Patching Vulnerabilities with Sanitization Synthesis *

Fang Yu
National Chengchi University
yuf@nccu.edu.tw

Muath Alkhalaf
UC Santa Barbara
muath@cs.ucsb.edu

Tevfik Bultan
UC Santa Barbara
bultan@cs.ucsb.edu

ABSTRACT

We present automata-based static string analysis techniques that automatically generate sanitization statements for patching vulnerable web applications. Our approach consists of three phases: Given an attack pattern we first conduct a vulnerability analysis to identify if strings that match the attack pattern can reach the security-sensitive functions. Next, we compute vulnerability signatures that characterize all input strings that can exploit the discovered vulnerability. Given the vulnerability signatures, we then construct sanitization statements that 1) check if a given input matches the vulnerability signature and 2) modify the input in a minimal way so that the modified input does not match the vulnerability signature. Our approach is capable of generating relational vulnerability signatures (and corresponding sanitization statements) for vulnerabilities that are due to more than one input.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model Checking, Reliability*

General Terms

Verification, Security

Keywords

Sanitization Synthesis, String Analysis, Automata

1. INTRODUCTION

Web applications are notorious for security vulnerabilities that can be exploited by malicious users. Due to global accessibility of web applications, malicious users all around the world can exploit a vulnerable web application and cause serious damage. So, it is critical that vulnerabilities are not only *discovered* fast, but they are also *repaired* fast.

In this paper, we present techniques for automatically detecting and patching string related vulnerabilities in web applications. We start with a set of attack patterns (regular expressions) that characterize possible attacks (either taken from an attack pattern specification library or written by the web application developer). Given an attack pattern, our string analysis approach works in three phases:

Phase 1: Vulnerability Analysis: First, we use automata-based static string analysis techniques to determine if the web application is vulnerable to attacks characterized by the given attack pattern and generate a characterization of the potential attack strings if the application is vulnerable.

Phase 2: Vulnerability Signature Generation: We then project these attack strings to user inputs and compute an over-approximation of all possible inputs that can cause an attack. This characterization of potentially harmful user inputs is called the *vulnerability signature* for a given attack pattern.

Phase 3: Sanitization Generation: Once we have the vulnerability signature, we automatically synthesize patches that eliminate the vulnerability using two strategies:

- *Match-and-block:* We insert *match statements* to vulnerable web applications and halt the execution when an input that matches a vulnerability signature is detected.
- *Match-and-sanitize:* We insert both match and *replace statements* to vulnerable web applications. When an input that matches a vulnerability signature is detected, instead of halting the execution, the replace statement is executed. The replace statement deletes a small set of characters from the input such that the modified string no longer matches the vulnerability signature.

For vulnerability analysis, we use a forward symbolic reachability analysis that computes an over-approximation of all possible values that string variables can take at each program point. We use deterministic finite automata (DFAs) to represent values that string expressions can take. Intersecting the results of the forward analysis at sinks (i.e., sensitive functions that can cause a vulnerability) with the attack pattern gives us the potential attack strings if the program is vulnerable. If the intersection is empty, then the program is not vulnerable with respect to the given attack pattern.

We use two different techniques for vulnerability signature generation. In the first one, we start with the DFA that represents the attack strings at the sink, and then use a

*This research is funded by the NSC grant 99-2218-E-004-002-MY3, and the NSF grants CCF-0716095 and CCF-0916112.

ICSE'11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

backward symbolic reachability analysis to compute an over-approximation of all possible inputs that can generate those attack strings. The result is a DFA that characterizes the vulnerability signature for the given attack pattern.

However, this approach does not work for vulnerabilities that are due to more than one input. For example, if an attack string is generated by concatenating two input strings, it might not be possible to prevent the attack by blocking only one of the inputs, since a string coming from one input can lead to an attack if it is concatenated with a suitably constructed string coming from another input. In such cases the vulnerability signature can include all possible input strings. Using such a vulnerability signature for automated patch generation would mean blocking or erasing all the user input, which would make the web application unusable. However, if we can do an analysis that keeps track of the relationships among different string variables, then we can block only the combinations of input strings that lead to an attack string.

We present a relational vulnerability signature generation algorithm based on multi-track deterministic finite automata (MDFA). An MDFA has multiple tracks and reads one symbol for each track in each transition. I.e., an MDFA recognizes *tuples of strings* rather than a single string. We use a forward symbolic reachability analysis where each generated MDFA has one track for each input variable and represents the relation between the inputs and the string expression at a particular program point. Intersecting the MDFA at a sink with the attack pattern and projecting the resulting MDFA to the input tracks gives us the vulnerability signature. The vulnerability signature MDFA accepts all combinations of inputs that can exploit the vulnerability.

Once we generate the vulnerability signature we generate match and replace statements based on the vulnerability signature. The match statement basically simulates the vulnerability signature automaton and reports a match if the input string is accepted by the automaton. In the match-and-block strategy this is all we need, and we halt the execution if there is a match. In the match-and-sanitize strategy, however, we also generate a replace statement that will modify the input so that it does not match the vulnerability signature. Since inputs that match the vulnerability signature may come from normal, non-malicious users (who, for example, may have accidentally typed a suspicious character), it is preferable to change the input in a minimal way. We present an automata theoretic characterization of this *minimality* and show that solving it precisely is intractable. We show that we can generate a replace statement which is close to optimal in practice by adopting a polynomial-time min-cut algorithm.

We implemented all the techniques mentioned above for PHP programs on top of our string analysis tool called Stranger (STRing AutomatoN GEneratoR) [17]. Stranger uses the front-end of Pixy [11], a vulnerability analysis tool for PHP that performs taint analysis to identify potentially vulnerable sinks (sinks that depend on external inputs) and generates dependency graphs that show how the external inputs flow into the sinks. Stranger then performs string analysis on the dependency graphs. Stranger also uses the automata package of MONA tool [3] to store the automata constructed during string analysis symbolically.

