Generating Vulnerability Signatures for String Manipulating Programs Using Automata-based Forward and Backward Symbolic Analyses

Fang Yu Muath Alkhalaf Tevfik Bultan
Computer Science Department
University of California at Santa Barbara
Santa Barbara, CA 93106–5110
Email: {yuf,muath,bultan}@cs.ucsb.edu

Abstract

Given a program and an attack pattern (specified as a regular expression), we automatically generate string-based vulnerability signatures, i.e., a characterization that includes all malicious inputs that can be used to generate attacks. We use an automata-based string analysis framework. 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 the attack pattern yields the potential attack strings if the program is vulnerable. Using backward analysis we compute an over-approximation of all possible inputs that can generate those attack strings. In addition to identifying existing vulnerabilities and their causes, these vulnerability signatures can be used to filter out malicious inputs. Our approach extends the prior work on automata-based string analysis by providing a backward symbolic analysis that includes a symbolic pre-image computation for deterministic finite automata on common string manipulating functions such as concatenation and replacement.

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. 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 are: 1) Cross Site Scripting (XSS), 2) Injection Flaws (such as SQL injection) and 3) Malicious File Execution. All these vulnerabilities are due to 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 propose a string analysis approach that 1) identifies if a web application is vulnerable to attacks, and 2) if it is vulnerable, generates a characterization of user inputs that might exploit that vulnerability. Such a characterization is called a vulnerability signature. We focus on vulnerabilities related to string manipulation such as the ones listed above. Vulnerabilities related to string manipulation can be characterized as attack patterns, i.e., regular expressions that specify vulnerable values for sensitive operations (called sinks).

Given an application, vulnerability analysis identifies if there are any input values that a user can provide to the application that could lead to a vulnerable value to be passed to a sensitive operation. Once a vulnerability is identified, the next important question is what set of input values can exploit the given vulnerability. A vulnerability signature is a characterization of all such input values. A vulnerability signature can be used to identify how to sanitize the user input to eliminate the identified vulnerability, or it can be used to dynamically monitor the user input and reject the values that can lead to an exploit.

We use automata-based string analysis techniques for vulnerability analysis and vulnerability signature generation. Our tool takes an attack pattern specified as a regular expression and a PHP program as input and 1) identifies if there is any vulnerability based on the given attack pattern, 2) generates a DFA characterizing the set of all user inputs that may exploit the vulnerability.

Our string analysis framework uses deterministic finite automaton (DFA) to represent values that string

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

```
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 ?>
```

Figure 1. A Small Example

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 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 backward analysis computes an over-approximation of all possible inputs that can generate those attack strings. The result is a DFA for each user input that corresponds to the vulnerability signature. We implemented our approach in a tool called Stranger (STRing AutomatoN GEneratoR) that analyzes PHP programs. Stranger uses the front-end of Pixy, a vulnerability analysis tool for PHP that is based on taint analysis [1]. Stranger also uses the automata package of MONA tool [2] to store the automata constructed during string analysis symbolically. We used Stranger to analyze four real-world web applications. Our results demonstrate that our tool can detect vulnerabilities in Web applications and identify the corresponding vulnerability signatures.

2. An Overview

In this section we will give an overview of our analyses using the simple PHP script shown in Figure 1. This script is a simplified version of code from a real web application that contains a vulnerability. 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 pattern and the target. The goal is to find all the substrings of the target that match the match pattern and replace them with the replace pattern. In the replace command shown in line 4, the match pattern is the regular expression `[^A-Za-z0-9 .-@://]`, the replace pattern is the empty string (which corresponds to deleting all the substrings that match the match pattern), and the target is 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 contain a Cross Site Scripting (XSS) vulnerability. For example, a malicious user may provide an input that contains the string constant `<script` and execute 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 will 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 $\Sigma^* < \Sigma^*$. Our analysis first generates the dependency graph for the input PHP program. Figure 2 shows the dependency graph 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. Under each node we show the result of the forward and backward symbolic analyses 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 symbolic 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, node 8, the value of the 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 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 .-@:/]*`. The backward analysis starts from this intersection and traverses the dependency graph backwards to find out what input values can lead to string values at the sink node that falls into this intersection. Note that during backward analysis we do not need to compute any value for the nodes that are not on a path between an input node and a sink node. This means that during backward analysis we do not compute values for the nodes 3, 4, 5, 6, 9 and 10. The final result of the backward analysis is the result for the input node 1, which is characterized with the regular expression: `[^<]*<`Σ^*, i.e., any input string that contains the symbol < can lead to a string value at a sink node that matches the attack pattern. Using this information, the programmer can eliminate the vulnerability either by fixing the erroneous replace statement in line 4 or by adding another replace statement that removes the < symbol from the input.

3. Automata-Based String Analyses

In this section, we first define the dependency graphs and then describe how to perform forward and backward symbolic string analyses on dependency graphs. We describe the pre-image computations on string manipulating functions at the end of this section, which are essential for the backward analysis.

