1 \cup post(M'_1))$ are shown in Figure 10 (a) and (b), respectively. The sequence does not converge. On the other hand, if we apply the coarser widening operator at the second iteration, we get $M''_2 = M'_1 \nabla_c (M'_1 \cup post(M'_1))$ shown in Figure 10 (c), and reach the fixpoint at the next iteration. The result is an over approximation of the least fixpoint $L(M_\infty)$.

4 Stranger: A String Analysis Tool for PHP Programs

We have implemented the automata-based string analysis techniques described above in a tool for analyzing PHP programs. Stranger (STRing AutomatoN GEneRator) uses Pixy [19] as a front end and MONA [10] automata package for automata manipulation. Stranger takes a PHP program and an attack pattern (characterizing XSS, SQL Injection, or Malicious File Execution (MFE) vulnerabilities) as input and outputs the possible vulnerabilities with respect to the attack pattern. The architecture of Stranger is shown in Figure 11. The tool consists of the following parts.

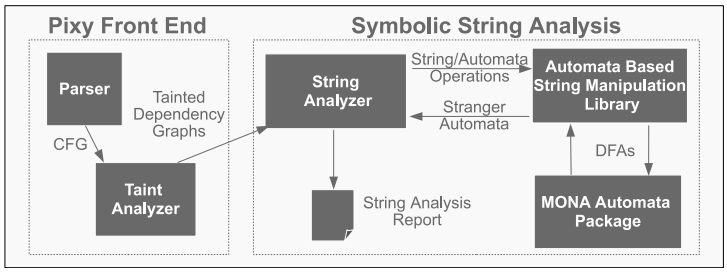


Fig. 11 The Architecture of STRANGER

4.1 PHP Parser and Taint Analyzer

The first step in our analysis is to parse the PHP program and construct the control flow graph (CFG). This is done by Pixy. PHP programs do not have a single entry point as in some other languages such as C and Java, so we process each script by itself along with all files included by that script. The CFG is passed to the taint analyzer in which alias and data dependency analyses are performed to generate data dependency graphs. Data dependency graphs specify how the inputs flow to sensitive functions (sinks) with respect to the *string operations*. The number of nodes in a dependency graph is linear with respect to the number of string operations. Loop structures cause cyclic dependency relations. Taint analysis determines if any user input (considered to be tainted-data) flows into a sink. If no tainted data flows into a sink, then taint analysis declares that sink to be secure; otherwise, the dependency graph for that sink is characterized as tainted and passed to the string analyzer for more analysis.

4.2 String Analyzer

The string analyzer implements our vulnerability analysis in Section 3.3 on the tainted dependency graphs found by taint analysis. The dependency graphs are further pre-processed to optimize the reachability analyses. First, a new acyclic dependency graph is built where all the nodes in a cycle (identifying cyclic dependency relations) are replaced by a single strongly connected component (SCC) node. The vulnerability analysis is conducted on the acyclic graph so that the nodes that are not in a cycle are processed only once. In the forward analysis, we propagate the post images to nodes in the topological order, initializing input nodes to DFAs accepting arbitrary strings. Upon termination, we intersect the language of the DFA of the sink node with the attack pattern. If the intersection is empty, we conclude that the sink is not vulnerable with respect to the attack pattern. Otherwise, we report each vulnerability with the path from the input node to the sink node. When we hit an SCC node, we switch to the working queue fixpoint computation (similar to Algorithm 2) on nodes that are part of the SCC represented by the SCC node.

During the fixpoint computation we apply automata widening in Section 3.6 (a fine widening operator) on reachable states to accelerate the convergence of the fixpoint computation. We added the ability to choose when to apply the widening operator. This option enables computation of the precise fixpoint in cases where the fixpoint computations converges after a certain number of iterations without widening. We also incorporate a coarse widening operator [4] that guarantees the convergence to avoid potential infinite iterations of the fixpoint computation.