Related Work: Due to its importance in security, string analysis has been widely studied. One influential approach has been grammar-based string analysis that statically computes an over-approximation of the values of string expressions in Java programs [4] which has also been used to check for various types of errors in Web applications [8, 13, 15, 9]. In [13, 15], multi-track DFAs, also known as *transducers*, are used to model replacement operations. There are also several recent string analysis tools that use symbolic string analysis based on DFA encodings [14, 7, 18]. Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs [14, 7]. HAMPI [12] is a bounded string constraint solver that searches for a string that satisfies a given set of string constraints by bounding the string length. This type of bounded analysis cannot be used for sound string analysis whereas the string analysis techniques we present in this paper are sound. In our earlier work, we have used single-track DFA based symbolic reachability analysis to verify the correctness of string sanitization operations in PHP programs [18, 17]. Our preliminary results on generating (non-relational) vulnerability signatures using single-track DFA were reported earlier [16]. Recently, we also reported our results on foundations of string analysis using multi-track automata [19]. These earlier results do not address the sanitization synthesis problem we discuss in this paper. Moreover, to the best of our knowledge this is the first paper that presents a relational vulnerability signature (i.e., a vulnerability signature that involves more than one input) generation technique for strings.

There has been previous work on automatically generating filters for blocking bad input [6]. Although this is similar to our match-and-block strategy, there are several significant differences with our work. First, earlier work [6] focuses on buffer-overflow vulnerabilities which are different than the string vulnerabilities we investigate here. Second, in earlier work [6] the generation of filters is done starting with an existing exploit, whereas we start with an attack pattern. Finally, unlike prior results, in this paper, we generate sanitization statements that repair bad inputs using the match-and-sanitize strategy. We also give an automata-theoretic characterization of the match-and-sanitize strategy, prove that generating optimum modifications is an intractable problem, and present a heuristic approach based on a min-cut algorithm.

```
1 <?php
2    $name = $_GET["name"];
3    $out = "NAME : " $name;
4    echo $out;
5 ?>
```

Figure 1: A Simple Example

2. AN OVERVIEW

Consider the PHP script shown in Figure 1. This script starts with assigning the user input provided in the `_GET` array to the variable `name` in line 2. It concatenates a constant string with variable `name` and assigns it to another variable `out` in line 3. Then it simply outputs the variable `out` using the `echo` statement in line 4.

The `echo` statement in line 4 is a sink statement since it can contain a Cross Site Scripting (XSS) vulnerability. For example, a malicious user can provide an input that contains the string constant `<script` and execute a command leading to a XSS attack. In order to prevent this vulnerability, it is necessary to *sanitize* the user inputs before using them

in an `echo` statement. In the rest of this section we give an overview of how our approach automatically detects this vulnerability and generates the sanitization statement. Let us assume that the attack pattern for this vulnerability is specified using the following regular expression $\Sigma^* < \Sigma^*$ (where Σ denotes any ASCII character).

Vulnerability Analysis: We first perform a forward symbolic reachability analysis that uses one DFA for each variable at each program point to represent the set of values that the variables can take. During forward analysis we iteratively update these DFAs by computing post-conditions (forward image) of program statements. For example our post-condition computation for an assignment statement takes a set of DFAs characterizing the values of the string variables at the right-hand-side of the assignment (before the assignment is executed) as input, and returns a DFA characterizing the possible values of the left-hand-side variable after the assignment statement is executed.

During forward analysis we characterize all the user input as Σ^*, i.e., the user can provide any string as input. Any variable that is assigned an input is represented by a DFA that accepts the language Σ^* at the next program point after the assignment. For example for the small script shown in Figure 1, our forward analysis will generate a DFA for the variable `name` at the beginning of statement 3 that accepts the language Σ^*. Computing the post-condition of the statement 3 will generate a DFA for the variable `out` at the beginning of statement 4 that accepts the language `NAME :` Σ^*. When our symbolic reachability analysis reaches a fixpoint each string variable at each program point is associated with a DFA that characterizes all possible values that variable can take at that program point. Our analysis is conservative in the sense that the resulting DFAs accept an over-approximation of all possible values of the variables they represent. Note that approximation is inevitable since string analysis problem is undecidable [19].

When our forward analysis converges, we take the intersection (using automata product) of the language of the DFA that corresponds to the string expression at the sink statement with the attack pattern. In our running example statement 3 is a sink statement, and the DFA that corresponds to the string expression at line 4 (which is simply the variable `out`) accepts the language `NAME :` Σ^*. When we take the intersection of this language with the attack pattern we obtain an automaton that accepts the language `NAME :` $\Sigma^* < \Sigma^*$. This automaton characterizes all possible attack strings at the sink statement. Since the language of this automaton is not empty, we know that the program is vulnerable.

Vulnerability Signature Generation: Next, we figure out which input values can create the attack strings at the sink statement. In our single-track DFA based approach, this is done with a backward symbolic reachability analysis. We start with the DFA that characterizes the attack strings (i.e, the DFA we compute at the end of the vulnerability analysis) and propagate the results backwards until we reach an input. During backward analysis we iteratively update these DFAs by computing pre-condition (backward image) of program statements. For example our pre-condition computation for an assignment statement takes a DFA characterizing the values of the string variable at the left-hand-side of the assignment (after the assignment is executed) as input, and returns a set of DFAs characterizing the possible values of the variables that are at the right-hand-side of the assignment before the assignment statement is executed. For the example shown in Figure 1, our backward analysis computes the pre-condition for the assignment statement in line 3 and generates a DFA for the variable `name` at the end of statement 3 that accepts the language $\Sigma^* < \Sigma^*$. When we compute the pre-condition of the assignment statement in line 2 we reach an input and generate the vulnerability signature for the input `_GET["name"]` as a DFA that accepts the language $\Sigma^* < \Sigma^*$.

