Semantic Differential Repair for Input Validation and Sanitization

Muath Alkhalaf, Abdulbaki Aydin and Tevfik Bultan
Department of Computer Science
University of California
Santa Barbara, CA, USA
{muath | baki | bultan}@cs.ucsb.edu

ABSTRACT

Correct validation and sanitization of user input is crucial in web applications for avoiding security vulnerabilities and erroneous application behavior. We present an automated differential repair technique for input validation and sanitization functions. Differential repair can be used within an application to repair client and server-side code with respect to each other, or across applications in order to strengthen the validation and sanitization checks. Given a reference and a target function, our differential repair technique strengthens the validation and sanitization operations in the target function based on the reference function. It does this by synthesizing three patches: a *validation,* a *length,* and a *sanitization* patch. Our automated patch synthesis algorithms are based on forward and backward symbolic string analyses that use automata as a symbolic representation. Composition of the three automatically synthesized patches with the original target function results in the repaired function, which provides stronger validation and sanitization than both the target and the reference functions.

1. INTRODUCTION

One of the main forms of interaction between a user and a web application is through text fields. For many text fields (such as username, email, zip code, etc.) a web application typically expects the user input to be in a certain format. Since the user input can contain typing errors, or may be purposefully written (by a malicious user) to violate the expected format, the web application has to validate the user input using input validation operations such as regular expression matching, and sometimes modify the input to put it in the expected format using sanitization operations such as string replacement.

If input validation or sanitization is not used, inputs that violate the expected format can easily cause an application to crash since the user input becomes the input parameter of the action that is executed based on the user request. Moreover, during action execution, user input can be passed as a parameter to security sensitive operations such as sending a query to the back-end database. In order to ensure the security of the application, the user inputs that flow into sensitive functions must be correctly validated and sanitized. Due to global accessibility of web applications, malicious users all around the world can exploit a vulnerable application, so any existing vulnerability in a web application is likely to be exploited by some malicious user somewhere. Given the significance of this security threat, one would expect web application developers to be extremely careful in writing input validation and sanitization functions. Unfortunately, web applications are notorious for security vulnerabilities such as SQL injection and cross-site scripting (XSS) that are due to improper input validation and sanitization.

In this paper, we focus on analyzing and repairing validation and sanitization functions in web applications. In order to repair a function, one first needs specification of the expected behavior. Manual specification of expected input formats for different types of input fields, or manual specification of attack patterns that characterize different types of vulnerabilities require extra effort from the developers and are error prone. In this paper, we present a differential analysis approach that eliminates the need to write manual specifications.

Web application developers often introduce redundant input validation and sanitization code in the client and server-side code of a web application. The checks done on the client-side improve the responsiveness of the application by preventing unnecessary communication with the server and reduce the server load at the same time. However, since a malicious user can by-pass the client-side checks, it is necessary to re-validate and re-sanitize at the server-side. Moreover, many applications repeat the checks for different types of fields in different parts of the application which can be exploited to obtain multiple instances of the validation and sanitization code with the same intended functionality. Finally, across different applications, one can easily find multiple instances of validation and sanitization code used to check standard formats (such as email) or to protect against same class of vulnerabilities (such as SQL injection and XSS). Using the semantic differential repair techniques presented in this paper, we exploit these redundancies within and application and across applications, and automatically repair input validation and sanitization functions by comparing them against each other.

Rest of the paper is organized as follows. After an overview of our differential repair approach (Section 2) we present our contributions which are: 1) the formal modeling of dif-

ISSTA '14 July 21–25, 2014, San Jose, California, USA

```
function reference_function($x){
  if (strlen($x) > 4)
    exit();
  else {
    $x = preg_replace('/</', '', $x);
    if ($x == '')
      exit();
    else
      return $x;
  }
}


function target_function($y){
  $y = preg_replace('/"/', '\"', $y);
  return $y;
}
```

Figure 1: A small, but illustrative example, showing a target function to be repaired based on a reference function.

ferential repair problem for input validation and sanitization functions (Section 3), 2) a novel differential repair algorithm that automatically generates three patches which together constitute a repair (Section 4), 3) specialized automata-based pre and post-image computations for efficient string analysis (Section 5), and 4) the implementation and experimental evaluation of the proposed approach (Section 6). We conclude the paper with a discussion of related work (Section 7) followed by our concluding remarks (Section 8).