4.3 String Manipulation Library

String manipulation library (SML) handles all core string and automata operations such as replacement, concatenation, prefix, suffix, intersection, union, and widening discussed in previous sections. During the vulnerability analysis, all string and automata manipulation operations that are needed to decorate a node in a dependency graph are sent to SML along with the string and/or automata parameters. SML, then, executes the operation and returns the result as an automaton. We defined a Java class called *StrangerAutomaton* as the type of the parameters and results. The class follows a well defined interface so that other automata packages can be plugged in and used with the string analyzer instead of SML. SML is also decoupled from the vulnerability analysis component so that it can be used with other string analysis tools. *StrangerAutomaton* encapsulates *libstranger.so* shared library that has the actual string manipulation code implemented in C (around 10K LOC) to get a faster computation and a tight control on memory. We used JNA (Java Native Access) to bridge the two languages. Another feature of Stranger is an option to produce a C trace of all string and automaton operations performed during a run to allow us to debug the code directly in gdb. Finally, the generated C trace can also be directly compiled with *libstranger.so* to yield the executable. This allows the extension of our string analysis to other languages (such as JavaScript and .NET) by generating the corresponding C traces (using *libstranger.so*) while parsing/analyzing the programs.

5 Experiments

5.1 Checking Vulnerable Benchmarks

We first experimented with Stranger on a number of benchmarks manually extracted from known vulnerable web applications: (1) MyEasyMarket-4.1 (a shopping cart program), (2) PBLguestbook-1.32 (a guestbook application), (3) BloggIT-1.0 (a blog engine), and (4) proManager-0.72 (a project management system). The Pixy front-end automatically generates the dependency graphs for these program segments and identifies that all of them may be vulnerable based on the taint analysis. In Table 1, we provide the data about the generated

	#nodes	#edges	#sinks	#inputs	#literals
1	21	20	1	1	51
2	41	44	1	2	99
3	32	31	1	1	142
4	119	117	3	3	450

Table 1 Sizes of Dependency Graphs

	Execution Time (seconds)		Memory Usage (Kb)
	total	forward analysis	
1	0.096	0.093	2700
2	0.132	0.124	5728
3	0.251	0.248	18890
4	0.557	0.462	116097

Table 2 Total Performance

	CONCAT		REPLACE	
	# operations	Time (seconds)	# operations	Time (seconds)
1	6	0.015	1	0.004
2	19	0.082	1	0.004
3	22	0.038	4	0.112
4	14	0.014	12	0.058

Table 3 String Function Performance

dependency graphs: #nodes and #edges indicate the number of nodes and edges in the dependency graph, #sinks indicates the number of sensitive sinks, #inputs indicates the number of `input` nodes. Since each program is identified to be vulnerable by taint analysis, there is at least one sensitive sink node and one input node in each dependency graph. We use #literals to denote the sum of the lengths of the constant strings that appear in the dependency graph. Note that these dependency graphs are built only for sensitive sinks where unrelated parts of the code have been removed manually. Hence, their sizes are much smaller than the original programs. In the next section we report our experiments on applying Stranger directly to web applications without any manual simplification.

In our experiments, we used an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We use 8 bits to encode each ASCII character. The performance of our string analysis is shown in Table 2. The total execution time includes the pre-processing time by the front-end (including parsing, dependency analysis and taint analysis) and the forward analysis. Table 3 shows the frequency and execution time of each of the string manipulating functions. Our computation does not suffer exponential blow-up. This shows some promise using symbolic DFA representation (provided by the MONA DFA library), in which transition relations of the DFAs are represented as Multi-terminal Binary Decision Diagrams (MBDDs).