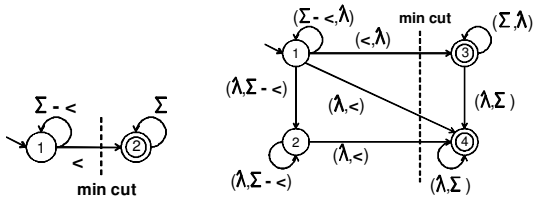
```
1    <?php
1.1    if (preg_match(
         '/([=-\xfd]|[\x00-;])*<([\x00-\xfd])*/',$_GET["name"]))
1.2      die("Invalid input");
2      $name = $_GET["name"];
3      $out = "NAME : " . $name;
4      echo $out;
5    ?>
```

(a) Patch 1 using match-and-block strategy

```
1    <?php
1.1    if (preg_match(
         '/([=-\xfd]|[\x00-;])*<([\x00-\xfd])*/',$_GET["name"]))
1.2      $_GET["name"] =
              preg_replace('/</',"",$_GET["name"]);
2      $name = $_GET["name"];
3      $out = "NAME : " . $name;
4      echo $out;
5    ?>
```

(b) Patch 2 using match-and-sanitize strategy

Figure 2: Patches for the example in Figure 1



(a) (b)

Figure 3: Vulnerability signatures

Sanitization Generation: The last phase of our analysis generates a patch that removes the vulnerability. The vulnerability signature gives an over-approximation of all possible input values that can exploit the vulnerability. Hence, if we do not allow input values that match the vulnerability signature then we can remove the vulnerability. In our *match-and-block* strategy we generate a patch that simply checks if the input string matches the vulnerability signature. If it does, it halts the execution without executing the rest of the script. The patch generated for the small example in Figure 1 based on the vulnerability signature $\Sigma^* < \Sigma^*$ and using the match-and-block strategy is shown in Figure 2(a). Note that the patched script will block any input string that contains the symbol <.

In our *match-and-sanitize* strategy, instead of blocking the execution, we modify the input in a minimal way to guarantee that the modified input cannot lead to any attack strings. We do this by analyzing the vulnerability signature DFA. Consider the DFA for the vulnerability signature $\Sigma^* < \Sigma^*$ shown in Figure 3(a) (we use $\Sigma - <$ to indicate any symbol other than <). Our goal is to find a minimal set of characters, such that if we remove those characters from a given string, the resulting string will not be accepted by the DFA. As we discuss in Section 3, this corresponds to finding

```
1 <?php
2    $title = $_GET["title"];
3    $name = $_GET["name"];
4    $out = "NAME : " . $title . $name;
5    echo $out;
6 ?>
```

Figure 4: Another Simple Example

a cut in the graph defined by the states and the transitions of the DFA, i.e., finding a set of edges such that when we remove them, there are no paths left in the graph from the initial state of the DFA to a final state. Note that each edge of the DFA is labeled with a symbol. After we find a cut, if we take the union of the symbols of the edges in the cut, we obtain a set of symbols such that any string accepted by the DFA must include at least one of the symbols in that set.

We use a min-cut algorithm to compute a cut that contains minimum number of edges. Then we generate a patch that deletes all the characters from the input that appear on the edges included in the cut set. For the DFA shown in Figure 3(a), the min-cut algorithm returns the single edge labeled with the symbol $<$. So we generate a patch that deletes all the $<$ symbols from the input as shown in Figure 2(b). Note that, unlike the patch shown in Figure 2(a), the patch generated based on the match-and-sanitize strategy continues to execute the script after the sanitization.

Relational Vulnerability Signature Generation: Consider the simple script shown in Figure 4. This example is similar to the one shown in Figure 1 with one significant difference: there are two input variables that both contribute to the string expression used at the sink statement at line 5.

Assume that we use our single-track automata based analysis described above to analyze this script. The set of attack strings generated for the sink statement at line 5 will again be: `NAME :` $\Sigma^* < \Sigma^*$. However, the result of the backward analysis will be different. The crucial step is the pre-condition computation for the statement in line 4. The input to this pre-condition computation will be a DFA that accepts the attack strings characterized by the regular expression given above. The result of the pre-condition computation will generate two DFAs, one for the variable `name` and one for the variable `title`, and these DFAs will characterize all possible values these two variables can take just before the execution of statement in line 4 that can lead to generation of an attack string at the sink statement in line 5. When we do this pre-condition computation we get two DFAs that accept the same language Σ^*, i.e., any value of either variable can lead to an attack string. Although this is a sound approximation it fails to capture the information that *at least one of these variables should contain the character* $<$. Note that this condition cannot be expressed as a constraint on an individual variable, it identifies a *relation* between the two string variables.

Our relational analysis uses a single multi-track automaton (MDFA) for each program point to capture the relationship between the input values and possible values of string expressions in the program. We use a forward analysis that operates on the dependency graph. We show the dependency graph for the example from Figure 4 in Figure 5. We write the string expression in the program that corresponds to each node in the dependency graph to the left side of the node and also give the line number. Our analysis starts from the input nodes and traverses the dependency graph while generating one MDFA for each internal node of the dependency graph. Each MDFA has one track for each input variable and one track for the string expression that corresponds to that node, and represents the relation between them. In Figure 5 we show a string constraint on the right side of each internal node. That string constraint characterizes the set of strings accepted by the MDFA for that node. For example, for node $n3$, the string constraint is $n3 = i1.i2$ which indicates that the string expression that corresponds to node $n3$ is equal to the concatenation of input $i1$ and input $i2$.

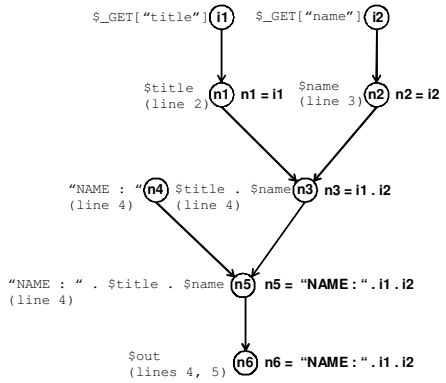


Figure 5: Dependency graph

When the analysis reaches a sink node, we intersect the track that corresponds to the string expression for the sink node (in our example this would be the track that corresponds to node $n6$) with the attack pattern DFA (by extending the attack pattern DFA to an MDFA by adding extra tracks that accept all strings). After the intersection, we project away the track for the sink node, leaving only the tracks for the input nodes. The resulting MDFA represents the relational vulnerability signature. For our example, the vulnerability signature MDFA is shown in Figure 3(b) (where each transition is marked with two symbols, one for each track, and if a track is marked with the symbol λ then that means that no symbol from that track is consumed when that transition is taken). Note that this automaton accepts tuples of strings, where either the first string in the tuple or the second string in the tuple contains at least one $<$ symbol.