2. OVERVIEW

In this section we give an overview of our automated differential repair technique that strengthens the validation and sanitization functionality of a given *target* function based on a given *reference* function.

Consider the example functions shown in Figure 1. The reference function starts with a validation check that blocks any string that is longer than 4 characters. This is followed by a sanitization operation which replaces the character '<' with ϵ (i.e., deletes '<'). Finally, the result of the sanitization operation goes through another validation check that blocks the empty string. The target function in Figure 1 does not do any validation. It only sanitizes the input string by replacing the character '"' with the string "\"" (i.e., it escapes the double quote characters).

The goal of our differential repair technique is to strengthen the validation and sanitization operations in the target function as much as the reference function. More precisely, the goal is to make sure that the repaired target function does not return a string that is not returned by the reference function or the original target function. We call the set of strings returned by a validation and sanitization function its *post-image* (which is the set of strings that reach the sink, i.e., the return statement). Hence, our goal is to make sure that the post-image of the repaired function does not contain any string that is not in the post-image of the reference function and the original target function.

The sink language for the reference function in Figure 1 is the language of all strings that are shorter than 5 characters and not empty and do not contain the character '<', while the post-image for the target function is the language of all strings that do not contain the character '"' unless it is preceded by the character '\'. For example, the string "foo" is an element in the reference function's post-image while the string "foo<" is not since it contains the '<' character. Also, the strings "foo" and "foo\"bar" are elements in the target function's post-image while the string "foo"bar" is not since it contains the character '"' without being preceded by the character '\'.

```
function validation_patch($x){
  if (preg_match('/<*|[^<].{4,}|<[^<].{3,}|<<[^<].{2,}|
                  <<<[^<].+/',$x))
    exit();
  else
    return $x;
}

function length_patch($x){
  if (preg_match(
    '/"".{1,2}|".{1,2}"|.{1,2}""|"[^"]{3,3}|[^"]{3,3}"/',$x))
    exit();
  else
    return $x;
}

function sanitization_patch($x){
  $x = preg_replace('/</', "", $x);
  return $x;
}

function repaired_function($x){
  return target_function(
           sanitization_patch(
             length_patch(
               validation_patch($x)
             )
           )
         );
}
```

Figure 2: The repaired function that is generated by our differential repair algorithm for the target function shown in Figure 1

Our differential repair algorithm works in three phases, where each phase generates a patch-function with a specific purpose: (1) a *validation patch*, (2) a *length patch,* and (3) a *sanitization patch.* The final repair is obtained by applying the composition of all three patch-functions together.

Validation patch.

The purpose of this phase is to generate a patch that makes sure that the repaired function rejects all the inputs that are rejected by the reference function. Figure 3 shows the validation-patch produced in this phase of the repair algorithm for our running example. The validation patch blocks all input strings that are either empty, consist of one or more '<' characters or longer than 4 characters. For example, the strings "", "<", "<<<" and "<html>" will be blocked by the validation patch. On the other hand, the strings "fo" and "<a>" will not be blocked.

The validation patch blocks the inputs that generate a string that is in the post-image of the target function but not in the post-image of the reference function. Note that our algorithm is able detect that some input strings are blocked by the reference function only after being sanitized such as the string "<<<" (which is first converted to empty string by deletion of '<' and then blocked by the reference function). So, for this case, to make sure that the string "<<<" is not in the post-image of the repaired function, the validation patch blocks it.

Length patch.

The purpose of this phase is to make sure that (1) the maximum length of the strings that are in the the post-image of the repaired function is not bigger than the maximum length of the strings that are in the post-image of the reference function and (2) the minimum length of the strings that are in the post-image of the repaired function is not smaller than the minimum length of the strings that are in the post-image of the reference function.

For the reference function in our example, the minimum length is 1, since it blocks the empty string, and the maximum length is 4. On the other hand, for the target function, after the validation patch is applied, the minimum length is 1 since it also blocks the empty string, but the maximum length is not 4 but 8. The reason is that the sanitization in the target function escapes the '`"`' character so that an input string of length 4 like "`""""`" (which passes the validation patch) is escaped to produce the string "`\"\"\"\"`" at the sink, which is of length 8.

