Optimal Sanitization Synthesis for Web Application Vulnerability Repair

ABSTRACT

We present a code- and input-sensitive sanitization synthesis approach for repairing string vulnerabilities that are common in web applications. The synthesized sanitization patch modifies the user input in an optimal way while guaranteeing that the repaired web application is not vulnerable. Given a web application, an input pattern and an attack pattern, we use automata-based static string analysis techniques to compute a sanitization signature that characterizes safe input values that obey the given input pattern and are safe with respect to the given attack pattern. Using the sanitization signature, we synthesize an optimal sanitization patch that converts malicious user inputs to benign ones with minimal editing. When the generated patch is added to the web application, it is guaranteed that the repaired web application is no longer vulnerable. We present two refinements to the basic sanitization synthesis algorithm that reduce the runtime sanitization cost significantly. We evaluate our approach on open source web applications using common input and attack patterns, demonstrating the effectiveness of our approach.

Keywords

Sanitization Synthesis, String Analysis, Automata

1. INTRODUCTION

Vulnerabilities that are due to errors in input validation and sanitization code (such as Injection Flaws and Cross-Site Scripting) have continued to be the topmost security risks in web applications in the past decade [23]. Input validation and sanitization code has to make sure that 1) the input string is in the required format, and 2) the strings that reach security sensitive functions (called sinks) are not malicious. Security experts specify malicious strings as *attack patterns*, which are regular expressions that characterize possible attacks. Similarly, application programmers specify input formats using regular expressions, which we call *input patterns*. Simply enforcing the intersection of these patterns on the input does not work since the attack patterns characterize the attack strings at sinks, i.e., at the point where the string value reaches a security sensitive function. Enforcing the input patterns at the sinks does not work either, since the application code can change the application state based on the input string before it reaches a sink, and modifying or rejecting the input at the sink may cause the application to enter in an inconsistent state.

Given the prevalence of erroneous input validation and sanitization in web applications, it would be valuable to have an approach that automatically generates provably correct sanitizers. In order to automatically generate a sanitizer, the proposed analysis must be *code-sensitive*, i.e., it has to take into account how the application code manipulates the input value before it reaches a sink. For example, by re-inserting a character that is deleted during sanitization, an application may reconstruct an attack before the input reaches the sink. In fact, an exploit that utilizes the string manipulation operations in the application code may be able to construct an attack string from an input that does not contain an attack string.

Moreover, an effective sanitizer should prevent attacks while minimizing its effect on users who are not malicious. Hence, sanitizers should be *input-sensitive* and modify each input in a minimal way while still guaranteeing security. Consider a post in a forum:

```
To write an XSS attack such as  <script>alert(2)
</script> in Stack Overflow, use a built-in escaping
mechanism like <c:out> or <h:outputText> to display
your comments.
```

This post is considered harmful with respect to a XSS vulnerability due to the script statement. It could be converted to a benign string with one simple editing, e.g., to escape only the < character in <`script`>. This requires sanitization functions to be *input-sensitive*, being able to identify and modify only malicious parts in the input (with respect to the attack pattern) while keeping the rest unmodified.

In this paper, we present a novel sanitization synthesis approach that is both code and input-sensitive. The sanitizers generated by our approach modify the input in a minimal way and guarantee that 1) the modified input obeys the input pattern, and 2) no string that matches an attack pattern reaches a sink. As shown in Figure 1, our approach consists of two main phases:

Phase 1: Sanitization Signature Generation: Given a web application, an input pattern and an attack pattern (both specified as regular expressions) we first extract dependency graphs for security sensitive functions (sinks) from the web application using static program analysis techniques, where each extracted dependency graph shows how the input values flow to a sink, including all the string operations performed on the input values before they reach the sink. We use automata-based symbolic string analysis techniques [37] where the values that string expressions can take during program execution are represented using deterministic finite automata (DFA). We first conduct a forward symbolic reachability analysis on the dependency graph starting from user inputs (which can be any string value). The forward symbolic reachability analysis computes an over-approximation of all possible string values that can reach the sink and generates a DFA that accepts this set of strings. Intersecting (using automata product) the language of the DFA generated for the sink with the DFA constructed from the attack pattern allows us to determine all possible attack strings with respect to the given attack pattern (which could be empty, meaning that application is not vulnerable). Compared to vulnerability detection techniques that are based on taint analysis [14, 15], the automata-based string analysis takes semantics of string manipulation operations into account, and is able to detect vulnerabilities due to inadequate implementation or use of sanitization functions that

taint analysis would overlook.

In order to generate sanitizer that removes any identified vulnerability, we need to identify the string values at the input that can be malicious. To do so, we conduct a backward symbolic reachability analysis on the dependency graph starting from the sink and the DFA that accepts the attack strings at that sink. Propagating attack strings back to the input node in the dependency graph results in a DFA that characterizes an over approximation of all malicious user inputs, and we call this the *vulnerability signature*. Since vulnerability signature over approximates all malicious user inputs, its complement (i.e., the set of all strings that are not in the vulnerability signature) corresponds to all strings that are safe with respect to the given attack pattern. By taking the intersection of this complement set with the set of strings that match the input pattern, we obtain the *sanitization signature*, i.e., the set of input strings that match the input pattern, and that are guaranteed not to cause any attack strings at the sink. We compute the DFA for the sanitization signature using the automata complement and automata product operations.

Phase 2: Optimal Sanitization Synthesis: In the second phase, we synthesize optimal sanitizers for repairing the vulnerable web application based on the sanitization signatures computed in the first phase. The generated sanitizers are optimal in the sense that they modify the input string in a minimal way. When the synthesized sanitizer is used as a patch to repair the web application (by inserting it to the first program location where the input value is read), the resulting repaired web application is guaranteed to be safe with respect to the given attack pattern.

Given an input string, if the input string is in the sanitization signature, the synthesized sanitizer does not modify the input. If the input string is not in the sanitization signature (which means that it is in the vulnerability signature, hence, it is potentially malicious), the synthesized sanitizer modifies the input to covert it to a benign one (i.e., to a value that is in the sanitization signature).

The synthesized sanitizer is optimal in the sense that the input is modified in a minimal way, where the amount of modification is formalized using the notion of edit-distance. The edit-distance between two strings is the smallest number of operations required to transform one string into the other. Edit-distance is typically used as a similarity measure between two strings; the shorter distance implies that the two strings are more similar. The sanitizers we generate convert a given malicious input to a corresponding benign input with minimal edit-distance. Unlike sanitization approaches that remove all suspicious characters from all input strings [36], the approach we present in this paper is input-sensitive and modifies each input in a minimal way.

In practice, the application developers may want to reject an input string instead of repairing it if the edit distance is too large. This can be easily accommodated in our approach by rejecting an input string if the minimal edit distance is higher than a threshold value determined by the developer. Note that setting the threshold value to zero would convert the generated sanitizer to a validator which rejects the input if it does not match the sanitization signature. Using such a threshold the application developers can control how much modification on the input is allowed by the sanitizer.

Let DFA A denote the sanitization signature automaton and $L(A)$ denote the language (i.e., the set of strings) accepted by A. Let string w be the input. The sanitizer we generate based on A works as follows: Given w, it outputs a string w' in $L(A)$, such that the edit-distance between w and w' is minimal (if w is in $L(A)$ then minimal edit distance is 0 and $w' = w$). We implemented three algorithms for sanitization synthesis. First one is based on the basic minimal edit-distance computation algorithm presented in [31, 16]. However, this algorithm has a high runtime cost and would not be feasible for sanitization in web applications. Second algorithm we implement is an improved version that computes the result in an incremental manner in order to reduce the runtime cost [2]. Third algorithm we implement is a new algorithm that we propose in this paper that precomputes information needed for the incremental algorithm and reduces the runtime overhead further. Let |S| be the size of alphabet and |Q| be the number of states in the sanitization signature. The amount of data store is only 2x|S|x|Q| in the worst case during the pre-computation of sanitization records. We show that our pre-computation algorithm reduces the runtime overhead significantly compared to previous two algorithms.