The patches shown in Figure 2 for the single input case are generated by converting the standard DFA representation to a regular expression and then using the PHP `preg_match` function to generate the match part of the patch. For the relational case, when we generate two regular expressions, one for each input, from the automaton shown in Figure 3(b), we again get Σ^* for both inputs, so all inputs match. This is acceptable if we use the match-and-sanitize strategy since, although all the input strings will be considered potentially vulnerable, only a small set of symbols that relate to the vulnerability will be replaced. For example, the patch generated using this approach for the example in Figure 4 is shown in Figure 6. However, if we use the match-and-block approach using the regular expression Σ^*, we will block all the inputs which is not acceptable. As we discuss in Section 3, in such cases it is necessary to generate match statements that use automata simulation instead of automata to regular expression conversion.

In order to generate the sanitization statements from relational vulnerability signatures, we find a min-cut in the vulnerability signature MDFA as we did for the single-track

```
1 <?php
1.1    if (preg_match('/([\x00-\xfd])*/', $_GET["title"])
       and preg_match('/([\x00-\xfd])*/', $_GET["name"])) {
1.2      $_GET["title"] =
               preg_replace('/</',"",$_GET["title"]);
1.3      $_GET["name"] =
               preg_replace('/</',"",$_GET["name"]); }
2      $title = $_GET["title"];
3      $name = $_GET["name"];
4      $out = "NAME : " . $title . $name;
5      echo $out;
6 ?>
```

Figure 6: Patch for the example from Figure 4

case. Then, for each track, we take the union of the symbols on that track for all the edges in the min-cut. In order to sanitize the input we need to remove the symbols for each track from the input that corresponds to that track. For example, based on the min-cut shown in Figure 3(b), we need to delete the symbol $<$ both from the inputs `_GET["name"]` and `_GET["title"]`. The automatically generated replace statements for this example are shown in Figure 6.

3. SANITIZATION GENERATION

In this section we describe how we generate sanitization statements given a vulnerability signature that is characterized either as a standard single-track automaton (DFA) or a multi-track automaton (MDFA). We discuss the details of vulnerability signature generation in later sections.

In order to implement the match-and-block and match-and-sanitize strategies we need to generate code for the *match* and *replace* statements.

Match Generation: There are two ways of doing matching: 1) *Regular-expression-based matching:* Generate a regular expression from the vulnerability signature automaton and then use the PHP function `preg_match` to check if the input matches the generated regular expression, or 2) *Automata-simulation-based matching:* Generate code that, given an input string, simulates the vulnerability signature automaton to determine if the input string is accepted by the vulnerability signature automaton, i.e., if the input string matches the vulnerability signature.

We first tried the regular-expression-based matching approach. However, this approach ends up being very inefficient due to the implementation of `preg_match` in PHP. The alphabet of the vulnerability signature automata consists of the 256 ASCII characters and the vulnerability signature automata can have a large number of states if there are a lot of complex string manipulation operations in the code. In one of the examples we analyzed the vulnerability signature automaton consists of 811 states. The size of the regular expression generated from the vulnerability signature automaton can be exponential in the number of states of the automaton [10]. Hence, we may end up with very large regular expressions. Moreover, the `preg_match` function in PHP does not only check if a given input matches the given regular expression but it also computes all the substrings that match the parenthesized subexpressions of the given regular expression. Since the DFA to regular expression conversion algorithm can generate a lot of parenthesized subexpressions, this means that the `preg_match` function will do a lot of unnecessary extra work during the match, resulting with an inefficient match implementation.

In order to do efficient matching we use the DFA simulation algorithm which has linear time complexity [10]. Given the vulnerability signature DFA, we generate a function that takes a string as input, simulates the DFA, and returns true if the DFA accepts the string or false otherwise. We insert the match function instead of the `preg_match` statements shown in the patches in Figures 2 and 6.

For the relational vulnerability signatures, we use a similar approach. Given a relational vulnerability signature characterized as an MDFA, we generate code that simulates the MDFA during the match generation. The MDFA simulation algorithm is similar to the DFA simulation algorithm, it just keeps a separate pointer for each input string to keep track of how much of each track is processed at any given time and advances the state of the MDFA based on the tuples of input symbols and the transition relation of the MDFA. The simulation time for MDFA is linear in the total length of the input strings.

Replace Generation: For the match-and-sanitize strategy, our automated sanitization generation algorithm takes the vulnerability signature automaton as input, and it generates a replace statement that modifies a given input string in such a way that the modified string is not accepted by the the vulnerability signature automaton (meaning that the modified string cannot cause an attack). We modify the input strings by just deleting a set of characters using the `preg_replace` function (our approach can be extended so that escape characters can be inserted in front of a set of characters rather than deleting them). In order to prevent extensive modification to the input, the set of characters to be deleted should be as small as possible. The question is how can we identify the set of characters to be deleted?

First, we will formalize this problem in automata-theoretic terms. Let $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ denote a DFA where Q is the set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states. $L(M)$ denotes the language accepted by M. We say $S \subseteq \Sigma$ is an *alphabet-cut* of M, if $L(M) \cap L_{\bar{S}} = \emptyset$, where $L_{\bar{S}} = (\Sigma \setminus S)^*$ is the set of all strings that do not contain any character in S. The *min-alphabet-cut* problem is finding the alphabet-cut S_{min}, such that for any other alphabet-cut S, $|S_{min}| \leq |S|$. For the example automaton in Figure 3(a) the min-alphabet-cut is $\{<\}$.

The min-alphabet-cut problem can also be stated in graph-theoretic terms. Given a DFA M, an *edge-cut* of M is a set of transitions $E \subseteq \delta$ such that, if the set of transitions in E are removed from the transition relation δ, then none of the states in F are reachable from the initial state q_0. Let S_E denote the set of symbols of the transitions in E. If E is an *edge-cut* of M then S_E is an *alphabet-cut* of M. Hence, finding the min-alphabet-cut is equivalent to finding an edge-cut with minimum set of distinct symbols. For the example automaton in Figure 3(a) the min-edge-cut is $\{(1, <, 2)\}$, which also corresponds to the min-alphabet-cut.

Note that, if the vulnerability signature DFA accepts the empty string, then there will not be any edge (or alphabet) cut since the initial state would be an accepting state. For the rest of our discussion we will assume that the DFA for the vulnerability signature does not accept the empty string (we can easily handle the cases where it accepts the empty string by first testing if the input string is empty and then inserting a single character to the input if it is).