3.1. Dependency Graph

A dependency graph specifies the data flow in the program. Formally speaking, a dependency graph $G = \langle N, E \rangle$ is a directed graph, where N is a finite set of nodes and $E \subseteq N \times N$ is a finite set of directed edges. An edge $(n_i, n_j) \in E$ identifies that the value of n_j depends on the value of n_i. Each node $n \in N$ can be (1) a `normal` node including `input`, `constant`, `variable`, or (2) an `operation` node including `concat` and `replace`. An `input` node identifies the data from untrusted parties, e.g., an input from web forms. A `constant` node is associated with a constant value. Both nodes have no predecessors. A `concat` node n has two predecessors labeled as the prefix node ($n.p$) and the suffix node ($n.s$), and stores the concatenation of any value of the prefix node and any value of the suffix node in n. A `replace` node has three predecessors labeled as the target node ($n.t$), the match node ($n.m$), and the replacement node ($n.r$). It performs the following operations for each value of $n.t$: (1) identifies all the matches, i.e., any value of $n.m$, that appear in $n.t$, (2) replaces all these matches in $n.t$ with any value of $n.r$, and (3) stores the replaced result in n. We define the following: 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 n is a `concat` node, $Pred(n) = \{n.p, n.s\}$. If n is a `replace` node, $Pred(n) = \{n.t, n.m, n.r\}$. For a dependency graph G, we also define $Root(G) = \{n \mid Pred(n) = \emptyset\}$ and $Leaf(G) = \{n \mid Succ(n) = \emptyset\}$.

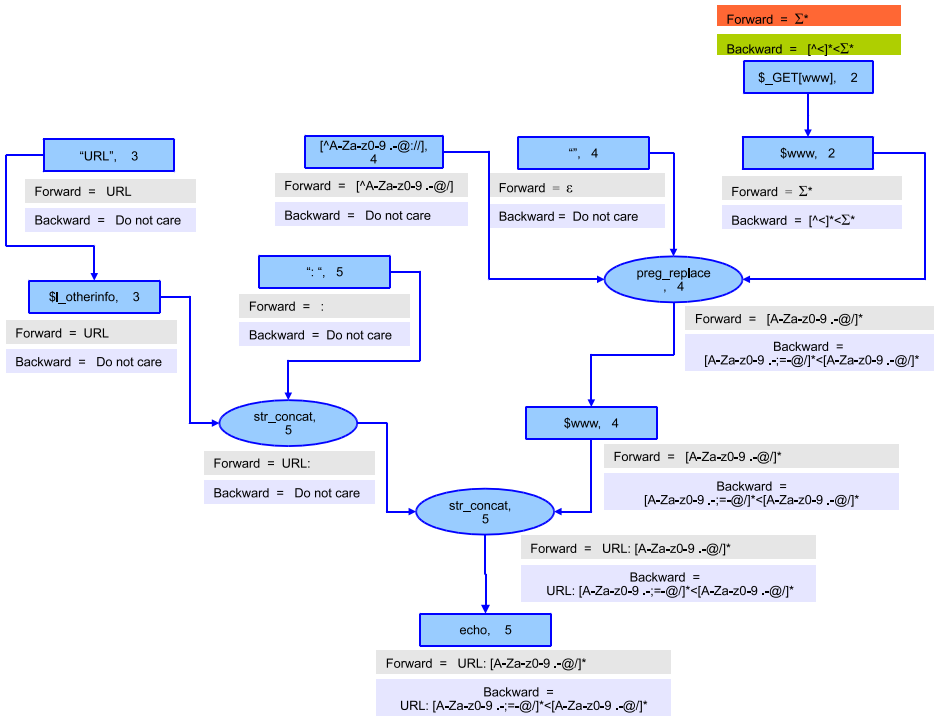

Figure 2. Results of Forward and Backward Analyses

3.2. Vulnerability Analysis

Our vulnerability analysis takes the following inputs: a dependency graph (denoted as G), a set of sink nodes (denoted as $Sink$), and an attack pattern (denoted as $Attk$). $Sink$ denotes the nodes that are associated with sensitive functions that might lead to vulnerabilities. $Attk$ is a regular expression represented as a DFA that accepts the set of attack strings.

Our vulnerability analysis is shown in Algorithm 1. The analysis consists of two phases. In the first phase, we perform a forward symbolic reachability analysis from root nodes to compute all possible values that each node can take (by calling forward analysis at line 3). We use this information to collect vulnerable program points, as well as the reachable attack strings of those vulnerable program points (at line 4-10). If the program is vulnerable, i.e., there exists some vulnerable program points, we proceed to the second phase (by calling backward analysis at line 12). In the second phase, we perform a backward symbolic reachability analysis from the vulnerable program points to compute all possible values of their predecessors that will result in attack strings at these vulnerable program points.

Our analysis is an automata-based analysis. The set of string values is approximated as a regular language and represented symbolically as a DFA that accepts the language. To associate each node with its automata, we create two automata vectors *POST* and *PRE*. The size of both is bounded by $|N|$. *POST*$[n]$ is the DFA accepting all possible values that node n can take. *PRE*$[n]$ is the DFA accepting all possible values that node n can take to exploit the vulnerability. Initially, all these automata accept nothing, i.e., their language is empty. $Vul \subseteq Sink$ is the set of vulnerable program points and initially is set to an empty set.

At line 3, we first compute *POST* by calling the forward analysis. At line 4, for each node $n \in Sink$, we generate a DFA tmp by intersecting the attack pattern and the possible values of n. If $L(tmp)$ is not empty, we identify that n is a vulnerable program point and add it to Vul at line 7. In fact, tmp accepts the set of reachable attack strings at node n that can be used to exploit the vulnerability. Hence, we assign tmp to *PRE*$[n]$ at line 8. If Vul is not empty, we compute *PRE* by calling our backward analysis at line 12. (We will discuss the backward analysis later.) Note that for $n \in Vul$, *PRE*$[n]$ has been assigned. We report vulnerability signatures for each `input` node based on *PRE* at line 13-15. If Vul is an empty set, we report that the program is secure with respect to the attack pattern.