This example shows that due to the sanitization operation in the target function, we get a length difference in the post-image languages even though the validation patch has already blocked all strings longer than 4. To address this issue we generate a length patch that blocks any input string that results in a string longer than 4 characters at the target sink even if the input string itself is shorter than 4 characters. For example, the length patch blocks the string "`"a"`" although it has 3 characters only since it will result in the string "`\"a\"`" of length 5 at the sink which is longer than 4 characters. On the other hand, the string "`foo`" will not be blocked by the length patch since it will reach the sink as it is, 3 characters long.

Figure 3 shows the length patch-function for our example. Note that the function assumes that the validation patch function is applied before it so it only blocks things not blocked by the validation patch function. In section 4.3 we explain how to automatically generate the length patch-function.

Sanitization patch.

The purpose of this final phase is to take care of the differences that are due to sanitization operations. Our goal is to make sure that the post-image of the repaired function is a subset of the post-image of the reference function.

In our example, there is one sanitization operation in the reference function in which the character '`<`' is deleted. Even after application of the validation and length patches, this behavior would not be fully replicated at the repaired target function. Although the validation patch will prevent some strings such as "`<<<`" from reaching the sink at the repaired function, there are still other strings, such as "`a<b`" for example, that will still be in the post-image of the repaired function but not in the post-image of the reference function, since the character '`<`' gets deleted. The goal of the sanitization patch is to remedy such situations, and make sure that the sanitization operations in the target function are as strong as the sanitization operations in the reference function.

Unlike the previous two phases, the sanitization patch does not block the input strings that are found in the difference between the post-images of the target and reference functions. Instead we use an algorithm called *min-cut algorithm* to generate a sanitization code that will delete (or escape) certain characters in the input strings such that the difference between the two post-images is removed. Using this min-cut algorithm, our differential repair algorithm will generate the sanitization patch-function shown in Figure 3. This function does not block input strings that contain the character '`<`', but rather, deletes this character from these input strings and returns the corresponding string without that character. This repair simulates the same sanitization behavior of the reference function in the new repaired function. In section 4.3 we explain the min-cut algorithm and how to automatically generate the sanitization patch.

Given the final sanitization phase, one might think that the first two phases are redundant. However, without the first two phases, the repair generated by our approach can become too conservative by rejecting all input strings or by deleting all characters from the input string. Dividing the repair generation to three separate phases enables us to generate a combined repair that is not overly conservative.

The final result of the differential repair algorithm for our running example is shown in Figure 3. The *repaired function*, is obtained by composing the three patch-functions, in the order in which they were introduced here, with the original target function.

3. MODELING SANITIZER FUNCTIONS

As we discussed above, the differential repair algorithm takes two validation and sanitization functions as input. In this section we will give a characterization of the functions that our differential repair algorithm takes as input.

Input validation and sanitization operations in web applications can be characterized using three types of functions: 1) *pure validator*, 2) *pure sanitizer* and 3) *validating-sanitizer* functions. A pure validator is a total function:

$$F_v : \Sigma^* \rightarrow \{\bot, \top\}$$

that takes a string $s \in \Sigma^*$ and returns either $\top$ indicating that the string is valid and should be accepted or $\bot$ indicating the string is not valid and should be rejected. Note that, a pure validator does not change the value of the input string, it either accepts or rejects it as it is.

A pure sanitizer is a total function:

$$F_s : \Sigma^* \rightarrow \Sigma^*$$

that maps an input string $s \in \Sigma^*$ to an output string $s' \in \Sigma^*$. Note that, a pure sanitizer does not reject any input string, however, it may modify some of the input strings.

A validating-sanitizer is a function:

$$F_{vs} : \Sigma^* \rightarrow \{\bot\} \cup \Sigma^*$$

that takes an input string $s \in \Sigma^*$ and either returns $\bot$ indicating that s is invalid or maps s to output string $s' \in \Sigma^*$. Note that, a validating-sanitizer may reject some inputs and modify some others. For the rest of the paper we call a validating-sanitizer function a sanitizer for short.