Theorem: *The min-alphabet-cut problem is NP-hard.*
We prove this by a reduction from the vertex cover problem.

A vertex cover of a graph $G = (V, E)$ is a set of vertices such that each edge of the graph is incident to at least one vertex of the set. The problem of finding a minimum vertex cover is known to be NP-complete. Vertex cover problem can be reduced to the *min-alphabet-cut* problem as follows. Given $G = (V, E)$ we build an automaton $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ with the set of states $Q = E \cup \{q_0, q_F\}$, the initial state q_0, set of final states $F = \{q_F\}$, alphabet $\Sigma = V$, and the transition relation δ defined as follows: $e = (v, v') \in E \Rightarrow (q_0, v, e) \in \delta \wedge (e, v', q_F) \in \delta$. The *min-alphabet-cut* for the automaton M is the minimum vertex cover for the graph G.

Since the min-alphabet-cut problem is intractable, rather than trying to find the optimum solution, we can consider using efficient heuristics that give a reasonably small cut that is not necessarily the optimum solution. In fact, there is a very good candidate for a heuristic solution. Given a DFA M, a *min-edge-cut* of M is an edge-cut E_{min} such that for any other edge-cut E, $|E_{min}| \leq |E|$. Note that the min-edge-cut minimizes the number of edges in the edge-cut whereas the min-alphabet-cut minimizes the set of symbols on the edges in the edge-cut. Interestingly, even though the min-alphabet-cut problem is intractable, there is an efficient algorithm for computing the min-edge-cut. We use the Ford-Fulkerson's max-flow min-cut algorithm [5] to find a min-edge-cut E_{min} where the complexity of the algorithm is $O(|\delta|^2)$. Note that $|S_{min}| \leq |E_{min}|$, i.e., the min-edge-cut provides and upper bound for the min-alphabet-cut. So if the min-edge-cut is small, then the set of distinct symbols on the edges of the min-edge-cut will give us a good approximation of the S_{min}.

Once we compute an alphabet-cut S using our heuristic, we generate a `preg_replace` statement that deletes the symbols in S from the input, making sure that the resulting string does not match the vulnerability signature.

The definition of the min-alphabet-cut problem is different for multi-track automata. Given an n-track MDFA M over $(\Sigma \cup \lambda)^n$, we say an n-tuple $S = (S_1, \ldots S_n)$, where $S_i \subseteq \Sigma$, is an alphabet-cut of M, if $L(M) \cap L_{\bar{S}} = \emptyset$, where $L_{\bar{S}} = (((\Sigma \setminus S_1) \cup \lambda) \times \ldots ((\Sigma \setminus S_n) \cup \lambda)))^*$ is the set of all strings whose i^{th} track does not contain any character in S_i. Let $|S| = |S_1| + \ldots + |S_n|$. The *min-alphabet-cut* problem for a MDFA M is finding the alphabet cut S_{min} of M, such that for any alphabet cut S of M, $|S_{min}| \leq |S|$.

Since min-alphabet-cut is intractable for single-track DFA, it is also intractable for MDFA. We use min-edge-cut also as an approximation for min-alphabet-cut for MDFA. When we find a min-edge-cut, we compute the corresponding multi-track alphabet-cut by computing a set of symbols for each track by collecting the set of distinct symbols (other than λ) on each track on the edges in the min-edge-cut. The resulting alphabet cut is an n-tuple $S = (S_1, \ldots, S_n)$, where each S_i is the set of symbols for track i, i.e., input i. For the example automaton in Figure 3(b), the min-edge-cut is $\{(1, (<, \lambda), 3), (1, (\lambda, <), 4), (2, (\lambda, <), 4)\}$, which also corresponds to the min-alphabet-cut $(\{<\}, \{<\})$.

Once we compute the alphabet-cuts, we generate one `preg_replace` statement for each input variable i, that deletes every symbol in S_i from the input i so that the resulting input strings do not match the vulnerability signature.

4. VULNERABILITY ANALYSIS

In this section we first define the dependency graphs and then describe our vulnerability analysis. 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, e.g., assign the value of the variable associated with n_i to the variable associated with n_j in the program. 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: 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: the target node ($n.t$), the match node ($n.m$), and the replacement node ($n.r$). For each value of $n.t$ it: (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 result in n.

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. For a dependency graph G, we also define $Root(G) = \{n \mid Pred(n) = \emptyset\}$ and $Leaf(G) = \{n \mid Succ(n) = \emptyset\}$.

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

Our vulnerability analysis approximates the set of string values as a regular language and represents them symbolically as a DFA that accepts that 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.

We use a forward symbolic reachability analysis based on a standard work queue algorithm. We iteratively update the automata vector $POST$ until a fixpoint is reached. At line 7, 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 lines 9 and 11, we incorporate two automata-based string manipulating functions [18]:

- CONCAT(DFA M_1, DFA M_2) returns a DFA M that accepts $\{w_1 w_2 \mid w_1 \in L(M_1), w_2 \in L(M_2)\}$.
- REPLACE(DFA M_1, DFA M_2, DFA M_3) returns a DFA M that accepts $\{w_1 c_1 w_2 c_2 \ldots w_k c_k w_{k+1} \mid k > 0, w_1 x_1 w_2 x_2 \ldots w_k x_k w_{k+1} \in L(M_1), \forall_i, x_i \in L(M_2), w_i$ does not contain any substring accepted by $M_2, c_i \in L(M_3)\}$.

At line 15, we incorporate the automata widening operator ∇ [2] to accelerate the fixpoint computation, which ensures termination and returns the *least* fixpoint under certain conditions. Upon termination of the while loop (lines 4 to 20) $POST[n]$ records the DFA whose language includes all possible values that n can take.