Algorithm 1 VULANALYSIS($G, Sink, Attk$)

```
1:  Init(POST, PRE);
2:  set Vul := {};
3:  FWDANALYSIS(G, POST);
4:  for each n ∈ Sink do
5:      tmp: = POST[n] ∩ Attk;
6:      if L(tmp) ≠ ∅ then
7:          Vul := Vul ∪ {n};
8:          PRE[n] := tmp;
9:      end if
10: end for
11: if Vul ≠ ∅ then
12:     BWDANALYSIS(G, POST, PRE, Vul);
13:     for each input n do
14:         Report the vulnerability signature PRE[n];
15:     end for
16:     return "Vulnerable";
17: else
18:     return "Secure";
19: end if
```

Algorithm 2 FWDANALYSIS(G, *POST*)

```
1:  queue WQ := NULL;
2:  WQ.enqueue(Root(G));
3:  while WQ ≠ NULL do
4:      n := WQ.dequeue();
5:      if n ∈ Root(G) then
6:          tmp := CONSTRUCT(n);
7:      else if n is concat then
8:          tmp : = CONCAT(POST[n.p], POST[n.s]);
9:      else if 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
```

3.3. Forward Analysis

The forward symbolic reachability analysis is based on a standard work queue algorithm (Algorithm 2). We iteratively update the automata vector *POST* until a fixpoint is reached. At line 6, CONSTRUCT(n) returns a DFA that: (1) accepts arbitrary strings if n is an `input` node, (2) accepts an empty string if n is a `variable` node, or (3) accepts the constant value if n is a `constant` node. At line 8 and line 10, we incorporate two automata-based string manipulating functions [3]:

- CONCAT(DFA M_1, DFA M_2) returns a DFA M that accepts $\{w_1w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.
- REPLACE(DFA M_1, DFA M_2, DFA M_3) returns a DFA M that accepts $\{w_1c_1w_2c_2 \dots w_kc_kw_{k+1} \mid k > 0, w_1x_1w_2x_2 \dots w_kx_kw_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.

At line 14, we incorporate the automata widening operator ∇ to accelerate the fixpoint computation [4]. Upon termination, *POST*$[n]$ records the DFA whose language includes all possible values that n can take. This information is then passed to our backward analysis.

3.4. Backward Analysis

Backward analysis uses the results of the forward analysis. Particularly, we are interested in computing all possible values of each node n that can exploit the identified vulnerability. The first challenge of the backward analysis comes from the pre-image computation on string manipulating functions. To tackle this challenge, we develop the following automata-based functions.

- PRECONCATPREFIX(DFA M, DFA M_2) returns a DFA M_1 so that $M = \text{CONCAT}(M_1, M_2)$.
- PRECONCATSUFFIX(DFA M, DFA M_1) returns a DFA M_2 so that $M = \text{CONCAT}(M_1, M_2)$.
- PREREPLACE(DFA M, M_2, M_3) returns a DFA M_1 so that $M = \text{REPLACE}(M_1, M_2, M_3)$.

We discuss how to implement these functions at the end of this section. The backward analysis is shown in Algorithm 3. For $n \in Vul$, *PRE*$[n]$ is set to the intersection of *POST*$[n]$ and $Attk$ before the backward analysis starts. The predecessors of $n \in Vul$ are the starting points of the backward analysis. Similar to the forward analysis, the computation is based on a standard work queue algorithm. We first put the predecessors of $n \in Vul$ into the work queue as shown at line 2-4. We iteratively update the *PRE* array (by adding pre-images) until we reach a fixpoint. If the successor of n is an operation node, the pre-image (tmp) of n is computed by calling the defined automata-based functions. (line 11, 13, 17). Otherwise, the pre-image of n is directly derived from the successor of n (line 20). Note that *POST*$[n]$ records all possible values that n can take. We use this information during the pre-image computation by restricting the arguments of operations such as replace. We union the pre-images of n as tmp' at line 22. Since we are interested only in reachable values of n, i.e., *PRE*$[n] \subseteq$ *POST*$[n]$ by definition, we intersect tmp' with *POST*$[n]$ at line 24. Similar to the forward analysis, we widen the result at line 25 to accelerate the fixpoint computation. At line

26, we intersect tmp' with *POST*$[n]$ again to remove unreachable values (that might have been introduced due to widening) at node n. If tmp' accepts more values than *PRE*$[n]$, we update *PRE*$[n]$ at line 28 and add the predecessors of n to the working queue at line 29. Upon termination, *PRE*$[n]$ records the DFA that accepts all possible values of n that may exploit the identified vulnerability.