In this paper, we model all input validation and sanitization operations in web applications as sanitizers. Note that, one can simulate a pure validator using a sanitizer: If an input is rejected by the validator, it is rejected by the sanitizer and if it is accepted by the validator it is returned without modification by the sanitizer. Obviously, any pure sanitizer is also a sanitizer that never rejects an input. Hence, by just focusing on sanitizers we are able to analyze all three types of behavior.

We extract one sanitizer function per input field which characterizes all the validation and sanitization operations that are used for that particular field. Validation and sanitization operations involve use of regular expressions and validation operations such as string *match*, *substring*, and sanitization operations such as string *replace, trim, addslashes, htmlspecialchars,* etc. In section 6 we will discuss how to extract sanitizer functions from a web application and when to map two functions to each other.

Differential Repair Problem.

Given a target sanitizer function F_T and a reference sanitizer function F_R, the goal of differential repair is to generate a new sanitizer function F^P, called a patch, such that when F_T is patched by composing it with F^P, the resulting repaired function returns a string only if F_R and F_T can both return that string. I.e., we want to make sure that a string is not in the post-image of the repaired function if it is not in the post-image of F_T or F_R. In order to formalize this, let us first define the sanitizer composition as follows: Given two sanitizer functions F_1 and F_2, their composition, $F_1 \circ F_2 : \Sigma^* \rightarrow \Sigma^* \cup \{\bot\}$, is a sanitizer function defined as:

$$F_1 \circ F_2(x) = \begin{cases} \bot & \text{if } F_2 = \bot \\ F_1(F_2(x)) & \text{if } F_2(x) \neq \bot \end{cases}$$

Now, let us also define the difference between the post-images of two sanitizer functions F_1 and F_2 as follows:

$$\textsc{Diff}(F_1, F_2) = \{x \mid \exists y \in \Sigma^* : F_1(y) = x \wedge (\forall z \in \Sigma^* : F_2(z) \neq x)\}$$

which is the set of strings that are in the post-image of F_1 but not in the post-image of F_2. Given this definition, the differential repair problem is to automatically construct a patch F^P such that $\textsc{Diff}(F_T \circ F^P, F_R) = \emptyset$, which means when we compose F_T with F^P we want to make sure that the result, $F_T \circ F^P$, is at least as strict as F_R, i.e., its post-image is a subset of the post-image of F_R. We call this new composed function the *differential repair* F_{DR}, where $F_{DR} = F_T \circ F^P$. Note that, due to the way we are constructing the differential repair, by composing the target function F_T with the automatically generated patch F^P, we guarantee that the repaired function F_{DR} is at least as strict as F_T, i.e., its post-image is also a subset of the post-image of F_T.

4. DIFFERENTIAL REPAIR

Given a target sanitizer F_T and a reference sanitizer F_R, our differential repair algorithm consists of three phases that produce three patches: (1) The *validation patch generation* phase produces F^V, (2) the *length patch generation* phase produces F^L, and (3) the *sanitization patch generation* phase produces F^S. The result of our differential repair algorithm is a patch that is the composition of these three individual patches: $F^S \circ F^L \circ F^V$ and the repair we generate is the composition of this patch with the target function, i.e., $F_{DR} = F_T \circ F^S \circ F^L \circ F^V$.

Our differential repair algorithm is shown in Algorithm 1. The algorithm takes a target sanitizer F_T and a reference sanitizer F_R as input and generates sanitizer F_{DR} as output which corresponds to differential repair of F_T with respect to F_R. Our differential repair algorithms is based on automata based symbolic string analysis, and uses Deterministic Finite Automata (DFA) to represent sets of strings. For example, it computes post or pre-images of given sanitizers as DFA (where the set of strings accepted by the automaton corresponds to the post or pre-image of a sanitizer). In Algorithm 1, each variable that has a name starting with M represents a DFA. The operations $\cap, \cup, \setminus, —$ are automata operations that generate automata that accept the intersection, union, difference and complement of the languages of the given automata, respectively, and the $=$ operation tests the language equivalence between two automata. The function $\mathcal{A}(L)$ takes a regular language $L \subseteq \Sigma^*$ and returns a DFA that recognizes L. We also use $\mathcal{L}(M)$ to denote the language accepted by the automaton M. The variables with a name starting with F represent sanitizers. In the remaining part of this section we discuss the three phases of the Algorithm 1.