	Reachable Attack (Sink)	
	#states	#bdd nodes
1	24	225
2	66	593
3	29	267
4	131	1221
	136	1234
	147	1333

Table 4 Automata that Accept Reachable Attack Strings

	Application	# of php files	total loc	# of sinks
1	Webchess 0.9.0	23	3375	421
2	EVE 1.0	8	906	114
3	Faqforge 1.3.2	10	534	375
4	Schoolmate	63	8287	898
5	Sendcard 3.4.1	72	11262	228
6	Necleus 3.64	71	37440	2764
7	Servoo	27	9288	6
8	Moodle 1.6	1353	374767	17310

Table 5 The Sizes of Analyzed Applications

Finally, Table 4 shows the data about the DFAs that Stranger generated with the number of states (#states) and the number of BDD nodes (#bdds). Reachable Attack is the DFA that accepts all possible attack strings at the sink node. For benchmark (4), there are three sinks with one input each, so we generate three Reachable Attack DFAs. As shown in Table 4, all the automata can be encoded with around one thousand BDD nodes, showing the efficiency of the MBDD encoding.

5.2 Detecting XSS vulnerabilities in Web Applications

In this section, we show the effectiveness of our approach on real-world Web applications. We have run Stranger to detect XSS vulnerabilities in several open source PHP Web applications [31] including Moodle, a popular course management system that consists of 1300+ files and 350,000+ lines-of-code in total. Other applications we analyzed are: Webchess 0.9.0 is a server for playing chess over the internet; EVE 1.0 is a tracker for players' activities for an online game; Faqforge 1.3.2 is a document management tool; Schoolmate is a class management system for elementary, middle and high schools; Sendcard is an advanced e-card program; Necleus is a content management system for developing weblogs. The size of these applications are showed in Table 5. Note that in these experiments we directly applied Stranger to analyze these applications without any manual modification.

As we mentioned earlier a sink is a sensitive function that can potentially be exploited if the application is vulnerable. For the XSS vulnerabilities the

sensitive functions include printf and echo. We also showed the number of sinks in each application in Table 5.

We use the attack pattern Σ^* <scriptΣ^* for detecting XSS vulnerabilities and report a vulnerability for a sink that can take an attack string (matching the attack pattern) as its input. We perform string analysis on top of tainted dependency graphs that the taint analysis generates for sinks that may use values depending on user inputs. The results of our analysis for these applications are summarized in Table 6.

We find vulnerabilities in all applications as shown in Table 6. For several applications such as Webchess, EVE, Faqforge, Schoolmate, and Necleus, the taint analysis and the string analysis report the same number of vulnerabilities. After checking the code of these applications manually, we find that they do not use sanitization functions. Since we assume that the values from a user input could be an arbitrary strings, a sink that uses values from user input(s) (reported as a vulnerability by the taint analysis) will be able to take an attack string as its input (reported as a vulnerability by the string analysis). On the other hand, when there are sanitization functions (e.g., replacement statements) in the applications, the string analysis is able to identify sinks that could not take any attack string as its input even if they are tainted sinks. For example, in Moodle, there are around 500 tainted sinks that are not vulnerable (taking an attack string as its input).

We summarize the performance in Table 6. The total time including pre-processing (parsing, dependency and taint analyses) time and string analysis time on all PHP files in these applications seem affordable and ranges from 8 seconds to 6800+ seconds.

Note that pre-processing times for some applications, such as Webchess, Faqforge, Sendcard, Necleus, and Moodle, are greater than the string analysis time because in these applications many of the sinks are determined to be secure by taint analysis, and, hence, they are not analyzed using string analysis. Despite having the least number of vulnerabilities, EVE 1.0 took a long time to be analyzed. The vulnerabilities in EVE 1.0 have many concatenation operations with long string constants and result in large automata (356 states and 3245 BDD nodes on average) in sink nodes to encode reachable attack strings. The last column in Table 6 shows the number of SCC nodes and the number of iterations to reach a fixpoint. Each SCC node presents a set of nodes that form a cyclic dependency relation in the dependency graph. It can be seen that this relation is not frequently appeared in real applications. During the fixpoint computation, we set the fine widening operator to be applied for the first 5 iterations and then followed by the coarse widening operator to be applied. There are 500 times that the widening operator has been invoked in total during the fixpoint computation on 184 SCC nodes. The result shows that the computation on the SCC nodes can reach the fixpoint with around 3 iterations on average under the settings.