Algorithm 3 BWDANALYSIS($G, POST, PRE, Vul$)

```
1:  queue WQ = NULL;
2:  for each n ∈ Vul do
3:      WQ.enqueue(Pred(n));
4:  end for
5:  while WQ ≠ NULL do
6:      n := WQ.dequeue();
7:      tmp' := NULL;
8:      for each n' ∈ Succ(n) do
9:          if n' is concat then
10:             if n is n'.l then
11:                 tmp := preConcatPrefix(PRE[n'],
                    POST[n'.r]);
12:             else
13:                 tmp := preConcatSuffix(PRE[n'],
                    POST[n'.l]);
14:             end if
15:         else if n' is replace then
16:             if n is n'.t then
17:                 tmp := preReplace(PRE[n'], POST[n'.m],
                    POST[n'.r]);
18:             end if
19:         else
20:             tmp := PRE[n'];
21:         end if
22:         tmp' := tmp' ∪ tmp;
23:     end for
24:     tmp' := tmp' ∩ POST[n];
25:     tmp' := (tmp' ∪ PRE[n])∇PRE[n];
26:     tmp' := tmp' ∩ POST[n];
27:     if tmp' ⊈ PRE[n] then
28:         PRE[n] := tmp';
29:         WQ.enqueue(Pred(n));
30:     end if
31: end while
```

3.5. Pre-image Computation

In this section, we discuss how to compute the pre-images of string manipulating functions, as well as the implementation of the following functions: preConcatPrefix(M_x, M_z), preConcatSuffix(M_x, M_y), and preReplace(M_x, M_m, M_r).

3.5.1. Concatenation. To compute the pre-image of concatenation nodes, we introduce concatenation transducers to specify the relation among its output and two input nodes. A concatenation transducer is a DFA over the alphabet that consists of 3 tracks. The 3-track alphabet is defined as $\Sigma^3 = \Sigma \times (\Sigma \cup \{\lambda\}) \times (\Sigma \cup \{\lambda\})$, where $\lambda \notin \Sigma$ is a special symbol for padding. We use $w[i]$ $(1 \leq i \leq 3)$ to denote the i^{th} track of $w \in \Sigma^3$. All tracks are aligned. $w[1] \in \Sigma^*$, $w[2] \in \Sigma^*\lambda^*$ is left justified, and $w[3] \in \lambda^*\Sigma^*$ is right justified. We use $w'[1], w'[2] \in \Sigma^*$ to denote the λ-free prefix of $w[1]$ and the λ-free suffix of $w[2]$. We say w is accepted by a concatenation transducer M if $w[1] = w'[2].w'[3]$. Note that a concatenation transducer binds the values of different tracks character by character and hence is able to identify the prefix and suffix relations precisely.

Below we show two examples of concatenation transducers. Let α indicate any character in Σ. In Figure 3, the third track of M can be used to identify all suffixes of X that follow any string in $(ab)^+$. In Figure 4, the second track of M can be used to identify all prefixes of X that are followed by any string in $(ab)^+$.

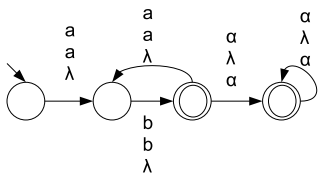

Figure 3. A transducer M for $X = (ab)^+.Z$

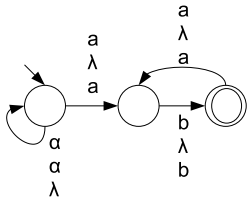

Figure 4. A transducer M for $X = Y.(ab)^+$

In the following, we describe how to construct these transducers in general, how to remove λ, and how to compute the pre-images of a concatenation node using concatenation transducers.

Prefix: We first consider how to compute the pre-image of the prefix node, i.e., Y in $X := YZ$, given regular sets characterizing possible values of the output node X and the suffix node Z. Let $M_x = \langle Q_x, \Sigma, \delta_x, q_x 0, F_x \rangle$, $M_z = \langle Q_z, \Sigma, \delta_z, q_{z0}, F_z \rangle$ accept values of X and Z respectively. preConcatPrefix(M_x, M_z) returns M_y.

- Extend M_x to a 3-track DFA M', so that M' accepts $\{w \mid w[1] \in L(M_x)\}$.
- Construct the concatenation transducer M that accepts $\{w \mid w[1] = w'[2].w'[3], w'[3] \in L(M_z)\}$. $M = \langle Q, \Sigma^3, \delta, q_0, F \rangle$, where:
  - $Q = \{q_0\} \cup Q_z$,

- $\forall a \in \Sigma, \delta(q_0, (a, a, \lambda)) = q_0$,
- $\forall a \in \Sigma, \delta(q_0, (a, \lambda, a)) = q'$ if $\delta_z(q_{z0}, a) = q'$.
- $\forall q, q' \in Q_z, \forall a \in \Sigma, \delta(q, (a, \lambda, a)) = q'$ if $\delta_z(q, a) = q'$.
- $F = \{q_0\} \cup F_z$ if $q_{z0} \in F_z$. $F = F_z$, otherwise.

- Intersect M' with M. The result accepts $\{w \mid w[1] = w'[2].w'[3], w[1] \in L(M_x), w'[3] \in L(M_z)\}$. We then project away the first and the third tracks. Let the result be $M'_y = \langle Q_y, \Sigma \cup \{\lambda\}, \delta, q'_{y0}, F'_y\rangle$.
- Remove λ tails if any. We construct $M_y = \langle Q_y, \Sigma, \delta_y, q_{y0}, F_y\rangle$ as below.
  - $\forall q, q' \in Q_y, \forall a \in \Sigma, \delta_y(q, a) = q'$ if $\delta'_y(q, a) = q'$.
  - $F_y = F'_y \cup F_\lambda$, where $F_\lambda = \{q \mid \exists q' \neq sink, \delta'_y(q, \lambda) = q'\}$.