Algorithm 1 $\textsc{DifferentialRepair}(F_T, F_R)$

```
1: M1 := A(Pre+⊥(FR));
2: M2 := A(Pre+⊥(FT));
3: if (L(M1 \ M2) ≠ ∅) then
4:    M^V := M1 \ M2;
5:    F^V := GenerateBlockingSimulator(M^V);
6: else
7:    F^V := IdentityFunction; M^V := A(∅);
8: end if
9: M1 := A(Post+(FR, Σ*));
10: M2 := A(Post+(FT, L(M^V)));
11: Md = M2 \ M1;
12: if (L(Md) ≠ ∅) then
13:    if (len_min(M2) < len_min(M1) ∨ len_max(M2) > len_max(M1))
       then
14:       M3 := RestrictLength(M2, M1);
15:       M^L := A(Pre+(FT, L(M2 \ M3)));
16:       F^L := GenerateBlockingSimulator(M^L);
17:       M2 := M3;
18:    else
19:       F^L := IdentityFunction;
20:    end if
21:    Md := M2 \ M1;
22:    if (L(Md) ≠ ∅) then
23:       Mmc := A(Pre+(FT, L(Md)));
24:       Σmc := Mincut(Mmc);
25:       F^S := GenerateSanitizer(Σmc, M1);
26:    else
27:       F^S := IdentityFunction;
28:    end if
29: else
30:    F^S := F^L := IdentityFunction;
31: end if
32: FDR := FT ∘ F^S ∘ F^L ∘ F^V;
33: return FDR;
```

4.1 Phase I: Validation Patch Generation

Our goal is to generate a validation patch F^V such that:

$$\forall x \in \Sigma^* : F_R(x) = \bot \Rightarrow F_T \circ F^V(x) = \bot,$$

i.e., the validation patch F^V guarantees that $F_T \circ F^V$ does not accept inputs that F_R rejects.

In order to compute the validation patch, we first need to identify the set of strings that are rejected by F_T and F_R. We call this the *negative pre-image* of a sanitizer. For a given sanitizer function F, this set is defined as:

$$\textsc{Pre}_\bot(F) = \{s \mid F(s) = \bot\}$$

In general, it is not possible to compute the pre or post-image of a sanitizer precisely since string analysis is an undecidable problem. We use automata-based backward symbolic string analysis techniques discussed in Section 5 to

compute an over approximation of the negative pre-image, $\text{PRE}^+_\perp(F)$, where $\text{PRE}^+_\perp(F) \supseteq \text{PRE}_\perp(F)$. This means that, we may conclude that certain strings are rejected by F when they are not. On the other hand, since we are computing an over-approximation, any string that is rejected by F is guaranteed to be in $\text{PRE}^+_\perp(F)$. Since we are using automata-based symbolic string analysis, the result of the negative pre-image computation is an automaton that accepts the language $\text{PRE}^+_\perp(F)$, and we denote this automaton as $\mathcal{A}(\text{PRE}^+_\perp(F))$.

Computing the pre-image of a sanitizer can be complicated due to the interaction of validation and sanitization operations. For example, consider the code segment:

```
trim($x);
if ($x == "")
   exit("String is empty");
```

where the input string is first sanitized by the `trim` operation and then it is validated by comparing the resulting string to empty string. To compute the set of inputs blocked by this piece of code (i.e., its negative pre-image) we need to compute the pre-image of the set of strings that satisfy the condition `$x == ""` after the `trim` operation. This is a language which is the Kleene closure of the union of all space characters that are removed by the `trim` operation.

We compute the negative pre-image of a sanitizer using automata-based backward symbolic string analysis. The analysis starts at a sink (where the sanitizer terminates) and goes backwards. The set of strings that can reach to a particular program point is represented as an automaton that accepts the corresponding set of strings. Our automata-based string analysis implements the pre-image and post-image computations for all common validation and sanitization operations as discussed in Section 5. Given a set of input strings as output of a string operation, pre-image computation computes the set of input strings that would result in that output set. Post-image computation does the reverse. The negative pre-image of a sanitizer is computed by a fixpoint computation that repeatedly applies the pre-image computations and uses automata-based widening [] to achieve convergence as discusses in Section 5.