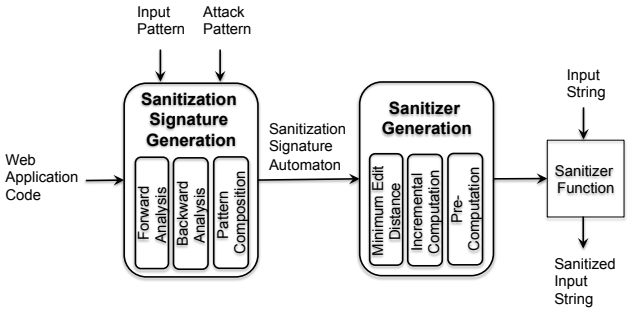


Figure 1: High level overview of the presented approach.

We implemented all the techniques mentioned above for PHP programs on top of the string analysis tool called Stranger (STRing AutomatoN GEneratoR) [35] and incorporated our automated sanitization synthesis algorithms in an online service called Patcher for repairing vulnerable web applications [39]. Given a web application and a set of attack patterns, Patcher generates a PHP file that contains the synthesized sanitization functions and signatures for all the identified vulnerabilities. By simply including this file and inserting patch statements in the code, the developers can repair the vulnerable web application, and eliminate the identified vulnerabilities.

Our automated repair strategy produces sound repairs with a precise criteria (minimum edit distance) on how the input is modified. Our approach is different than source code repair techniques that make syntactic modifications to source code [14], and must be evaluated by the developers in order to check if the modifications to the source code are acceptable. Our repair approach works more like a compiler, and generates code based on a well defined objective with guaranteed semantics. As manual evaluation of the automatically generated machine code by a compiler is unnecessary, manual evaluation of the automatically synthesized patches that our approach produces is also unnecessary. The question that developers can evaluate is the following: Is the minimum-edit distance a useful metric for sanitization? Although we did not conduct an empirical study on developers to address this particular question, our extended experience

in analyzing a large number of sanitization code indicates that, modifying the input in a minimal way is a key criterion in sanitization, and minimum-edit-distance captures this criterion precisely. Note that, it is easy to extend our approach by assigning weights to characters so that in calculating minimum edit distance some characters are less or more likely to be removed. The assignment of these weights is likely to be application specific, so, in the general approach we present in this paper we do not use a weighted approach.

Our main contributions in this paper can be summarized as follows: 1) We introduce the concept of sanitization signature and show how it can be automatically computed in a code-sensitive manner using both input and attack patterns. 2) We combine automata-based string analysis techniques with minimal edit distance algorithms and present a novel automated sanitization synthesis technique that generates input-sensitive sanitizers that modify the input in a minimal way. 3) We propose two efficient algorithms for the editing distance problem with respect to a target automaton (sanitization signature) and show that it can be applied for patching vulnerable user inputs. The first version improves the direct composition algorithm in terms of space and is expected to be also faster for longer input strings. Since run time is crucial in the application, we propose a second version that pre-computes the editing graph for each alphabet symbol and use it to achieve a dramatic speed-up. The cost of the pre-computation is actually low, linear to the size of different alphabet symbols used in the target automaton. The algorithms are interesting on their own right and should be applicable in many different applications.

2. SANITIZATION SIGNATURE GENERATION

In order to generate sanitization signatures, we first extract a dependency graph from the input web application using existing static analysis techniques [15]. The dependency graph shows how input values flow to sinks. Each edge of the dependency graph is labeled with a string manipulation operation denoting how the string values are modified.

Let us define the set of string *Operations* P using an abstract grammar as follows:

$$\begin{aligned} R &::= \emptyset \mid a \mid RR \mid R + R \mid \overline{R} \mid R^* \\ S &::= s_1 \mid s_2 \mid s_3 \mid \cdots \\ P &::= \mathsf{input} \mid R \mid S + S \mid SS \mid S[S \mapsto S] \end{aligned}$$

where $s_1, s_2, \ldots$ denote string expressions and $a \in \Sigma$ is an input symbol. Observe that R is the class of regular expressions. A string operation is either a regular expression, the union $(S + S)$ or concatenation (SS) of string expressions, or the replacement $(S[S \mapsto S])$ of string expressions.

A *dependency graph* is a directed graph $G = \langle V, E, cmd \rangle$ with a vertex labeling function $cmd : V \to P$. An edge $(v, v') \in E$ means that the operation associated with v' depends on the operation associated with v. Each vertex of the dependency graph represents a string expression (that is constructed by a string operation that may use other string expressions, i.e., other vertices in the dependency graph). An input operation input obtains a string from an external source such as a text field in a web page. A vertex associated with a regular expression specifies string constants. Subsequently, a vertex labeled by an input operation or a regular expression has no predecessors (since they do not depend on any other string expressions). In addition to inputs and regular expressions, union, concatenation, and replacement operations can be specified in a vertex of the dependency graph. The final vertex in a dependency graph (i.e., the vertex with no successors) denotes a sink, i.e., a security sensitive function that can be target of an attack, such as execution of an SQL command stored in a string variable.

Let L_{attack} be a regular language of attack strings (specified by the attack pattern) and L_{input} be a regular language of input strings that obey the input format (specified by the input pattern). For each vertex associated with an input operation, we would like to compute the *sanitization signature*, i.e., a regular language such that, when the input value is in that language it is guaranteed that: 1) the input value is in L_{input}, and 2) the sink does not receive a string value that is in L_{attack}.

We compute the sanitization signature using symbolic forward and backward reachability analyses that annotate each vertex of the dependency graph with a deterministic finite automaton (DFA). The DFA annotating a vertex in the dependency graph characterizes the set of string values that the string expression corresponding to that vertex can take during program execution.

In implementing the symbolic forward and backward reachability analysis, we use the automata based pre- and post-image computations for string operations implemented in the Stranger tool [34, 35, 37] and an automata based widening operation that guarantees convergence [4, 37].

Let $G = \langle V, E, cmd \rangle$ be a dependency graph. During forward analysis we construct a deterministic finite automaton for each vertex $v \in V$ based on the operation $cmd(v)$ associated with v and the DFAs that annotate the predecessors of v. Hence, each step of the forward analysis corresponds to a post-image computation for a string operation. When $cmd(v) = \mathsf{input}$, we would like to represent an arbitrary input. Hence we construct a DFA accepting an arbitrary string in Σ^*. Similarly, when $cmd(v) = R$ for some regular expression R, we construct a DFA accepting the regular language specified by the expression R. When $cmd(v) = s_l + s_r$ where $s_l, s_r \in S$ denote two predecessors of the vertex v, we construct a DFA accepting the union of strings from the two predecessors. Let M_l and M_r be the DFAs of the predecessors s_l and s_r respectively. Then, the DFA M of v accepts the union of the languages of M_l and M_r. That is, $L(M) = L(M_l) \cup L(M_r)$. When $cmd(v) = s_l s_r$, we construct a DFA accepting strings from the predecessor s_l followed by those from the predecessor s_r. The DFA M of v subsequently accepts the concatenation of the languages of M_l and M_r. I.e., M has the property that $L(M) = \{uw : u \in L(M_l), w \in L(M_r)\}$. Finally, when $cmd(v) = s_o[s_f \mapsto s_t]$, we construct a DFA accepting any pattern from s_o whose substrings from s_f are replaced by strings from s_t. Let M_o, M_f, M_t be deterministic finite automata of the predecessors s_o, s_f, and s_t respectively. The DFA M of v accepts the following language $\{w : k > 0, w_1 x_1 w_2 x_2 \cdots w_k x_k w_{k+1} \in L(M_o), w = w_1 y_1 w_2 y_2 \cdots w_k y_k w_{k+1}, x_i \in L(M_f), y_i \in L(M_t)$ for all $1 \leq i \leq k$, and $w_j \notin \{ux'v : x' \in L(M_f), u, v \in \Sigma^*\}$ for all $1 \leq j \leq k+1\}$.