To check whether the reported vulnerabilities (by the string analysis) are false alarms, we select some applications (Webchess, EVE, Faqforge, and Schoolmate), and try to remove the reported vulnerabilities by adding sanitization

	# of Vulnerabilities		Execution Time (sec)		Memory (KB)	# of SCC nodes/
	String	Taint	Total	Forward	Average	# of Iterations
1	27	27	38.26	1.88	5842	0/0
2	8	8	12.00	7.35	57424	0/0
3	20	20	7.65	0.22	6124	0/0
4	153	153	1707.05	1654.49	107333	5/10
5	12	13	103.84	26.08	4523	3/12
6	28	28	419.43	289.67	14455	1/2
7	1	6	26.99	0.03	1393	0/0
8	1976	2468	6829.67	1788.32	12609	175/476

Table 6 Vulnerability Analysis Results

statements. We manually inserted replacement statements to sanitize the values of each input that may exploit the identified vulnerabilities in these applications. For example, the following code sanitized the values assigned to `$name` in file Member.php in EVE 1.0.

```
$_POST["name"] = preg_replace('/</', "&lt;", $_POST["name"]);
$name = $_POST["name"];
```

Table 7 summarizes the result of our analysis on the properly sanitized applications. Note that since taint analysis overlooks the contents of string variables, the inserted statements do not affect the result of the taint analysis. On the other hand, string analysis shows the effectiveness of our sanitization routines. For Webchess, EVE and Faqforge, it reports that all sinks will not take any value that matches that attack pattern and concludes the security of the protected applications with respect to the the attack pattern. It also shows that all vulnerabilities identified by taint analysis in these applications are false alarms. Since there is no vulnerability reported by forward string analysis, Stranger concludes the correctness of the applications with the inserted sanitization routines w.r.t. the attack pattern. The total time including parsing, taint analysis and forward string analysis on all entries of PHP files in the applications improves to less than 40 seconds. The time for forward analysis also shows that Stranger can perform the presented replace operations efficiently.

On the other hand, for Schoolmate, string analysis reports 52 vulnerabilities even after 56 replacement statements have been inserted. We manually inspected each reported vulnerability and found that these could be false alarms due to database queries and unmodeled built-in functions of which we assume arbitrary return values (to keep our approach sound). The false alarms can be removed if we model all the functions that are used or sanitize their returned values.

6 Related Work

Due to its importance in security, string analysis has been widely studied. Christensen, Møller and Schwartzbach [12] pioneer the very first string analysis

	# of Vulnerabilities		# of Inserted	Execution Time (sec)	
	String Analysis	Taint Analysis	Statements	Total	Forward
1	0	27	33	39.15	2.96
2	0	8	23	9.35	5.43
3	0	20	20	7.79	0.28
4	52	153	56	713.27	660.91

Table 7 Vulnerability Analysis Results After Inserting Replacement Statements

(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. Kirkegaard et al. apply JSA to statically analyze the XML transformations in Java programs [21] by using DTD schemas as types and modeling the effect of XML transformation operations. Gould et al. [15] use this grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications [12]. Christodorescu et al. [13] present an implementation of the grammar-based string analysis technique for executable programs for the x86 architecture. There are some other tools for string analysis [39,11,30,14]. Shannon et al. [30] 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 [39] support string assignment and validation operations. Fu et al. [14] and Choi et al. [11] support string-based replacement (as opposed to language-based replacement). None of the tools mentioned above addresses language-based replacement operations which causes the approximations computed by these tools to be too coarse for analyzing some sanitization routines.

Minamide [23] proposes a grammar-based string analysis that supports language-based replacement operations by escaping replace operations to finite-state transducers. Instead of approximating the grammar to a regular language, Minamide performs string operations on context-free grammars and is able to validate HTML pages generated by web applications. Wassermann et al. [36,37] combine taint propagation with Minamide's string analysis [23] to detect SQL injections and XSS vulnerabilities in PHP Web applications.