We use this automata-based backward symbolic string analysis in lines 1 and 2 of Algorithm 1 two construct two automata M_1 and M_2, that accept an over-approximation of the negative pre-images of F_T and F_R, respectively, where $\mathcal{L}(M_1) = \text{PRE}^+_\perp(F_R)$ and $\mathcal{L}(M_2) = \text{PRE}^+_\perp(F_T)$. The next step (line 3) checks if the reference function F_R rejects more input values than the target function F_T by computing the difference between negative pre-images of M_1 and M_2. If the difference is empty then F^V is assigned the identity function (line 7) which is a sanitizer function that returns the input as it is without blocking any value (i.e., it is a no-op). If the difference is not empty, the target function must be patched to reject the values rejected by the reference function. To achieve this we automatically generate a patch that rejects only the strings that are rejected by F_R but not F_T.

Note that the validation patch we generate is not sound due to over-approximation of the negative pre-image of the target function F_T. The set of strings that are in $\text{PRE}^+_\perp(F_R) \cap (\text{PRE}^+_\perp(F_T) \backslash \text{PRE}_\perp(F_T))$ will not be blocked by the patch we generate, whereas they should be blocked in order to reach our precise goal. We can make the validation patch sound by blocking all the strings in $\text{PRE}^+_\perp(F_R)$ without computing the set difference with $\text{PRE}^+_\perp(F_T)$, but, that would result in generation of a validation patch in many cases even when it is not necessary. Our experiments indicate that the imprecision in our pre-image computation is not a problem in practice since for all the examples we manually checked we observe that $\text{PRE}^+_\perp(F_R) \cap (\text{PRE}^+_\perp(F_T) \backslash \text{PRE}_\perp(F_T)) = \emptyset$.

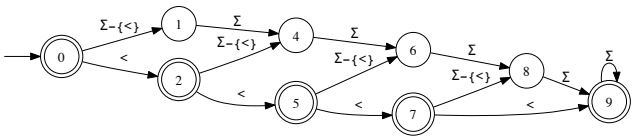

Figure 3: The validation patch automaton M^V for the example in Figure 1. The validation patch F^V blocks the strings accepted by this automaton.

Figure 4 shows the validation patch automaton M^V that is automatically generated for the example shown in Figure 1 where Σ represents the ASCII characters. To save space we collapsed all transitions between any two states s_i and s_j into one transition t_{ij}. We annotate this transition with a set of characters $\Sigma_C \subseteq \Sigma$ such that if a character c is in Σ_C then there is a transition on c between s_i and s_j. The sink state along with transitions into and out of it are omitted.

Since our analysis represents the set of strings at each program point using DFA, we generate the patch repair function F^V based on the DFA that is computed by our analysis. The validation patch code that is generated with GENERATEBLOCKINGSIMULATOR filters the inputs by simulating the resulting automaton M^V in Figure 4 to determine if the input string is accepted by M^V. If the input string is accepted by the automaton M^V, then F^V will return $\perp$ to block the input, otherwise it will return the input string without modification.

4.2 Phase II: Length Patch Generation

The goal of length patch generation is to generate a patch F^L such that:

$$\forall x \in \Sigma^* : \\ ((\exists y, z \in \Sigma^* : |F_R(y)| \leq |F_T \circ F^V(x)| \leq |F_R(z)|) \Rightarrow F^L(x) = x) \wedge \\ (\neg(\exists y, z \in \Sigma^* : |F_R(y)| \leq |F_T \circ F^V(x)| \leq |F_R(z)|) \Rightarrow F^L(x) = \perp)$$

i.e., given the target function F_T composed with the validation patch F^V, F^L rejects any input string that will cause the output of $F_T \circ F^V$ to contain a string of length longer or shorter than all the strings in the output of the reference function F_R.