If the dependency graph contains cycles (i.e., if the string manipulating code in the web application contains loops or recursion), then just propagating the values along the edges of the dependency graph is not sufficient. In that case, it

is easy to convert the above computation to a least fixpoint computation by initially annotating each intermediate vertex of the dependency graph with a DFA that does not accept any string value, and iteratively updating the DFAs according to the above rules. However, since the lattice defined by sets of strings contains infinite chains, this fixpoint computation is not guaranteed to converge. In order to guarantee convergence we use an automata widening operation for the vertices in the dependency graph that are part of a cycle [4, 37]. Widening operation guarantees that the fixpoint computation converges in the presence of cycles, and the result is an over-approximation of the least fixpoint.

After the DFA M_s for the sink $s \in V$ is obtained, M_s accepts (an over approximation) of all string values that can reach the sink assuming that input vertices can take arbitrary string values. $L(M_s) \cap L_{attack}$ is the language of all malicious attacks that can reach the sink node. Let S_s accept this language. We would like to identify safe input strings which do not yield any attack at the sink. To this end, we construct a DFA S_{input} via backward reachability analysis along the reverse path from the sink to the input node. Each step of the backward analysis corresponds to a pre-image computation for a string operation. When $cmd(v) = s_l s_r$, we would like to construct a DFA accepting strings for the predecessor s_l or the predecessor s_r. For instance, given the DFA M of v and M_r, we compute S_l so that M accepts the concatenation of the languages of S_l and M_r. Precisely, S_l has the property that $L(S_l) = \{u : uw \in L(M), w \in L(M_r)\}$. We can compute S_r in a similar way. When $cmd(v) = s_o[s_f \mapsto s_t]$, we construct a DFA S_o for the predecessor s_o so that given M accepts any string from $L(S_o)$ whose substrings from s_f are replaced by strings from s_t. Let M, M_f, M_t be DFAs for v, the predecessors s_f, and s_t respectively. The DFA S_o of the predecessor s_o can be computed as an over-approximation by constructing the DFA that accepts the following language $\{w : k > 0, w_1x_1w_2x_2 \cdots w_kx_kw_{k+1} \in L(M), w = w_1y_1w_2y_2 \cdots w_ky_kw_{k+1}, x_i \in L(M_t), y_i \in L(M_f) \cup L(M_t)$ for all $1 \leq i \leq k$, and $w_j \notin \{ux'v : x' \in L(M_t), u, v \in \Sigma^*\}$ for all $1 \leq j \leq k+1\}$.

As with the forward analysis, during backward analysis we use the automata widening operation to guarantee convergence in the presence of cycles in the dependency graph.

When the backward analysis reaches the input node, the computed DFA for the input node, denoted as S_{input} accepts an over approximation of all malicious input strings. These are all input strings that can cause an exploit for the given attack pattern. The sanitization signature that characterizes safe input strings (which obey the input format, and are guaranteed to not yield any attack at the sink with respect to the given attack pattern) is defined as the DFA that accepts the intersection of the language L_{input} (the strings that obey the input format) and the language $\Sigma^* \setminus L(S_{input})$ (the strings that are guaranteed to not cause an attack). We compute the sanitization signature automaton that accepts the language $L_{input} \cap (\Sigma^* \setminus L(S_{input}))$ using automata complement and automata product operations.

3. OPTIMAL SANITIZATION SYNTHESIS

In this section, we present algorithms that find optimal corrections for malicious (or not in the format) inputs. After the correction, the modified input will be in the sanitization signature. To be more specific, given a DFA A that represents the sanitization signature (computed as described in the previous section) and a string w that represents the input, the algorithms presented in this section find a string w' in $L(A)$ with the minimal edit-distance to w. Formally, the goal is to find $w' \in L(A)$ such that $dist(w, w') \leq dist(w, w'')$ for all $w'' \in L(A)$ where $dist(w, w')$ denotes the edit distance between two strings which we define below.

A *symbol operation* on a string is deleting a symbol, inserting a symbol, or substituting a symbol in the string. Let $u, w \in \Sigma^*$. One can transform u to w by a sequence of symbol operations. The *distance* between u and w (written $dist(u, w)$) is the least number of symbol operations transforming u to w. Clearly, $dist(u, u) = 0$ and $dist(u, w) \geq 0$. Moreover, one can show that $dist(u, w) = dist(w, u)$ and $dist(u, v) + dist(v, w) \geq dist(u, w)$ for every $u, v, w \in \Sigma^*$.

Below we present our sanitization synthesis approach in three parts. The first part presents the basic sanitization algorithm that converts an input string to another string that is in the sanitization signature with minimal edits. The next part describes an improved version of the sanitization algorithm that computes the result in an incremental manner. Responsiveness of web-applications is an important concern, so our goal is to keep the runtime overhead of sanitization as small as possible. In the last part, we describe a new algorithm that pre-computes information needed for the incremental algorithm. This revised algorithm further reduces the runtime computation time needed for each input.

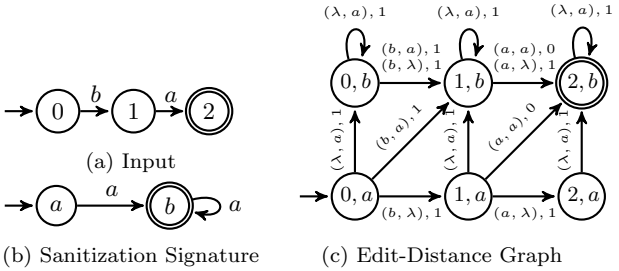
3.1 Optimal Sanitization

We begin with the example in Figure 2 where the input string ba is encoded in the DFA in (a) and the sanitization signature a^+ is encoded by the DFA in (b). We assume in the example the set of alphabet symbols is $\Sigma = \{a, b\}$. In order to find the optimal sanitization, we build the labeled graph in (c) from the two automata. We call such a graph an *edit-distance graph*.

The nodes in the graph come from the states of the two automata. For example, the initial node $(0, a)$ comes from the two initial states 0 and a; the final node $(2, b)$ comes from the two final states 2 and b. There are three outgoing edges from the node $(0, a)$. Each of them models a different edit operation. The edge that goes to the node $(0, b)$ is labeled by $(\lambda, a), 1$, which means that it consumes nothing from the input (the λ symbol) and turns it to a. Hence this models an *insert* operation. Moreover, the cost of using this edge is one. In such a case, the state of the input DFA stays the same and the sanitization signature DFA moves to the state b. The edge that goes to the node $(1, a)$ models a *delete* operation. The input DFA moves to the state 1 while the state of the sanitization signature DFA stays the same. The edge to $(1, b)$ models the *substitute* operation.