Balzarotti et al. [2] combine both dynamic and static techniques to verify PHP programs. They support language-based replacement by incorporating FSA [33], 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 [4] to tackle this problem and obtain a tighter approximation that enables us to verify a larger set of programs.

Choi et al. [11] also investigate 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 [4]. The intuition behind this widening operator is applicable to any symbolic fixpoint computation that uses automata. In [4] 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 for proving the properties we were interested in.

Wassermann et al. [38] use string analysis in test input generation for Web applications. Their approach is based on concolic execution [29], where results of a concrete execution are used to collect constraints on program execution. These constraints are then used to generate new test cases. They use an automata based image computation similar to ours to propagate constraints. However, they do not discuss replacement operations which are crucial for string manipulation, and their approach targets test generation rather than generating a sound approximation of all possible inputs that can exploit a vulnerability. For example, their approach does not provide a sound approximation in the presence of loops.

HAMPI [20] is a bounded string constraint solver. It outputs a string that satisfies all the constraints, or reports that the constraints are unsatisfiable. Note that this type of bounded analysis cannot be used for sound string analysis whereas the string analysis techniques we present in this paper are sound. A string constraint solver called KUDZU [28] is built on top of the HAMPI. It uses the same approach of bounding the lengths of the execution paths (by bounding loops) and using a bounded string solver. In comparison, our approach handles unbounded paths (using widening) and handles unbounded strings (using automata). Bjoner et al. [6] present a path feasibility analysis based on solving bounded path conditions for string manipulating programs. Instead of solving string constraints directly, they solve their length constraints using an SMT solver. If the length constraints are unsatisfiable, it implies that the string constraints are unsatisfiable. If the length constraints are satisfiable, they use the satisfying assignment to bound the length of string variables and solve the string constraints over bounded string variables. Tateishi, Pistoia and Trip [32] adopt monadic second order logics to model relations of strings and their indices, achieving index sensitive string analysis. Compared to our analysis, their approach cannot deal with loops, and unlike the use of monadic second order logics provided by MONA [10], we simply use its DFA package and benefit from its MBDD encodings and efficient computations.

Hooimeijer and Weimer [17,18] present an automata-based decision procedure for solving equations over regular language variables. Since they use a single track automata encoding, the techniques in this paper can only provide an approximation for solving equations over string variables, as discussed in [44]. One potential solution is using multitrack automata to model relations among string variables [44]. Hooimeijer et al. propose a new symbolic automata representation [16] and use finite state transducers [35] to analyze behaviors of sanitization routines. Their tool Bek is able to identify whether a target

string is a valid output of a sanitization routine. Unlike this transducer-based approach, we develop an efficient automata construction for the string replace operation on top of MBDDs. The construction prevents the potential explosion in the size of the automata due to multiple conjunctions of transducers. Veanes and Bjorner [34] combine SMT with symbolic automata and show its effectiveness to encode and manipulate strings having large alphabets such as Unicode. The presented approach is also capable of encoding large alphabets by increasing BDD variables in MBDDs. In fact, Yu et al. [43] show that by adjusting BDD variables for various encodings one can adjust the precision and performance of string analysis.

The use of automata as a symbolic representation for verification has been investigated in other contexts (e.g., [9,8,3,4]). We extend the regular model checking techniques to verification of string manipulation operations. Preliminary results from this work were presented in [42,40]. We refine the analysis algorithms and detail the automata constructions in this paper along with new experimental results showing the effectiveness of the presented approach on large-scale web applications. While we focus on detecting vulnerabilities in this work, techniques for generating effective patches via automata-based string analysis [41] can also be incorporated with the presented approach.