The validation patch makes sure that any input string rejected by the reference sanitizer is also rejected by the repaired target sanitizer. However, this does not mean that the set of strings that are returned by the repaired target sanitizer and the reference sanitizer are the same after the validation patch since they may be using different sanitization operations. The length patch is the first step in establishing that the repaired target sanitizer does not return any string that is not returned by the reference sanitizer. The length patch makes sure that the length of any string returned by the repaired target function is not larger or smaller than all the strings returned by the reference sanitizer.

The lines 9-20 in Algorithm 1 construct the length patch. The lines 9 and 10 compute the automata that accept the set of strings that are returned by the reference sanitizer and the

target sanitizer that is composed with the validation patch. Given $L \subseteq \Sigma^*$, the set of strings returned by the sanitizer F when its input set is restricted to L is defined as:

$$\textsc{Post}(F, L) = \{s \mid \exists s' \in L : \exists s \in \Sigma^* : F(s') = s\}$$

We call this the *post-image* of sanitizer F with respect to L. Due to undecidability of string analysis we compute an over-approximation of this set, namely, $\textsc{Post}^+(F, L) \supseteq \textsc{Post}(F, L)$. The lines 11 and 12 in Algorithm 1 check if there are any strings that are returned by the target sanitizer composed with the validation patch that are not returned by the reference sanitizer by checking if $\textsc{Post}^+(F_T, \mathcal{L}(\overline{M^V}))\setminus \textsc{Post}^+(F_R, \Sigma^*) = \emptyset$. If the difference is empty, then we consider $F_T \circ F^V$ to be as strict as F_R and the analysis concludes by assigning IdentityFunction (i.e., no-op) to length and sanitization patches F^L and F^S (line 30).

Note that, due to over-approximation in our analysis, it is not guaranteed that $F_T \circ F^V$ is as strict as F_R even if the difference is empty. However, again manual inspection of our experiments indicate that our approximate analysis always finds the differences if they exist since the precision of our post-image computation is quite good in practice.

If a difference is found, then we check if the difference corresponds to a length difference in line 13. Let us first define len_{max} and len_{min} for an automaton. Given an automaton M, $len_{max}(M) = \infty$ if M accepts an infinite set, and $len_{max}(M)$ is the length of the longest string accepted by M otherwise. We can check if $len_{max}(M) = \infty$ by checking if there are cycles in M on any path from the starting state to an accepting state. If there is at least one cycle, then $len_{max}(M) = \infty$. If there are no cycles, then $len_{max}(M)$ is finite, and we use a depth first search to compute the length of the longest string accepted by M. On the other hand, given an automaton M, $len_{min}(M)$ is the length of the shortest string accepted by M. If the start state is an accepting state then $len_{min}(M) = 0$. Otherwise, $len_{min}(M)$ is computed by finding the length of the shortest path from the start state to an accepting state.

If a length difference is found, then we restrict the length of the set of strings accepted by F_T to remove the length difference using the following operation in line 14:

$$\textsc{RestrictLen}(M_2, M_1) \equiv M_2 \cap \bigcup_{i=len_{min}(M_1)}^{len_{max}(M_1)} \Sigma^i$$

After the length restriction, in line 15, we use the pre-image computation to compute an over-approximation of the set of input strings that cause the length discrepancy. Given a set of strings $L \subseteq \Sigma^*$ returned by a sanitizer function F, we call the set of input strings that are mapped by F to L the pre-image of F with respect to L and we define it as:

$$\textsc{Pre}(F, L) = \{s \mid \exists s' \in L : F(s) = s'\}$$

Again, due to over-approximation, we compute the set $\textsc{Pre}^+(F, L) \supseteq \textsc{Pre}(F, L)$. This over-approximation may result in blocking input strings that do not contribute to the length discrepancy.

In line 16 we generate the length patch F^L that blocks the strings that are accepted by the automaton M^L in Figure 5 and returns the strings rejected by M^L without any change. Figure 5 shows the length patch automaton M^L that our algorithm computes for the example shown in Figure 1.