From the node $(1, b)$ to the node $(2, b)$, there is an edge labeled $(a, a), 0$, which means that the symbol of the input matches the sanitization signature and hence no modification is needed. In such a case, the cost is zero. For finding an optimal patch, we need to find a path in the edit-distance graph from the initial node to the final node, with the lowest cost. This can be done by applying the standard single-source shortest path algorithm. In this example, we obtain the following shortest path $(0, a) \xrightarrow{(b,a),1} (1, b) \xrightarrow{(a,a),0} (2, b)$. From the edge labels of the path, we find that the input ba will be modified to aa and the total cost is one, which equals $dist(ba, aa)$. We will formalize this sanitization pro-

cedure below.



(a) Input

(b) Sanitization Signature

(c) Edit-Distance Graph

Figure 2: Computing Optimal Sanitization

Let $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a DFA of the sanitization signature produced by the procedure in Section 2. We create a DFA $X = \langle P, \Sigma, \delta_P, p_0, F_P\rangle$ for the input string w such that $L(X) = \{w\}$.

We create the edit-distance graph $\langle V, E\rangle$ for the DFA A and the DFA X as follows. The set of nodes $V = P \times Q$. A labeled edge $(p, q) \xrightarrow{(a_p, a_q), c} (p', q') \in E$ if and only if one of the following conditions holds:

- **Insert:** The state of the input DFA remains unchanged and the DFA of the sanitization signature A fires a transition. The cost of this operation is one. Formally, $p = p'$, $a_p = \lambda$, $\delta(q, a_q) = q'$, and $c = 1$.
- **Delete:** The input DFA X fires a transition. The state of the sanitization signature DFA A remains unchanged. The cost of this operation is one. Formally, $\delta_P(p, a_p) = p'$, $q = q'$, $a_q = \lambda$, and $c = 1$.
- **Substitute:** Both the input DFA X and the DFA A fire transitions. The symbol of the transition fired by A does not match the symbol of the transition fired by X. The cost of this operation is one. Formally, $\delta(q, a_q) = q'$, $\delta_P(p, a_p) = p'$, $a_q \neq a_p$, and $c = 1$.
- **Match:** Both the input DFA X and the DFA A fire transitions. The symbol of transition fired by A matches the symbol of the transition fired by X. The cost of this operation is zero, because no modification to the input string has been made. Formally, $\delta(q, a_q) = q'$, $\delta_P(p, a_p) = p'$, $a_q = a_p$, and $c = 0$.

After constructing the edit-distance graph using the above construction, we compute the path with minimal cost from the initial node (p_0, q_0) to some final node (p_f, q_f) with $p_f \in F_P$ and $q_f \in F$. Such minimal cost path can be found by first applying Dijkstra's shortest path algorithm to find a minimal cost path represented by a sequence of symbols $((a_0, a_0{}'), c_0), ((a_1, a_1{}'), c_1), \ldots, ((a_m, a_m{}'), c_m)$. We obtain the sanitized input by removing all padding symbols λ from the string $a_0{}'a_1{}' \ldots a_m{}'$ and the cost of sanitization is given as $c_0 + c_1 + \ldots + c_m$.

THEOREM 1 (CORRECTNESS). *Let A be the DFA of the sanitization signature, w be the input string, and w' be the sanitized string produced by the above procedure. Then $dist(w, w') \leq dist(w, w'')$ for all strings $w'' \in L(A)$.*

The edit-distance graph in the worst case has $|P| \times |Q|$ nodes. The most time consuming step of our procedure is the computation of the shortest path. Because the worst case time complexity for the Dijkstra's shortest path algorithm is bounded by the square of the number of nodes. In the worst case, the time complexity of this sanitization procedure is $(|P| \times |Q|)^2$.

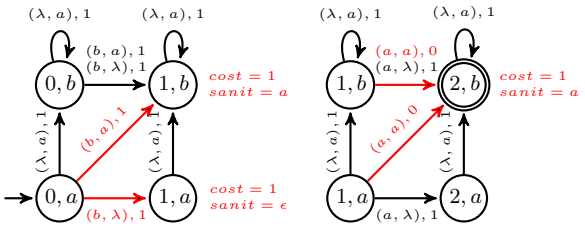
3.2 Incremental Sanitization Computation

The optimal sanitization algorithm described in the previous section builds an edit-distance graph of $|P| \times |Q|$ nodes. Here, we introduce an optimization. The new algorithm makes use of the fact that the DFA of the input is of the shape of a straight line, that is, it does not have loops, it is connected, and each state has at most one outgoing transition. For an input string $a_1 a_2 \ldots a_m$, if we already computed the optimal sanitization for the prefix $a_1 a_2 \ldots a_i$, then we only need to remember the sanitized string and the cost of the sanitization so far. Most of the other information computed can be discarded, because the algorithm will never read from the prefix again. We explain the main idea with the example in Figure 3, where we assume the same input string ba and the same sanitization signature a^+ as in Figure 2. In the incremental construction, we process a DFA transition at a time and create a corresponding edit-distance graph.

We begin with the transition $0 \xrightarrow{b} 1$ and apply the construction of edit-distance graph described in the previous section to get the edit-distance graph in Figure 3(a). We apply Dijkstra's single source shortest path algorithm from the initial node $(0, a)$ to all nodes associated with the DFA state 1, which is the state after reading b. In this case, we only have two such nodes $(1, a)$ and $(1, b)$. We then obtain a minimal cost path to the node $(1, a)$ with an empty string ϵ as the optimal sanitization and a minimal cost path to the node $(1, b)$ with an optimal sanitization a. We call a pair of a sanitized string and the cost a *sanitization record*. We maintain a mapping Record from a graph node to a sanitization record in the algorithm. Here we have $\mathsf{Record}(1, a) = ((\epsilon), 1)$ and $\mathsf{Record}(1, b) = ((a), 1)$ We then drop the edit-distance graph of the symbol b.

In the next iteration, we construct the edit-distance graph of the transition $1 \xrightarrow{a} 2$ (Figure 3(b)). Here we start the single source shortest path algorithm from all *initial nodes* (those associated with the state 1, namely the nodes $(1, a)$ and $(1, b)$). The initial condition of each node is recoded in the mapping Record computed from the previous iteration. The algorithm then computes a minimal cost path from $(1, a)$ and from $(1, b)$ to the final node $(2, b)$. In this example, from both initial nodes, the algorithm finds paths to the final node with the cost 1 (the paths are $(1, b) \xrightarrow{(a,a),0}, (2, b)$ and $(1, a) \xrightarrow{(a,a),0}, (2, b)$, respectively). Because the two sanitizations have the same cost, the algorithm can choose any of them as the final result. In Figure 3, the algorithm chooses the latter.

Here, we formally describe the incremental algorithm. Let $A = \langle Q, \Sigma, \delta, q_0, F\rangle$ be the DFA for the sanitization signature. For the input string w, we first build a DFA X such that $L(X) = \{w\}$. For convenience, we use 0 for the initial state, j for the state after reading the j-th symbol in w. For each length-one substring $w[c]$ of w, $c \in [1, |w|]$, we create an edit-distance graph G_c using the following procedure. Assume that now our input string is $w[c]$. We create a DFA $X_c = \langle \{c-1, c\}, \Sigma, \delta, c-1, \{c\}\rangle$ such that



(a) Edit-Distance Graph for the Symbol "b"

(b) Edit-Distance Graph for the Symbol "a"

Figure 3: Computing Optimal Sanitization Incrementally

$L(X_c) = \{w[c]\}$. Notice that X_c has only two states $c-1$ and c. The state $c-1$ is the only initial state and c is the only final state. We construct $G_c = \langle V, E\rangle$ using the procedure in Section 3.1. It follows that $V = \{c-1, c\} \times Q$, the set of initial nodes is $\{(c-1, q) \mid q \in Q\}$, and the set of final nodes is $\{(c, q) \mid q \in Q\}$. Here we have two exceptions. For the graph G_1 the initial node is $(0, q_0)$ and for the last graph $G_{|w|}$ the final nodes are $\{(c, q) \mid q \in F\}$. When computing the shortest path to each final node, the algorithm obtains from the map Record the sanitization record of each initial node, if there is one. Similarly, after the shortest path to all final nodes from each initial node are computed, the algorithm updates the sanitization record.