Suffix: We next consider how to compute the pre-image of the suffix node, i.e., Z in $X := YZ$, given regular sets characterizing possible values of X and the prefix node Y. Again, let $M_x = \langle Q_x, \Sigma, \delta_x, q_x 0, F_x\rangle$, $M_y = \langle Q_y, \Sigma, \delta_y, q_{y0}, F_y\rangle$ accept values of X and Y respectively. preConcatSuffix(M_x, M_y) returns M_z.

- Extend M_x to a 3-track DFA M', so that M' accepts $\{w \mid w[1] \in L(M_x)\}$.
- Construct the concatenation transducer M that accepts $\{w \mid w[1] = w'[2].w'[3], w'[2] \in L(M_y)\}$. $M = \langle Q, \Sigma^3, \delta, q_{y0}, F\rangle$, where:
  - $Q = Q_y \cup \{q_f\}$
  - $\forall q, q' \in Q_y, \forall a \in \Sigma, \delta(q, (a, a, \lambda)) = q'$ if $\delta_y(q, a) = q'$.
  - $\forall q \in F_y, \forall a \in \Sigma, \delta(q, (a, \lambda, a)) = q_f$.
  - $\forall a \in \Sigma, \delta(q_f, (a, \lambda, a)) = q_f$.
  - $F = \{q_f\} \cup F_y$.
- Intersect M' with M. The result accepts $\{w \mid w[1] = w'[2].w'[3], w[1] \in L(M_x), w'[2] \in L(M_y)\}$. We then project away the first and the second tracks. Let the result be $M'_z = \langle Q'_z, \Sigma \cup \{\lambda\}, \delta'_z, q'_{z0}, F'_z\rangle$.
- Remove λ heads if any. This final step can be done by constructing $M_z = \langle Q_z, \Sigma, \delta_z, q_{z0}, F_z\rangle$ as below.
  - $Q_z = q_0 \cup Q'_z$.
  - $\forall q \in Q'_z, \forall a \in \Sigma, \delta_z(q, a) = q'$ if there exists $q' \in Q'_z, \delta'_z(q, a) = q'$.
  - $\forall q \in Q'_z, \forall a \in \Sigma, \delta_z(q_0, a) = q'$ if there exists $q', q'' \in Q'_z$, $\delta'_z(q'', \lambda) = q$ and $\delta'_z(q, a) = q'$.
  - $F_z = \{q_0\} \cup F'_z$, if $\exists q \in F'_z$ and there exists $q', q'' \in Q'_z$, so that $\delta'_z(q'', \lambda) = q$ and $\delta'_z(q, a) = q'$. $F_z = F'_z$, otherwise.

3.5.2. Replacement. Recall that a `replace` node has three input nodes: target, match, and replacement. We only consider the pre-image of the target node given regular sets characterizing possible values of the output node, the match node, and the replacement node. Let M_x = replace(M_t, M_m, M_r). We are interested in computing M_t, given M_x, M_m, and M_r. An intuitive solution of preReplace(M_x, M_m, M_r) is replace(M_x, M_r, M_m). However, since not all matches of M_r that appear in M_x are due to the replacement operation, this may break the soundness of our approach. Consider a simple example. M_t, M_m and M_r accept $\{aab\}$, $\{b\}$, and $\{a\}$, respectively. M_x = replace(M_t, M_m, M_r) accepts $\{aaa\}$. M'_t = replace(M_x, M_r, M_m) accepts $\{bbb\}$. Since $\{bbb\}$ does not include $\{aab\}$, this intuitive approach is not sound. Instead, we conservatively model preReplace(M_x, M_m, M_r) as replace(M_x, M_r, $M_m \cup M_r$). The result is an over approximation of the pre-image of the target node. For the simple example, M'_t = replace(M_x, M_r, $M_m \cup M_r$) accepts $(a|b)(a|b)(a|b)$, which includes all $L(M_t)$ such that replace(M_t, M_m, M_r) accepts $\{aaa\}$.

Deletion replace(M_t, M_m, M_r) performs deletion if M_r accepts the empty string. I.e., it will delete all the matches in $L(M_t)$. In this case, to compute the pre-image of the target, we would not be able to find a match of M_r (an empty string in this case) to replace with M_m. In this case, replace(M_x, M_r, $M_m \cup M_r$) will return M_x. To deal with deletion, we conservatively generate a DFA M that accepts $L(M_m)$ to be repeated many times between any character of $L(M_x)$. Formally speaking, M accepts $\{w_0 c_0 w_1 c_1 \dots w_n c_n w_{n+1} \mid c_0 c_1 \dots c_n \in L(M_x), \forall_i, w_i \in L^*(M_m)\}$, where $L^*(M_m)$ denotes the closure of $L(M_m)$. To construct $M = \langle Q, \Sigma, \delta, q_0, F\rangle$, the basic idea is inserting M_m to each state of M_x. $|Q|$ is bounded by $|Q_m| \times |Q_x|$. Depending on M_m, we consider two cases to insert M_m. First, let $\sigma_m = L(M_m) \cap \Sigma$ be the set of accepted single characters. If $\sigma_m \neq \emptyset$, we insert a self loop for each $a \in \sigma_m$ for all $q \in Q_x$, i.e., $\forall q \in Q_x, a \in \sigma_m, \delta(q, a) = q$. Second, let $M'_m = \langle Q'_m, \Sigma, \delta'_m, q'_{m0}, F'_m\rangle$ accept $L(M_m) \backslash \sigma_m$. If $L(M'_m) \neq \emptyset$ (i.e., M_m accepts some words that are not single character), we insert M'_m for all $q \in Q_x$, which can be done by setting (1)$\delta(q, a) = q'$ if there exists $q' \in Q'_m, \delta'_m(q'_{m0}, a) = q'$, and (2) $\delta(q', a) = q$ if there exists $q', q'' \in Q'_m, \delta'_m(q', a) = q''$ and $q'' \in F'_m$.