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

```
1: Init(POST, PRE);
2: queue WQ := NULL;
3: WQ.enqueue(Root(G));
4: while WQ ≠ NULL do
5:    n := WQ.dequeue();
6:    if n ∈ Root(G) then
7:       tmp := CONSTRUCT(n);
8:    else if n is concat then
9:       tmp : = CONCAT(POST[n.p], POST[n.s]);
10:   else if n is replace then
11:      tmp : = REPLACE(POST[n.t], POST[n.m], POST[n.r]);
12:   else
13:      tmp : = ⋃_{n'∈Pred(n)} POST[n'];
14:   end if
15:   tmp := (tmp ∪ POST[n])∇POST[n];
16:   if tmp ⊈ POST[n] then
17:      POST[n] := tmp;
18:      WQ.enqueue(Succ(n));
19:   end if
20: end while
21: set Vul := {};
22: for each n ∈ Sink do
23:   tmp: = POST[n] ∩ Attk;
24:   if L(tmp) ≠ ∅ then
25:      Vul := Vul ∪ {n};
26:      PRE[n] := tmp;
27:   end if
28: end for
29: return Vul;
```

Algorithm 2 VulSigGen($G, POST, PRE, Vul$)

```
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
32: return PRE;
```

In the for loop (lines 22 to 28), 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 25. 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 26. This information is then passed to our vulnerability signature generation algorithm.

5. VULNERABILITY SIGNATURES

In this section we present our non-relational vulnerability signature generation algorithm (Algorithm 2) which is a backward symbolic reachability computation based on single-track DFAs. For $n \in Vul$, $PRE[n]$ is set to the intersection of $POST[n]$ and $Attk$ during the vulnerability analysis phase 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 in lines 11, 13 and 17 by calling the defined automata-based functions: preConcatPrefix, preConcatSuffix, and preReplace which we define below. 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.

Pre-image Computation: Below, we explain how we compute the following pre-image 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 $M = \text{REPLACE}(M_1, M_2, M_3)$.

Concatenation: To compute the pre-image of concatenation nodes, we introduce concatenation transducers to specify the relation among its output and two input nodes. Transducers are multi-track automata we use for image computation (these are different than the multi-track automata we use for relational vulnerability signature generation in the next Section). A concatenation transducer is a MDFA 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 $w'[2], w'[3] \in \Sigma^*$ to denote the λ-free prefix of $w[2]$ and the λ-free suffix of $w[3]$. We say w is accepted by a concatenation transducer M if $w[1] = w'[2].w'[3]$. Since a concatenation transducer binds the values of different tracks character by character it is able to identify the prefix and suffix relations precisely.

Here we only 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. The pre-image of the suffix node can be computed in a similar way. Let M_x and M_z, accept values of X and Z respectively. preConcatPrefix(M_x, M_z) returns M_y which is constructed using the following steps:

- 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 $\{w \mid w[1] = w'[2].w'[3], w'[3] \in L(M_z)\}$.
- 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.
- Remove λ tails if any to construct M_y.

Replacement: Recall that a `replace` node has three input nodes: target, match, and replacement. Let us 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), then our goal is to compute M_t, given M_x, M_m, and M_r. 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.

6. RELATIONAL SIGNATURES

A relational vulnerability signature M of n inputs is a MDFA over the n-track alphabet Σ^n, defined as $(\Sigma \times \{\lambda\}) \times \ldots \times (\Sigma \times \{\lambda\})$ (n times), where $\lambda \notin \Sigma$ is the special symbol for padding. We further restrict M, so that all tracks are aligned and for any $w \in L(M)$, $w[i] \in \lambda^*\Sigma^*\lambda^*$ $(1 \leq i \leq n)$. Let $w'[i]$ denote the longest λ-free substring of $w[i]$.

Given a dependency graph G, a set of input nodes **In**, a sink node *sink*, and an attack pattern *Attk*, our goal is to generate a relational vulnerability signature M such that: (1) M is a $|\mathbf{In}|$-track MDFA. Each track is associated with an input variable X_n, $n \in \mathbf{In}$. (2) For any word w ($w[i] \in \lambda^*\Sigma^*\lambda^*$), we have $w \in L(M)$ if the following condition holds: if we set $w'[i]$ as the initial value of the input node i and propagate the values of the nodes along with G accordingly, the value of the node *sink* matches the pattern *Attk*. I.e., w identifies the malicious inputs whose combination may exploit the vulnerability.

The algorithm to generate a relational vulnerability signature is shown in Algorithm 3. We perform *forward* fixpoint computation on the dependency graph where `replace` nodes are ignored. Our relational vulnerability signature algorithm is not capable of handling replace statements. However, since we run the vulnerability signature generation after a vulnerability is detected, we argue that it is reasonable to ignore the sanitization statements in the code (which is the typical use for the replace statements). After we generate the relational vulnerability signature, the existing sanitization statements can be commented out and replaced with the automatically generated sanitization statements.

Similar to the other analyses we presented, we use a standard work queue algorithm incorporating the automata widening operator. Each node is associated with a signature, a i+1-track MDFA where the first i tracks are associated with some input variables, e.g., $X_n, n \in \mathbf{In}$, and the last track (output track) is associated with X_o used to represent the values of the current node. More specifically, i $(0 \leq i \leq |\mathbf{In}|)$ is the number of the input variables whose values have been used to construct the values of the current node. We use a MDFA vector S where $S[n]$ is the signature associated with node n and it specifies the relations among the values of the input variables and the values of n.

Initially, for each input node $n \in \mathbf{In}$, $S[n]$ is a 2-track MDFA (associated with X_n and X_o) that accepts the identity relation on X_n and X_o, i.e., the value of the current node is equal to the value of the input variable X_n. For a node $n \in Root(G) \setminus \mathbf{In}$, $S[n]$ is a single-track DFA (associated with X_o) that either accepts Σ^* if n is a `variable` node, or accepts a constant value if n is a `constant` node. I.e., the current value of the node is an arbitrary string or a constant. In both cases, it is not related to any input variable. For the rest, i.e., $n \notin Root(G)$, $S[n]$ accepts an empty set.

After we initialize S at line 1, we perform the fixpoint computation. Between lines 6 and 18, we iteratively update the signature at each node until the queue is empty (reaching a fixpoint). To deal with the union or widening operator on S_1 and S_2 that may be associated with the different sets of input variables, say $\mathbf{X_1}$ and $\mathbf{X_2}$, we extend both tracks to $\mathbf{X}_1 \cup \mathbf{X}_2$ and X_o by padding λs in the added tracks. We then apply standard union or widening to these extended MDFAs.