To be more specific, the algorithm works in an iterative manner. At the i-th iteration, the algorithm creates the edit-distance graph G_i, computes the shortest paths from initial nodes to all final nodes, updates the map Record, and then drops the graph G_i for saving space. When the graph $G_{|w|}$ is computed, we can obtain the optimal sanitized strings from the set of sanitization records $\{\mathsf{Record}(|w|, f) \mid f \in F\}$.

THEOREM 2 (CORRECTNESS). *Let A be the DFA of the sanitization signature, w be the input string, and w' be the sanitized strings generated by the incremental algorithm. Then $dist(w, w') \leq dist(w, w'')$ for all strings $w'' \in L(A)$.*

Each edit-distance graph in the worst case has $2|Q|$ nodes. For the computation of shortest path in each edit-distance graph, one needs to try at most $|Q|$ source nodes, one for each state of the sanitization signature DFA. The worst case time complexity for the Dijkstra's shortest path algorithm is bounded by the square of the number of nodes $(2|Q|)^2$. The total number of edit-distance graphs to be computed is $|P|$. In the worst case, the time complexity needed for the analysis of each edit-distance graph is $|P| \times 2|Q| \times (2 \times |Q|)^2 = |P| \times |Q|^3 \times 8$. Comparing with the basic algorithm, the incremental algorithm uses less space and is expected to be faster for longer input strings.

3.3 Pre-computation of Edit-Distance Graphs

It is crucial to keep the runtime cost of sanitization low since it will directly affect the responsiveness of the given web application. It can be worthwhile to sacrifice some space efficiency in order to reduce the runtime cost of sanitization. The most time-consuming step in the sanitization approach we described in the previous section is the computation of sanitization records. Here we present an approach that pre-computes all possible edit-distance graphs and the sanitization records offline. So we do not need to do the heavy computation for each new input. Another advantage of the approach based on pre-computation is that there can be a lot of redundancy when there are multiple occurrences of the same symbol in the input string. For example, for an input string *bbabb*, the incremental algorithm will recompute the edit-distance graph of the string b several times which is potentially very wasteful.

Indeed, since the alphabet symbols are finite, it is possible to pre-compute the edit-distance graph for each alphabet symbol. The only thing we need to deal with is that, in this approach the initial sanitization records are unknown.

The pre-computation algorithm consists of two-stages: (1) pre-compute edit-distance graphs and the sanitization records offline and (2) use the pre-computed sanitization records to compute the optimal sanitization at runtime. For the first stage, the sanitization record of each initial node will affect the final decision of shortest paths. Consider the example in Figure 3(b). If the initial cost of the node $(1, b)$ is larger than the initial cost of the node $(1, a)$, then the minimal cost path has to be $(1, a) \xrightarrow{(a,a),0} (2, b)$. Otherwise, if the initial cost of the node $(1, a)$ is larger than the initial cost of the node $(1, b)$, then the minimal cost path has to be $(1, b) \xrightarrow{(a,a),0} (2, b)$.

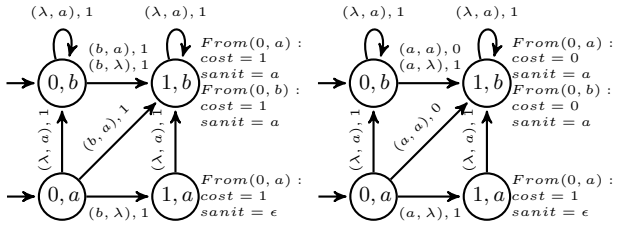
The sanitization records are unknown in the pre-computation stage. Therefore, in each final node, we have to remember a sanitization record for each initial node. This can be done very easily. For each edit-distance graph and for each initial node, we apply the single source shortest path algorithm and compute the sanitization record of each final node.

Example pre-computed edit-distance graphs for the DFA in Figure 2(b) can be found in Figure 4. We put the sanitization record on the right side of each final node. Notice that we got only the sanitization record from the node $(0, a)$ in the node $(1, a)$, because $(1, a)$ is unreachable from the node $(0, b)$.

In the second stage, we make use of the pre-computed edit-distance graphs and sanitization records to speed up the sanitization finding procedure. The algorithm for this stage is almost identical to the incremental algorithm, except that we replace the procedure of building new edit-distance graphs with a new procedure that checks the pre-computed edit-distance graph. To be more specific, instead of building the edit-distance graph G_c, the algorithm checks the sanitization records of the pre-computed graph of the symbol $w[c]$. For each final node v, the algorithm obtains the sanitization records of the corresponding initial nodes from Record and computes an optimal sanitization record r. It then updates the map $\mathsf{Record}(v) = r$.

THEOREM 3 (CORRECTNESS). *Let A be the DFA of the sanitization signature, w be the input string, and w' be the sanitized string generated by the procedure described in this subsection. Then $dist(w, w') \leq dist(w, w'')$ for all strings $w'' \in L(A)$.*

We now analyze the complexity of the two stages of the algorithm. For the first stage, each pre-computed edit-distance graph in the worst case has $2|Q|$ nodes. For the computation of shortest path in each edit-distance graph, one needs to try at most $|Q|$ source nodes, one for each state of the sanitization signature, for getting the sanitization records. The worst case time complexity for the Dijkstra's shortest path algorithm is bounded by the square of the number of



(a) Pre-computed Edit-Distance Graph of the Symbol "b"

(b) Pre-computed Edit-Distance Graph of the Symbol "a"

Figure 4: Pre-computed Edit-Distance Graphs

nodes $(2|Q|)^2$. The total number of edit-distance graph to be computed is $|\Sigma|$. In the worst case, the time complexity needed for the analysis of each edit-distance graph is $|\Sigma| \times |Q| \times (2|Q|)^2 = |\Sigma| \times |Q|^3 \times 4$.

For the second stage, the total number of iterations is $|P|$. For each iteration, the algorithm gets the pre-computed sanitization records and reads from Record at most k times, where k equals to the maximum number of initial nodes $|Q|$. Therefore, the worst case complexity of the second stage is $|P| \times |Q|$. Note that, only the time complexity of second stage affects the runtime cost since the first stage is computed offline based on the sanitization signature automaton.

4. EXPERIMENTS

In this section we discuss the experiments we conducted to evaluate our optimal sanitization synthesis approach.

4.1 Signature Genration and Patch Synthesis

We have integrated the optimal patch generation technique we present in this paper to the online service Patcher [24]. Patcher allows users to upload their PHP web applications for detecting, viewing and repairing vulnerabilities. Patcher consists of several components. Pixy [15] is a vulnerability analysis tool for PHP, and it 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. Given a dependency graph that characterizes string manipulation operations, Stranger [35] implements forward and backward symbolic reachability analysis we described earlier. Stranger uses the automata package of MONA tool [5] to store the automata constructed during string analysis symbolically.

We use the sanitization signature generation technique we discussed earlier to generate the sanitization signature using the forward and backward symbolic reachability analyses implemented in Stranger [35] . Based on the sanitization synthesis techniques we described in this paper we generate sanitization statements and insert them at the program points where user inputs are read by the application. These sanitization statements then modify malicious inputs to benign ones at run time.