In sum, preReplace(M_x, M_m, M_r) returns:

- replace(M_x, M_r, $M_m \cup M_r$) if M_r accepts non empty strings, and
- M if M_r accepts an empty string.

4. Stranger: A String Analysis Tool for PHP Programs

We built a tool called Stranger (STRing AutomatoN GEneratoR) based on the string analysis techniques explained above. Stranger uses Pixy [1] as a front end and MONA [2] automata package for automata manipulation. Stranger takes a PHP program as input and automatically analyzes it and outputs the possible XSS and SQL injection vulnerabilities in the program. For each input that leads to a vulnerability, it also outputs an automaton in a dot format that characterizes all possible string values for this input which may exploit the vulnerability, i.e., it outputs the vulnerability signatures. In the following sections we will give a general overview of the tool architecture and the vulnerability and string analysis implementation details.

The architecture of Stranger is shown in Figure 5. The tool consists of three parts. The *PHP Parser* which parses the PHP code and constructs a control flow graph (CFG). The *Taint Analyzer* which performs alias and dependency analyses, builds the dependency graphs, analyzes them and outputs tainted ones in which tainted user input is not properly sanitized. The *String Analyzer* implements vulnerability (forward and backward) analysis on dependency graphs (as described in the previous section) for all sensitive sinks that are found to be tainted by taint analysis. If a sink is found to be secure by the string analyzer (with respect to the specified attack pattern), then it is guaranteed to be secure. If a sink is found to be vulnerable, then backward analysis computes the vulnerability signature.

4.1. Taint Analysis

The first step in our analysis is to parse the PHP program and construct the control flow graph (CFG). 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 dependency analyses are performed and dependency graphs are analyzed to identify tainted ones. If taint analysis reports a dependency graph to be secure, then it is guaranteed to be secure. Otherwise tainted dependency graphs are passed to the string analyzer for more inspection. We restrict the previous definition for dependency graph in our implementation to the ones that have only one sensitive sink node.

4.2. String Analyzer

The string analyzer implements the string analysis technique explained earlier. We will concentrate here on details related to Stranger implementation.

4.2.1. Pre-analysis Preparation. Before starting the analysis process, the dependency graphs are processed to optimize the analysis. First, a new acyclic dependency graph is built in which all cycles are removed. All the nodes in a cycle are replaced by a single strongly connected component node and a mapping is constructed from each SCC node to the set of nodes inside that cycle. Then, the acyclic graph is topologically sorted starting from the sink node towards the root nodes by reversing the edges in the dependency graph. This is possible as we only have one sink node per each dependency graph.

4.2.2. Forward Analysis Implementation. We have partially changed the previously specified algorithm when we implemented it to optimize the computation. The analysis is conducted on the acyclic graph instead of the original one so that we only need to process the nodes that are not in a cycle once. We start from the nodes that are last in the topological order (since the topological order starts from the sink node). We decorate the root nodes (which do not depend on any other nodes) with their initial forward analysis automaton value according to their types as follows:

- If the node represents a user input then we *always* consider it as tainted (untrusted) and initialize it to Σ^*. User input includes values from web forms, database and files.
- If the node represents an uninitialized variable then depending on an option to enable or disable register_globals in PHP we initialize it to Σ^* or ϵ following the PHP semantics in which an uninitialized variable that is used in a string operation is considered to be empty string.

After processing the root nodes, we complete the decoration of the nodes following their topological order. When we hit an SCC node then we switch to a work queue fixpoint computation algorithm based on Algorithm 2, initializing the queue to the predecessors of the SCC node. The only differences in the implementation are:

- We enqueue a node when it is an element in the cycle and its new calculated value changes.
- When we dequeue a node, we calculate the new values for all of its successors.

During the fixpoint computation we need to apply widening to try to avoid infinite computation and reach

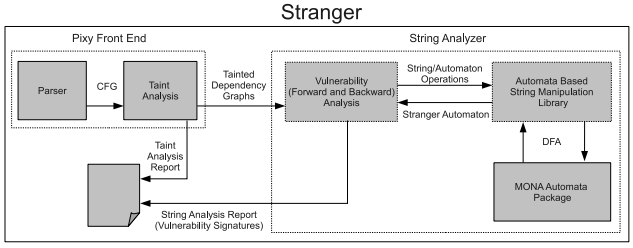

Figure 5. The Architecture of Stranger

a fixpoint. The problem with widening is that analysis looses some precision after its application. So we have added the ability to choose the number of iterations before applying the widening operator. This is to allow for more precision as the computation may converge after a certain number of iterations without the need for the widening. We also have an option to choose when to apply a coarser version of the widening operator to further accelerate the convergence of the fixpoint computation.

4.2.3. Backward Analysis Implementation. The changes to the implementation of backward analysis are pretty similar to the ones done to the forward analysis including topological sorting and the application of widening operator on multiple stages. We also performed an additional optimization by working on a slice of the dependency graph for each input that only includes the nodes on the paths from that input node to the sink.