7 Conclusion

We presented symbolic string analysis techniques for analyzing string manipulating programs with the goal of identifying string related vulnerabilities. Our approach is based on an automata-based symbolic forward reachability analysis. We implemented our approach in a tool called Stranger for automated analysis of PHP programs. Our tool successfully finds known/unknown vulnerabilities in existing web applications, and proves the absence of vulnerabilities with respect to the given attack patterns when the inputs are properly sanitized. In addition to vulnerability detection, it is essential to patch identified vulnerabilities effectively.

One extension of the presented work is *patch synthesis* [41]. Using automata, we are able to characterize malicious inputs and furthermore, generate effective patches to block (or modify) malicious inputs to prevent potential security exploits. Another extension is security checking on both client and server sides to prevent bleaches due to inconsistent behaviors between them [1].

Finally, we take advantage of symbolic encoding on DFAs to perform effective automata constructions and emptiness checking. However, it is known that using DFAs may suffer state explosion during determinization. One of our future work is to implement the presented string analysis using non-deterministic finite automata (NFA).

References

1. M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *ICSE*, pages 947–957, 2012.
2. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S&P*, pages 387–401, 2008.
3. C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.
4. C. Bartzis and T. Bultan. Widening arithmetic automata. In *CAV*, pages 321–333, 2004.
5. M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *WIA*, pages 6–25, 1997.
6. N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *TACAS*, pages 307–321, 2009.
7. R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Trans. on Comput.*, C-20(2):149 – 153, 1971.
8. A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *CAV*, pages 372–386, 2004.
9. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, pages 403–418, 2000.
10. BRICS. The MONA project. `http://www.brics.dk/mona/`.
11. 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.
12. A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.
13. M. Christodorescu, N. Kidd, and W.-H. Goh. String analysis for x86 binaries. In *PASTE*, pages 88–95, 2005.
14. X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC*, pages 87–96, 2007.
15. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654, 2004.
16. P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *SEC*, pages 1–1, 2011.
17. P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *PLDI*, pages 188–198, 2009.
18. P. Hooimeijer and W. Weimer. Strsolve: solving string constraints lazily. *Autom. Softw. Eng.*, 19(4):531–559, 2012.
19. N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *S&P*, pages 258–263, 2006.
20. A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA*, pages 105–116, 2009.
21. C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of xml transformations in java. *IEEE Trans. on Soft. Eng.*, 30(3), 2004.
22. N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Int. J. Found. Comput. Sci.*, 13(4):571–586, 2002.
23. Y. Minamide. Static approximation of dynamically generated web pages. In *WWW*, pages 432–441, 2005.
24. OWASP. Top 10 2007. `https://www.owasp.org/index.php/Top_10_2007`.
25. OWASP. Top 10 2010. `https://www.owasp.org/index.php/Top_10_2010-Main`.
26. OWASP. Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-T10`.
27. Y. Sakuma, Y. Minamide, and A. Voronkov. Translating regular expression matching into transducers. *J. Applied Logic*, 10(1):32–51, 2012.
28. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *S&P*, pages 513–528, 2010.
29. K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE*, pages 263–272, 2005.

30. D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION*, pages 13–22, 2007.
31. Sourceforge. Open sources. `http://sourceforge.net`.
32. T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. In *ISSTA*, pages 166–176, 2011.
33. G. van Noord. FSA utilities toolbox. `http://odur.let.rug.nl/~vannoord/Fsa/`.
34. M. Veanes and N. Bjørner. Symbolic automata: The toolkit. In *TACAS*, pages 472–477, 2012.
35. M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: algorithms and applications. In *POPL*, pages 137–150, 2012.
36. G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.
37. G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.
38. G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.
39. Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*, pages 13–13, 2006.
40. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, pages 154–157, 2010.
41. F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *ICSE*, pages 251–260, 2011.
42. F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN*, pages 306–324, 2008.
43. F. Yu, T. Bultan, and B. Hardekopf. String abstractions for string verification. In *SPIN*, pages 20–37, 2011.
44. F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. *Int. J. Found. Comput. Sci.*, 22(8):1909–1924, 2011.