We experimented on 10 open source PHP applications that include 2961 files with totally 395132 lines of code. Patcher adopts a distributed framework and employs a 32-bit Java Virtual Machine (JVM) with 2 GB of memory as an individual Worker to run each taint/string analysis task. Table 1 summarizes the analysis result. In these PHP source codes, we discover 482 of SQL injection tainted sinks, 3536 of XSS tainted sinks and 1477 of MFE tainted sinks. We conduct forward analysis on 482 SQL tainted sinks against four kinds of SQL injection attack patterns, on 3536 XSS tainted sinks against nine kinds of XSS attack patterns and on 1477 MFE (Malicious File Execution) sinks against one attack pattern. With respect to each attack pattern, we found that there are totally 1719 SQL injection vulnerabilities, 14747 cross site scripting vulnerabilities and 562 malicious file execution vulnerabilities. Note that a tainted sink may correspond to multiple vulnerabilities against different attack patterns. For each vulnerability, Patcher conducted sanitization analysis to characterize malicious inputs. It has generated 1595 (out of 482×4) vulnerability signatures for SQL injections, 13477 (out of 3536×9) for XSS attacks and 562 (out of 1477 × 1) for MFE attacks. Patcher takes the compliment of these vulnerability signatures to synthesize sanitization statements.

Note that the reachability analysis on the dependency graphs loses precision during the fixpoint computation (using widening), in replace operation variations (such as first-match or longest-match), in un-modeled functions (using arbitrary words), and when relational constraints influence control flow. We did not observe false positives due to widening or replacement approximations in experiments. However, we did observe false positives due to un-modeled built-in functions and over approximation of relations among input variables. This could be improved by more sophisticated modeling and relational string analysis. In the next subsection, we evaluate the runtime performance of the synthesized sanitizers.

4.2 Runtime Performance Evaluation

To be useful in practice, runtime cost of the sanitizers we generate must be low. As we discussed in Section 3 runtime cost depends on the length ($|P|$) of the input string and the number of states ($|Q|$) of the sanitization signature automaton.

To evaluate the runtime performance of synthesized sanitizers, we randomly generate sample inputs and check the average time of finding the optimal sanitization of these inputs against three sanitization signatures (denoted as $|Q|$=12, $|Q|$=62, and $|Q|$=74). These signatures are manually selected from the previous analysis to represent typical signatures with different number of states. For each string length, we randomly generate 1000 sample strings (restricted to characters that appear in the attack pattern). Figures 5 and 6 highlight the runtime cost of the `Incremental` algorithm and the `Pre-computation` algorithm, respectively, for three different sanitization signatures, demonstrating the linear increase in time as the string length increases. This confirms linear time complexity in $|P|$ of both algorithms. The performance improvement between the `Incremental` and `Pre-computation` algorithms is quite significant. Consider an input string with 40 characters and the sanitization signature with $|Q|$ =74. The `Incremental` algorithm takes around one sec (Figure 5) to find an optimal sanitization, while using the `Pre-computation` algorithm it takes only 4 milli-seconds (Figure 6), achieving a 250 times speed-up. Furthermore, it takes nearly 60 seconds using the `Basic` algorithm on the same example. We achieve a nearly 15000 times speed-up using the `Pre-computation` algorithm rather than searching the optimal edition in the whole automaton

Table 1: Analysis Summary

Application	# of Files	# of Lines	# of Tainted Sinks				# of Vulnerabilities				Analysis Time
			SQLI	XSS	MFE	Total	SQLI	XSS	MFE	Total	milli-sec
benchmarks	10	397	0	8	0	8	0	56	0	56	126816
e107	218	11303	0	0	0	0	0	0	0	0	64746
examples	5	47	0	0	0	0	0	0	0	0	1249
market	22	2566	43	23	12	78	12	34	8	54	114189
moodle1_6	1319	273468	0	1888	773	2661	0	2280	103	2383	15313148
nucleus3.64	67	23522	0	7	16	23	0	54	2	56	101439
PBLGuestbook	3	1566	8	1	2	11	28	0	2	30	207525
php-fusion-6-01-18	1156	58542	129	1399	597	2125	371	9393	423	10187	37699265
schoolmate	63	8287	291	149	0	440	1140	1149	0	2289	11576501
sendcard_3-4-1	72	11215	1	60	40	101	4	502	17	523	817539
servoo	26	4219	10	1	37	48	40	9	7	56	153862
Total	2961	395132	482	3536	1477	5495	1595	13477	562	15634	66176279

with direct composition. This improvement facilitates runtime patching in practice for real applications.

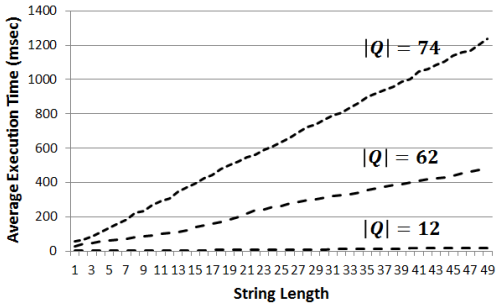


Figure 5: Performance of the `Incremental` algorithm

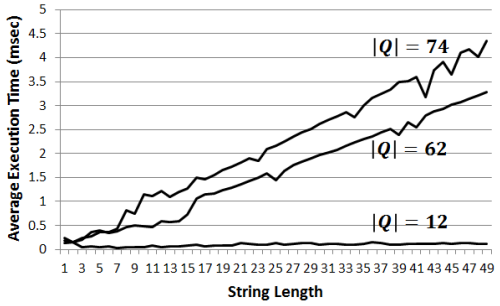


Figure 6: Performance of the `Pre-computation` algorithm

4.3 Sanitization Evaluation

We evaluate the sanitizer quality in this subsection by investigating sanitized inputs. Table 2 shows a list of input patterns that are used to specify desired formats of user inputs, as well as a list of attack patterns that are used to specify attack strings that may raise a vulnerability when it is taken by a sensitive function.

Given an attack pattern A, let $S(A)$ denote the result DFA of backward analysis for a vulnerable sink with respect to the attack pattern A. Let $\overline{M}$ denote the DFA that accepts the complement set of $L(M)$. Let I be a given input pattern. The sanitization signature used by our optimal patch sanitizer characterizes safe (and in the desired format) inputs. The signature is composed by taking the automata intersection of I and $\overline{S(A)}$. On the other hand, to compare the min cut approach presented in [36], we also generate the vulnerability signature that characterizes all malicious (or not in the format) inputs. The signature is composed by taking the automata union of $\overline{I}$ and $S(A)$.

For the min-cut sanitizer, we first compute *the minimal cut* to identify the set of characters that have to be deleted. The min cut approach ensures that after removing all the characters in the cut to reform the input, the input is not accepted by the vulnerability signature. We use a very simple vulnerable PHP script in which a user input is directly taken by a sensitive function to ease the comparison on the original input to the sanitized one. In general, the backward analysis result can be far different from the attack pattern, depending on string operations involved in the script.

Table 3 shows several examples on how an input string is modified by our optimal patch sanitizer and the min-cut sanitizer. Lets first consider cases with only attack patterns ($\#A$). Both optimal patch sanitizer and min-cut sanitizer can modify an attack string into a safe one. However, our optimal patch sanitizer achieves minimal edition for all cases; while min-cut sanitizer on the other hand may delete extra characters that are harmless and do not contribute to the attack string in the sink with respect to the attack pattern.