4.2.4. Modeling PHP String Manipulating Functions. As part of our analysis, we need to model the PHP string manipulating functions for both taint and string analyzers to correctly simulate their semantics during our analysis. For the taint analyzer, we have two types of functions: (1) *Safe functions* that are guaranteed to produce safe string values even when they are mixed within other strings in the program. These functions are considered to be strong sanitization functions that produce untainted output. An example of such functions is $md5$. (2) *Dangerous functions* that may output insecure string values even after they try to sanitize the input. These functions are considered to output insecure tainted string values. These include weak sanitization functions that sanitize the input but do not guarantee its safety when it is mixed within other strings in the program. As an example, suppose that an attacker can exploit a vulnerability in a web application when a string variable reaches a sink with the string value `<script`. Suppose that a sanitization function works by replacing all `<script` with ϵ. In this case, if the attacker passes `<<scriptscript` as an input, the web application will be exploited as the string value that reaches the sink is going to be `<script`. For the string analyzer, functions are modeled using a set of core string operations such as $concat$ and $replace$. For example, function $stripslashes$ is modeled as two consecutive replace operations as: `replace("\'","'")` followed by `replace("\\","\")`. Using the previous operations, we can not guarantee a precise modeling for the semantics of all functions. So we may sometimes output an over approximation of the semantics of the original function. Note that some of the string manipulating functions are modeled to return the same output as the input. These functions do not affect the final value of the string analysis computation. An example for such functions is $strtoupper$ which returns the upper case of the input string. Finally, for replace functions that uses regular expressions such as $preg\_replace$, we use a regular expression parser that supports a large subset of PHP regular expression syntax.

4.3. String Manipulation Library

String manipulation library (SML) handles all core string and automata operations such as $union$, $intersection$, $concatenation$, etc. During vulnerability analysis, all string and automaton manipulation operations that are needed to decorate a node in a dependency graph are sent to SML along with the string and/or automaton parameters. SML, then, executes the operation and returns back the result as an automaton. A Java class called $StrangerAutomaton$ has been used as the type of the parameters and results. The class follows a well defined interface so that other automaton 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 any other string analysis tools for any other language. $StrangerAutomaton$ encapsulates $libstranger.so$ shared library that is writ-

ten in C and has the actual string manipulation code. We used JNA (Java Native Access) to bridge the two languages. The core string and automaton operations are written in C to get a faster computation and a tight control on memory. Stranger, also, has 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. This can be generalized to produce a higher intermediate language that can be used with other string analysis backends that can not be plugged directly into Stranger.

5. Experiments

We experimented with Stranger on a number of benchmarks 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 taint analyzer automatically generates the tainted dependency graphs and identifies that all of them may be vulnerable. In Table 1, we show the result of the taint analysis and some data about these graphs: #sinks indicates the number of sensitive sinks, #inputs indicates the number of `input` nodes. Since the application is identified as vulnerable by taint analysis, both values are at least one. #literals is the sum of the length of constant strings that are used in the graph. Note that these dependency graphs are built for sensitive sinks where unrelated parts have been shrunk. Hence, their sizes are much smaller than the original programs.

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

Table 1. Dependency Graphs

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 character in ASCII. The performance of our vulnerability analysis is shown in Table 2. The backward analysis dominates the execution time from 77% to 96%. Taking a closer look, Table 3 shows the frequency and execution time of each of the string manipulating functions. PRECONCAT (including prefix and suffix) consumes a large portion, particularly for (4) `proManager-0.72` that has a large size of constant literals involved. One reason is generating concatenation transducers during the computation. Note that the transducer has 3-tracks and uses 24 bits to encode its alphabet. On the other hand, our computation does not suffer exponential blow-up as expected for explicit DFA representation. This shows the advantage of using symbolic DFA representation (provided by the MONA DFA library), in which transition relations of the DFA are represented as Multi-terminal Binary Decision Diagrams (MBDDs).

	total time(s)	fwd time(s)	bwd time(s)	mem(kb)
1	0.569	0.093	0.474	2700
2	3.449	0.124	3.317	5728
3	1.087	0.248	0.836	18890
4	16.931	0.462	16.374	116097

Table 2. Total Performance

	CONCAT	REPLACE	PRECONCAT	PREREPLACE
	#operations/time(s)			
1	6/0.015	1/0.004	2/0.411	1/0.004
2	19/0.082	1/0.004	11/3.166	1/0.0
3	22/0.038	4/0.112	2/0.081	4/0.54
4	14/0.014	12/0.058	26/11.892	24/3.458

Table 3. String Function Performance

Finally, Table 4 shows the data about the DFAs that Stranger generated. Reachable Attack is the DFA that accepts all possible attack strings at the sink node. Vulnerability Signature is the DFA that accepts all possible malicious inputs that can exploit the vulnerability. We closely look at the vulnerability signature of (1) `MyEasyMarket-4.1`. The signature actually accepts α^* `<`α^* `s`α^* `c`α^* `r`α^* `i`α^* `p`α^* `t`α^* with respect to the attack pattern Σ^* `<script`Σ^*. α is the set of characters, e.g., `!`, that are deleted in the program. An input such as `<!script` can bypass the filter that rejects Σ^* `<script`Σ^* and exploit the vulnerability. This shows that simply filtering out the attack pattern can not prevent its exploits. On the other hand, the exploit can be prevented using our vulnerability signature instead.