Below we describe how to concatenate two signatures: ConcatSignature(S_1, S_2), where S_1 is the signature of the prefix node and S_2 is the signature of the suffix node. Let $S_1 = \langle Q_1, \Sigma_1, \delta_1, I_1, F_1 \rangle$ be a MDFA whose tracks are associated with the set of input variables $\mathbf{X}_1$ and X_o where $\Sigma_1 = (\Sigma \cup \lambda)^{|\mathbf{X}_1|} \times \Sigma$. Let S_2 be a MDFA whose tracks are associated with the set of input variables $\mathbf{X}_2$ and X_o where $\Sigma_2 = (\Sigma \cup \lambda)^{|\mathbf{X}_2|} \times \Sigma$. We first extend S_1 and S_2 to two MDFA S_1^λ and S_2^λ that are associated with $\mathbf{X}_1 \cup \mathbf{X}_2$ and X_o. We extend S_1 (prefix) to S_1^λ by adding λ in the added tracks, while we extend S_2 (suffix) to S_2^λ by adding λ in both the added tracks and the common tracks that are also associated with S_1. ConcatSignature(S_1, S_2) returns the $(|\mathbf{X}_1 \cup \mathbf{X}_2| + 1)$-track MDFA that accepts the concatenation of S_1^λ and S_2^λ.

After reaching a fixpoint, at line 19, we intersect the signature of *sink* with the attack pattern on the output track. Let M_{Attk} accepts $\{w \mid w[X_o] \in Attk\}$. This is done by the standard intersection of $S[sink]$ and M_{Attk}. After the intersection, the output track identifies the reachable attack strings, and the input tracks identify all the malicious inputs whose combination can yield an attack string. At line 20, we project away the output track from M, and return the result at line 21 as the relational vulnerability signature of $\langle G, \mathbf{In}, sink, Attk \rangle$.

7. EXPERIMENTS

We first evaluated our approach for XSS vulnerabilities using five known examples. Then we applied our analysis to three open source web applications to search for XSS and SQLI vulnerabilities. In our experiments we used an Intel machine with 3.0 GHz processor and 4 GB of memory running Ubuntu Linux 8.04. We used 8 bits to encode each ASCII character. For the XSS vulnerabilities the sinks include the `printf` and `echo` functions, and for the SQLI vulnerabilities the sinks include the `mysql_query` function. We used the attack pattern Σ^*`<SCRIPT`Σ^* (indicating an embedded script) for the XSS vulnerability and the attack pattern Σ^*`' or 1=1`Σ^* (indicating a *true* condition in a query) for the SQLI vulnerability. Our approach can deal with any

Algorithm 3 RELSIGGEN($G, \mathbf{In}, sink, Attk$)

```
1: INIT(S, G, In);
2: queue WQ := NULL;
3: for n ∈ In ∪ Root(G) do
4:    WQ.enqueue(Succ(n));
5: end for
6: while WQ ≠ NULL do
7:    n := WQ.dequeue();
8:    if n is concat then
9:       tmp : = CONCATSIGNATURE(S[n.p], S[n.s]);
10:   else
11:      tmp : = ⋃_{n'∈Pred(n)} S[n'];
12:   end if
13:   tmp := (tmp ∪ S[n])∇S[n];
14:   if tmp ⊈ S[n] then
15:      S[n] := tmp;
16:      WQ.enqueue(Succ(n));
17:   end if
18: end while
19: M:= S[sink] ∩ M_Attk;
20: Project the output track away from M;
21: return M;
```

attack pattern specified as a regular expression.

For sanitization synthesis our tool generates match statements as a C extension to PHP that simulates the vulnerability signature automaton. For replace statements, it generates a PHP function `preg_replace` to delete characters that are identified by the alphabet-cuts generated from the vulnerability signature automata.

Patching Known Vulnerabilities We first analyzed five benchmarks manually extracted from (1) `MyEasyMarket-4.1` (a shopping cart program), (2) `BloggIT-1.0` (a blog engine), and (3) `proManager-0.72` (a project management system). Each benchmark represents a known XSS vulnerability [1] containing a single sink. The dependency graphs of these benchmarks are rather small (around 20-30 nodes) but include loops, concatenations with large constants, and nested replacements (from customized or PHP built-in sanitization routines) and represent typical string manipulation operations in PHP web applications.

As expected, our vulnerability analysis reported that all benchmarks are vulnerable and returned the corresponding vulnerability signature for each input. Benchmark 3 has two user inputs contributing to the vulnerability and we generated both single-track (denoted as 3S) and multi-track signatures (denoted as 3R). Table 1 shows the number of edges in the min-edge-cut for the vulnerability signature automata we computed, and the alphabet-cuts that correspond to these min-edge-cuts.

Sig.	1	2	3S	3R	4	5
#edges	1	8	4	3	4	4
alp.-cut	{<}	{S,',"}	Σ, Σ	{<}, {<}	{<,',"}	{<,',"}

Table 1: Minimum Edge and Alphabet Cuts

Our results show that our techniques are very effective. As we can see, the min-edge-cut results in a very small alphabet-cut, and many of them (1 and 3R) are the optimum solutions. For benchmark 3, using single-track signatures (3S), gives the alphabet-cut Σ for both inputs. I.e., we need to delete all characters. As we discussed earlier, this is due to the fact that single track signatures can not keep the relation among inputs, and are too coarse (in this case, Σ^*) to synthesize practical sanitization code. On the other hand, the relational signature (3R) gives the optimum solution deleting only '<' from input 1 and input 2.

We favored non-alphanumeric characters while generating the alphabet-cuts by increasing the weights of the alphanumeric characters during the min-cut algorithm (we assume that alphanumeric characters are more likely to represent normal user input and we prefer not to delete them unless necessary), This resulted in having non-alphanumeric characters in all the cuts but one. Finally, existing sanitization routines in the analyzed applications can add some additional characters to the alphabet-cut due to the conservative nature of our analysis that over-approximates the vulnerability signatures. For example, in 2 '' and "" are introduced by the PHP sanitization routine `mysql_real_escape_string`.

Analyzing and Patching Open Source Applications: We applied our analysis to three open source PHP web applications: (1) `Webchess 0.9.0` (a server for playing chess over the internet) (2) `EVE 1.0` (a tracker for players activity for an online game), and (3) `Faqforge 1.3.2` (a document management tool). The sizes of these applications are shown in 2. These applications are downloaded from *sourceforge* and are directly analyzed using the techniques presented in this paper without any manual modification.