The advantage of minimal edition turns out to be significant when we consider input patterns. For all cases where we use input patterns ($\#I$) to specify the desired formats on inputs, our optimal patch sanitizer can modify typo inputs to the desired ones with minimal edition; the min-cut sanitizer deletes all the characters and returns an empty string ϵ. This shows the restriction of a cut-based approach: the modification based on only deletion has very limited ability to transform a string into a specific format.

Finally, we discuss the cases with the composition of input patterns and attack patterns ($\#I$, $\#A$). In this case, our optimal patch sanitizer can still find a desired format that is safe with respect to the attack pattern. Note that input patterns themselves are not sufficient to be used to guarantee safe inputs. For example, the input pattern I_8 specifies an email address format where it allows arbitrary characters

Table 2: Examples of attack patterns and input patterns

#	Attack Pattern	Regular Expression
A_1	Script Tag	/.*\<SCRIPT.*\>.*/
A_2	Escaping JavaScript	/.*\\";.*\/\/.*/
A_3	SQL ASCII	/.*((\%27) |(\')) \s*((\%6F) |o|(\%4F)) \s*((\%72) |r |(\%52)) \s*.*/
A_4	SQL UNION	/.*((\%27)|(\'))union.*/
A_5	VBScript	/.*vbscript:.*/
A_6	Exec Directive	/.*exec(\s|\+)+(s|x)p.*/
#	Input Pattern	Regular Expression
I_1	Time String	/([1-9]|1[012]):(0[0-9]|[1-5][0-9]) ?(am|AM|pm|PM)/
I_2	Date String	/(0?[1-9]|1[012])(/|-|\.)(0?[1-9]|[12][0-9]|3[01])(/|-|\.)(19|20)[0-9][0-9]/
I_3	US Telephone Number	/\([0-9][0-9][0-9]\)[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]/
I_4	Currency String	/\$?[1-9][0-9]{0,2}(,[0-9][0-9][0-9])*(\.[0-9]([0-9]{0,2}))?/
I_5	URL "Slug"	/([a-z]|[0-9]|-)([a-z]|[0-9]|-)([a-z]|[0-9]|-)([a-z]|[0-9]|-)+/
I_6	Hex Number	/(#|0x)[a-f0-9]+/
I_7	User ID String	/([a-zA-Z]|[0-9]|-/)+/
I_8	Email Address	/[^@]+\@([0-9]|[a-z]|-)+\.[a-z]+/
I_9	URL	/(https:\/\/)(-|[0-9]|[a-z])+\.([a-z])(\/|-|[a-z])*(\/)/
I_{10}	IP Address	/(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?) \.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[0-1]?[0-9][0-9]?)/

(except @) used as the user name. As shown in Table 3 , for the bottom cases of I_8, an input that may result in an XSS attack (or a SQL injection) can pass the sanitizer without any modification. Such a malicious input is modified to a benign one when we compose our sanitization signature with attack patterns, e.g., (I_8, A_1) and (I_8, A_3).

5. 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 [6] which has also been used to check for various types of errors in Web applications [10, 21, 32, 33]. In [21, 32], multi-track DFAs, also known as *transducers*, are used to model replacement operations. Several string analysis tools that use symbolic string analysis based on DFA encodings have been proposed [28, 9, 13, 37]. BEK [13] adopts symbolic finite automata and transducers to conduct string analysis and sanitizer analysis. Veanes et al. [30] adopt parallel computation on symbolic string analysis improving the scalability of automata-based string analysis of string manipulating programs.

String constraint solvers [1, 19, 40, 18, 17, 26] provide decision procedures for word equations with length constraints that can be generated via applying symbolic execution to string manipulating programs [26]. HAMPI [17] and Kaluza [26] are bounded string constraint solvers that search for a string that satisfies a given set of string constraints by bounding the string length. PASS [18] adopts parameterized arrays instead of bit-vectors [17, 26]. This type of bounded analysis cannot be used for sound string analysis whereas the string analysis techniques we present in this paper are sound. CVC4 [19], Z3-Str [40] and NORN [1] are SMT-based solvers for reasoning satisfiability of string constraints over unbounded strings and integers. Among them the solver Norn also has the capability of computing *Craig interpolant*, which enables a CEGAR-based software model checking procedure [11] for the analysis of string manipulating programs. Aydin, Bang and Bultan [3] integrate automata-based string analysis [37] with model counting techniques to string constraint solving. These solvers can be used to detect vulnerabilities in string manipulating programs while generating a witness for vulnerable programs, but not how a witness can be fixed or how a vulnerability can be fixed as we did in this work.

There has been previous work on securing applications automatically, e.g., by separation of data and code [29, 25, 8, 12] and by placing sound sanitizers [27, 20]. Su and Wasserman [29] present SQLCHKS for SQL injection attacks, generating filters that block user queries in which the input substrings change the syntactic structure of the rest of the query. Samuel, Saxena, and Song [25] secure web application vulnerabilities by enforcing predefined web language frameworks. deDacota [8] realizes separation of code and data in web pages. WEBLOG [12] also ensures that user inputs are never treated as SQL keywords. ScriptGard [27] can detect and repair incorrect placement of sanitizers. Livshits and Chong [20] propose automatic sanitizer placement by analyzing the flow of tainted data in the program. Compared to these work, we address how to to eliminate string vulnerabilities by synthesizing effective sanitization statements for user inputs that composes desired input patterns and safe inputs with respect to attack patterns. As for the implications of patching a web application with multiple sanitizers synthesized from multiple attack patterns and input patterns, it is possible to synthesize one sanitizer that does all sanitizes' work by taking the signature as the intersection of all the sanitization signatures.

DFA based symbolic reachability analysis has been used to verify the correctness of string sanitization operations in PHP programs before [38, 37]. In [36] a vulnerability signature (that characterizes all bad inputs) and a patch generation technique based on vulnerability signatures is presented. The generated patches are based on a min-cut algorithm that sanitizes the input by deleting a fixed set of characters. The min-cut approach finds a cut such that deleting the set of characters in the cut from any input transforms the input to a safe one (where empty string is always considered safe). Our experiments show that when min-cut approach is used together with the input patterns, the cut includes all characters in many cases. This makes the min-cut approach delete all the input string, resulting in an empty string.

Sanitization is commonly used in web applications to modify the user input to obtain a safe and desirable string, which indicates that developers are not content with using a pure validation approach that just rejects the offending input. Most existing approaches rely on existing threat models and

Table 3: Optimal Patch Sanitizer v.s. Min-Cut Sanitizer [36]

#	Input String	Optimal Patch (`Pre-computation`)	Min-Cut Patch
A_1	<IMG """"><SCRIP T>alert("XSS")</SCRIPT>">	<IMG """"><SSCRIP T>alert("XSS")</SCRIPT>">	IMG """">SCRIP T>alert("XSS")/SCRIPT>">
A_1	<SCR<SCRIPT IPT test	<SCR<SCRIPT!IPT test	SCRSCRIPT IPT test
A_2	<IMG \"alert('XSS');//	<IMG \"yalert('XSS');//	IMG \"alert('XSS');//
A_3	' or "=" –	'! or "= " –	or = –
A_4	'union ALL SELECT password FROM users WHERE username = 'admin'/*	'uunion ALL SELECT password FROM users WHERE username = 'admin'/*	'nion ALL SELECT password FROM sers WHERE sername = 'admin'/
A_5	<IMG SRC='vb script : msgbox("XSS")'	<IMG SRC='_b script:msgbox("XSS")'	<IMG SRC='b script : msgbox("XSS")'
A_6	exec+xp_cmdshell 'cmd.exe dir c:'	exxec+xp_cmdshell 'cmd.exe dir c:'	exe+xp_cmdshell 'cmd.exe dir :'
I_1	2-31 am	2:31 am	ϵ
I_2	102-302-2031	10-30-2031	ϵ
I_3	000003-0010	(000)003-0010	ϵ
I_4	12345	$1.345	ϵ
I_5	add_calendar_eventS	add-calendarevent	ϵ
I_6	0x1ggg	0x1	ϵ
I_7	Jimmy.Lin	Jimmy-Lin	ϵ
I_8	example#gmail.com	example@gmail.com	ϵ
I_9	ftp://example.com/#test/	https://example.com/test/	ϵ
I_{10}	192,4444.19.3	192.44.19.3	ϵ
I_8	<SCRIPT>alert('2')</SCRIPT>@nccu.edu	<SCRIPT>alert('2')</SCRIPT>@nccu.edu	ϵ
I_8, A_1		<SCRIP>alert('2')</SCRIPT>@nccu.edu	ϵ
I_8	' or "=" – @nccu.edu	' or "=" –@nccu.edu	ϵ
I_8, A_3		' ! or "=" –@nccu.edu	ϵ