It is also worth to note that both vulnerability signatures of (2) `PBLguestbook-1.32` accept arbitrary strings. By manually tracing the program, we find that both inputs are concatenated to an SQL query string without proper sanitization. Since an input can be any string, the pre-image of one input is the prefix of Σ^* `OR '1'='1'`Σ^* that is equal to Σ^*, while the pre-image of another input is the suffix of Σ^* `OR '1'='1'`Σ^* that is also equal to Σ^*. This case shows a limitation in our approach. Since we do not model the relations among inputs, we can not specify the condition that one of the inputs must contain `OR '1'='1'`.

	Reachable Attack (Sink)		Vulnerability Signature (Input)	
	#states	#bdd nodes	#states	#bdd nodes
1	24	225	10	222
2	66	593	2	9
			2	9
3	29	267	92	983
4	131	1221	57	634
	136	1234	174	1854
	147	1333	174	1854

Table 4. Attack and Vulnerability Signatures

6. Related Work

Due to its importance in security, string analysis has been widely studied. Christensen, Møller and Schwartzbach [5] proposed a grammar-based string analysis (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. [6] use this grammar-based string analysis technique to check for errors in dynamically generated SQL query strings in Java-based web applications [5]. Christodorescu et al. [7] 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 [10], [11], [12], [13]. Shannon et al. [12] 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 [10] support string assignment and validation operations. Fu et al. [13] 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 [8] 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. [9] combine taint propagation with Minamide's string analysis [8] to detect SQL injections, and identify several vulnerabilities in real-world web applications written in the PHP language.

Wassermann et al. [14] use string analysis in test input generation for Web applications. Their approach is based on concolic execution [15], where results of a concrete execution is used to collect constraints on program execution. These constraints are then used to generate new test cases. They use an automata based backward image computation similar to our backward analysis 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.

None of the tools mentioned so far address the vulnerability signature generation problem. There has been earlier work on vulnerability signature generation [16], [17], [18]. The techniques discussed in [16] and [17] require an input that exploits a vulnerability (i.e., an exploit) in order to generate the vulnerability signatures. For example, in [16], this is obtained by running an instrumented version of the program. Our approach does not need an exploit as input since we combine forward and backward symbolic analysis. The approach presented in [18] is a backward analysis similar to second phase of our analysis. However, they require loop invariants to be provided by the user in order to handle loops whereas we use an automated approach based on widening. Also, they focus on weakest precondition computation for binary programs. None of the earlier results on vulnerability signature generation [16], [17], [18] focus on string manipulation operations. Instead, they use existing symbolic execution engines which cannot handle the string manipulation operations that we focus on in this paper. In order to analyze vulnerabilities of PHP applications it is necessary to handle string manipulation operations faithfully as we do in our work.

The techniques proposed in this paper build on our earlier results on string analysis reported in [3], [19]. These earlier results only discuss forward symbolic analysis and do not address vulnerability signature generation problem. Our key contributions in this current paper are 1) The backward symbolic analysis based on backward image computation for string operations such as concatenation and replacement, 2) A new approach to vulnerability signature generation problem that combines symbolic forward and backward analysis, 3) A tool that implements our symbolic analysis techniques and combines it with a PHP front end.

7. Conclusion

We presented symbolic string analysis techniques for identifying vulnerabilities and vulnerability signatures. Our approach is based on automata-based symbolic forward and backward reachability computations. We implemented our approach in a tool for automated analysis of PHP programs. Our tool successfully finds vulnerabilities in existing web applications and generates vulnerability signatures identifying how these vulnerabilities can be eliminated.

References

[1] N. Jovanovic, C. Krügel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, 2006, pp. 258–263.

[2] BRICS, "The MONA project," http://www.brics.dk/mona/.

[3] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra, "Symbolic string verification: An automata-based approach," in *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN 2008)*, 2008, pp. 306–324.

[4] C. Bartzis and T. Bultan, "Widening arithmetic automata," in *Proceedings of the 16th International Conference on Computer Aided Verification*, 2004, pp. 321–333.

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium, SAS '03*, ser. LNCS, vol. 2694. Springer-Verlag, June 2003, pp. 1–18.

[6] C. Gould, Z. Su, and P. Devanbu, "Static checking of dynamically generated queries in database applications," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 645–654.

[7] M. Christodorescu, N. Kidd, and W.-H. Goh, "String analysis for x86 binaries," in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2005)*. ACM Press, Sep. 2005.

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

[9] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[10] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006, pp. 13–13.

[11] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh, "A practical string analyzer by the widening approach," in *APLAS*, 2006, pp. 374–388.

[12] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *TAICPART-MUTATION '07: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–22.

[13] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A static analysis framework for detecting sql injection vulnerabilities," in *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1- (COMPSAC 2007)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 87–96.

[14] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su, "Dynamic test input generation for web applications," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, 2008, pp. 249–260.

[15] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2005)*, 2005, pp. 263–272.

[16] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*, 2007, pp. 117–130.

[17] D. Brumley, J. Newsome, D. X. Song, H. Wang, and S. Jha, "Towards automatic generation of vulnerability-based signatures," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, 2006, pp. 2–16.

[18] D. Brumley, H. Wang, S. Jha, and D. X. Song, "Creating vulnerability signatures using weakest preconditions," in *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF 2007)*, 2007, pp. 311–325.

[19] F. Yu, T. Bultan, and O. H. Ibarra, "Symbolic string verification: Combining string analysis and size analysis," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2009)*, 2009, pp. 322–336.