	Application	# of PHP files	total loc	# of sinks XSS	# of sinks SQLI
1	Webchess 0.9.0	23	3375	421	140
2	EVE 1.0	8	906	114	17
3	Faqforge 1.3.2	10	534	375	133

Table 2: The Sizes of Analyzed Applications

Table 3 summarizes the results of our XSS and SQLI vulnerability analyses respectively and the performance for signature generation. STRANGER discovered 55 XSS and 61 SQLI vulnerabilities in these applications. In Table 3, (single, 2, 3, 4) indicates the number of detected vulnerabilities that have single input, two inputs, three inputs and four inputs, respectively. For example, there are 20 vulnerabilities detected in Faqforge and they all have single input (denoted as (20, 0, 0, 0)). That is, for each vulnerable sink, we have only one input contributing a value to the sink. We automatically generate multi-track signatures for vulnerabilities that have multiple inputs, and single-track signatures for vulnerabilities that have single input. There are three SQLI vulnerabilities for which we were not able to synthesize sanitization patches due to the large number of inputs and due to a bound in the MONA DFA package implementation that limits the number of tracks we can declare for a given DFA. During the SQLI vulnerability analysis of Faqforge the taint analysis phase does not report any sinks that depend on user inputs, so our tool does not execute any of the string analysis phases and does not report and vulnerabilities.

Based on the results shown in Table 3, the analysis cost seems affordable: the *total* time indicates the total time to analyze all PHP files in these applications from start to the end, which includes pre-processing (parsing, dependency and taint analyses) time and string analysis (vulnerability analysis and vulnerability signature generation) time. It ranges from 8 seconds to 290 seconds. The *fwd* time indicates the total time to detect vulnerabilities from all tainted sinks (determined by first running a taint analysis) in all PHP scripts. The *bwd* time indicates the total time to generate single track signatures for all detected vulnerabilities that have single input, and the *relational* time indicates the total time to generate multi track signatures for all detected vulnerabilities that have multiple inputs.

	# of Vul. (single, 2, 3, 4)	Time (seconds) total	fwd	bwd	relational	Mem (Kb) average
XSS Vulnerability Analysis						
1	(24, 3, 0, 0)	46.08	1.73	0.92	6.30	16850
2	(0, 0, 8, 0)	288.50	6.80	–	127.80	125382
3	(20, 0, 0, 0)	7.87	0.22	0.22	–	9948
SQLI Vulnerability Analysis						
1	(43, 3, 1, 2)	110.7	4.87	12.04	38.03	136790
2	(8, 3, 0, 0)	23.9	1.5	8.47	5.2	17280
3	(0, 0, 0, 0)	6.7	–	–	–	< 1

Table 3: XSS/SQLI Vulnerability Analysis Results

Replace performance: The average time spent in generating the alphabet-cut from the vulnerability signature automata for XSS (SQLI) was 0.06 (0.07) seconds per automaton for Webchess, 0.3 (0.1) seconds per automaton for EVE, and 0.05 seconds per automaton for Faqforge. We notice that mincut for XSS analysis results for EVE took the largest time as it contains only three-input signature automata as apposed to mostly single-input signature automata in the other two.

All of the generated alphabet-cuts contain only a single character per each input. For each XSS single track automata the cut is only the character '<' which is the optimum cut (consequently the optimum sanitization with respect to the attack pattern). Similarly, for the XSS relational signature automata we obtained '<' for each input. The automatically generated sanitization (replace) statements from our analysis were almost the same as the ones that are manually written except that they delete the '<' character instead of replacing it with the HTML entity "<" as is typically done in manual sanitization. For the SQLI vulnerabilities, the results were similar, where the reported cuts contained only the character '=' instead of '<'.

Our analysis produces two artifacts: a PHP extension that contains a `stranger_match_*` function for each vulnerable input, and a set of `preg_replace` statements, one for each vulnerable input. In PHP, user inputs from `$_GET` and `$_POST` are always available at the first program point in the script. This means that we can sanitize the inputs at the first PHP line of the target script. Inserting these calls can easily be automated as we have the file names for each of the input variables along with the variables' names from the parsing phase. Note that we are analyzing PHP scripts statically in a sound manner where we only deal with one script at a time along with all the files it includes. We used the result of our analysis to sanitize the three applications above by placing the automatically generated sanitization statements at the beginning of each vulnerable script. Then we ran our forward vulnerability analysis which reported zero vulnerabilities with regard to the attack pattern mentioned above demonstrating that our analysis sound and guarantees that after the sanitization statements are inserted, sensitive functions will not receive any input that matches the attack pattern.

8. CONCLUSIONS

Most critical security vulnerabilities in web applications are caused by inadequate manipulation of input strings. In this paper we presented a set of techniques that 1) identify vulnerabilities that are due to string manipulation, 2) generate a characterization of inputs that can exploit the vulnerability, and 3) generate sanitization statements that eliminate the vulnerability.

9. REFERENCES

[1] 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.

[2] C. Bartzis and T. Bultan. Widening arithmetic automata. In *CAV*, pages 321–333, 2004.

[3] BRICS. The MONA project. `http://www.brics.dk/mona/`.

[4] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, pages 1–18, 2003.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[6] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: securing software by blocking bad input. In *SOSP*, pages 117–130, 2007.

[7] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao. A static analysis framework for detecting sql injection vulnerabilities. In *COMPSAC*, pages 87–96, 2007.

[8] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *ICSE*, pages 645–654, 2004.

[9] G.Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, pages 171–180, 2008.

[10] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *S&P*, pages 258–263, 2006.

[12] 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.

[13] Y. Minamide. Static approximation of dynamically generated web pages. In *WWW*, pages 432–441, 2005.

[14] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid. Abstracting symbolic execution with string analysis. In *TAICPART-MUTATION*, pages 13–22, 2007.

[15] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, pages 32–41, 2007.

[16] F. Yu, M. Alkhalaf, and T. Bultan. Generating vulnerability signatures for string manipulating programs using automata-based forward and backward symbolic analyses. In *ASE*, pages 605–609, 2009.

[17] F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for php. In *TACAS*, pages 154–157, 2010.

[18] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic string verification: An automata-based approach. In *SPIN*, pages 306–324, 2008.

[19] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *CIAA*, 2010.