can become ineffective when new attacks emerge. Our approach is fully parameterized with respect to attack and input patterns and can be applied to future vulnerabilities when they are characterized via attack patterns. For instance, there are several web development frameworks that can be used to prevent XSS when rendering the HTML content. XHP is an augmentation of PHP developed at Facebook to allow XML syntax for the purpose of creating custom and reusable HTML elements for server side rendering. Consider a vulnerable-like XHP script.

```
echo <span>Hi, {$_POST['name']}</span>;
```

The script that echos raw user input "$_POST['name']" is prone to an XSS attack, but by using XHP, the rendering point of "$_POST['name']" (encapsulated in a pair of brace symbols) will be treated as an HTML text (PCDATA). An attacking script in $_POST['name'] will then be escaped by XHP automatically. AngularJS and ReactJS frameworks are similar to XHP but for client side rendering, where, when variables that are encapsulated with double braces, (e.g., "this.prop.name" below), the content of the variable will be escaped automatically at run time.

```
render: function() {
  return (<span>Hi, {this.prop.name}</span>);}
```

However, these techniques are code-insensitive and, therefore, cannot guarantee that only safe values reach the sink. The echo point of the XHP script below could be vulnerable to command injections since the "script" HTML tags are written in the script, where only the execution function name is from the POST.

```
echo <script>{$_POST['foo']}{"()"};</script>;
```

If the POST value is "alert" rather than "foo", the script will execute the alert function in JavaScript. Note that "alert" is still "alert" after escaping provided in modern frameworks and hence it is not protected. To enforce the desired function to be executed, developers can specify the input pattern as the desired function names. Our patch is capable of preventing attacks or removing typos, e.g., by converting "fo0" to "foo", to enforce the correct execution.

Finally, computing the edit-distance between a string and a finite automaton or a string arises in a variety of applications in computational biology, text processing, and speech recognition. Wagner [31] sketched an algorithm which uses finite state automata to calculate the minimal edit distance required for correcting an erroneous word belonging to a regular language. Kashyap and Oommen [16] adopted dynamic programming principles to calculate the edit distance recursively with more memory for keeping results of internal edit distance computation, reducing the complexity of the while calculating process [31]. Allauzen and Mohri [2] propose a linear space algorithm to find out minimal edit distance between two finite automata. They further consider a word lattice as a weighted automaton, and extend the algorithm to the edit-distance between a string and a weighted automaton. We adopt the linear algorithm [2] with pre computation on needed information to improve the performance at runtime. While word corrections have been widely used in many applications such as spelling correction [22, 7], we are not aware of previous work used in synthesizing optimal web security sanitizers. Our pre-computation algorithm itself alone provides an effective way to modify input strings. The techniques could be integrated with modern web security techniques, e.g., in implementing automatic escaping, and could be applied to other contexts as well.

6. CONCLUSIONS

We introduce the concept of code-sensitive and input-sensitive sanitization. By combining automata-based string analysis techniques with minimal edit distance algorithms, we present a novel automated sanitization synthesis technique that generates input-sensitive sanitizers that are capable of composing desired input patterns with safe inputs, guaranteeing that no strings matching attack patterns can reach sinks. We propose minimal edit distance algorithms that improve the performance of automatically generated sanitizers significantly.

7. REFERENCES

[1] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, and J. Stenman. String constraints for verification. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 150–166, 2014.

[2] C. Allauzen and M. Mohri. Linear-space computation of the edit-distance between a string and a finite automaton. In *In London Algorithmics 2008: Theory and Practice*, 2008.

[3] A. Aydin, L. Bang, and T. Bultan. Automata-based model counting for string constraints. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 255–272, 2015.

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

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

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

[7] S. Cucerzan and E. Brill. Spelling correction as an iterative process that exploits the collective knowledge of web users. In D. Lin and D. Wu, editors, *Proceedings of EMNLP 2004*, pages 293–300, Barcelona, Spain, July 2004. Association for Computational Linguistics.

[8] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. dedacota: toward preventing server-side xss via automatic code and data separation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, CCS '13, pages 1205–1216, 2013.

[9] 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.

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

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN*, pages 235–239, 2003.

[12] T. L. Hinrichs, D. Rossetti, G. Petronella, V. N. Venkatakrishnan, A. P. Sistla, and L. D. Zuck. Weblog: A declarative language for secure web development. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 59–70. ACM, 2013.

[13] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium*, 2011.

[14] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, 2004.

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

[16] R. L. Kashyap and B. J. Oommen. An effective algorithm for string correction using generalized edit distance - ii. computational complexity of the algorithm and some applications. *Inf. Sci.*, 23(3):201–217, 1981.

[17] 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.

[18] G. Li and I. Ghosh. PASS: string solving with parameterized array and interval automaton. In *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, pages 15–31, 2013.

[19] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 646–662, 2014.

[20] B. Livshits and S. Chong. Towards fully automatic placement of security sanitizers and declassifiers. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 385–398, 2013.

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

[22] K. Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Comput. Linguist.*, 22(1):73–89, Mar. 1996.

[23] OWASP. Top 10 2013. `https://www.owasp.org/index.php/Top_10_2013-T10`.

[24] Patcher. Patcher online service. `http://soslab.nccu.edu.tw/patcher`.

[25] M. Samuel, P. Saxena, and D. Song. Context-sensitive auto-sanitization in web templating languages using type qualifiers. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 587–600, 2011.

[26] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *S&P*, pages 513–528, 2010.

[27] P. Saxena, D. Molnar, and B. Livshits. Scriptgard: automatic context-sensitive sanitization for large-scale legacy web applications. In *CCS*, pages 601–614, 2011.

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

[29] Z. Su and G. Wassermann. The essence of command injection attacks in web applications. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 372–382, 2006.

[30] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 139–152, 2015.

[31] R. A. Wagner. Order-n correction for regular languages. *Commun. ACM*, 17(5):265–268, May 1974.

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

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

[34] 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.

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

[36] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *ICSE*, pages 251–260, 2011.

[37] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra. Automata-based symbolic string analysis for vulnerability detection. *Formal Methods in System Design*, 44(1):44–70, 2014.

[38] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. *Int. J. Found. Comput. Sci.*, 22(8):1909–1924, 2011.

[39] F. Yu and Y.-Y. Tung. Patcher: An online service for detecting, viewing and patching web application vulnerabilities. In *Proceedings of the 47th Hawaii International Conference on System Sciences*, pages 4878–4886, 2014.

[40] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: a z3-based string solver for web application analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 114–124, 2013.