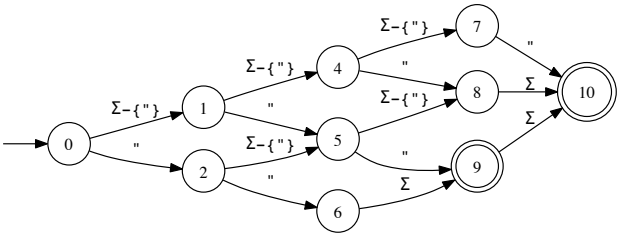

Figure 4: The length patch automaton M^L for the example in Figure 1. The length patch F^L rejects the strings accepted by this automaton.

4.3 Phase III: Sanitization Patch Generation

The third and final phase in the repair algorithm is the sanitization patch generation, which results in a patch function F^S such that:

$$\begin{aligned}\forall x \in \Sigma^* \quad : \quad & (\forall y \in \Sigma^* : F_R(y) \neq x) \Rightarrow \\ & (\forall z \in \Sigma^* : F_T \circ F^S \circ F^L \circ F^V(z) \neq x)\end{aligned}$$

which means that, after adding the sanitization patch F^S to the previously generated patches, we want the differential repair $F_{DR} = F_T \circ F^S \circ F^L \circ F^V$ to be as strict as F_R in terms of the set of strings it returns.

The lines 21-28 in Algorithm 1 generate the sanitization patch. First, in the lines 21, 22, we check if there is a difference between what F_T returns (after validation and sanitization patches are applied) and what F_R returns assuming any input. If no difference is found, then we consider $F_T \circ F^L \circ F^V$ to be as strict as F_R and the analysis concludes by assigning IdentityFunction to F^S (line 27). This indicates that there is no sanitization patch. Note that, as we discussed before, due to over-approximation our repair algorithm can miss differences, however we have not observed this in our experiments.

If a difference is found, then, in the line 23, we compute an over-approximation of the set of input strings that result in such a difference. The set $\mathcal{L}(M_{mc})$ represents an over-approximation of the set of all input strings that are the cause of the difference between the set of strings returned by F_R and $F_T \circ F^L \circ F^V$. We call M_{mc} the *mincut* automaton and in the line 24 we use this mincut automaton to generate a mincut alphabet (as explained below), such that if the symbols in the mincut alphabet are removed from the input strings, then the difference between the post-images of F_R and $F_T \circ F^L \circ F^V$ disappear. Then, in the line 25, we generate the sanitization patch F^S to either delete or escape the set of symbols in the mincut alphabet. Finally, in the lines 32 and 33, we construct and return the differential repair function F_{DR} as the composition of the target function F_T with the three patch functions generated during the three phases of the repair algorithm.

The mincut algorithm [32] takes the DFA M_{mc} as input, and produces a set of characters Σ_{mc} such that sanitization repair function F^S modifies a given input string in such a way that the modified string is not accepted by the M_{mc}. In the basic case, F^S modifies the input strings by just deleting a set of characters using the `replace` function (later, we extend this basic case to handle escaping characters instead of 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, then, is, how do we identify

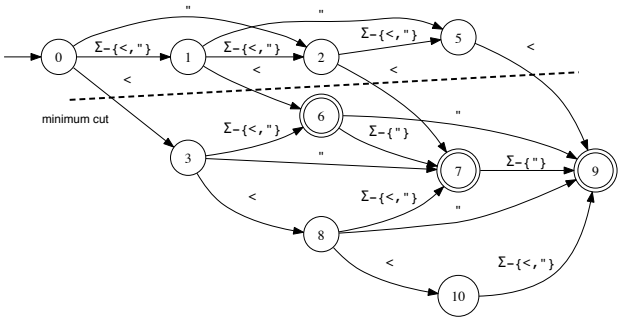

Figure 5: The mincut automaton M_{mc} for the example in Figure 1. The dotted line shows the mincut edges with the corresponding mincut alphabet $\{<\}$.

the set of characters to be deleted?

First, we will formalize this problem in automata-theoretic terms [32]. We say $S \subseteq \Sigma$ is an *alphabet-cut* of M, if $\mathcal{L}(M) \cap L_{\bar{S}} = \emptyset$, 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, $|S_{min}| \leq |S|$.

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.

Note that, if M 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 $\mathcal{L}(M) \neq \emptyset$ (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).

It has been shown that the min-alphabet-cut problem is NP-hard [32], so, 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. One heuristic solution is to minimize the number of edges in a cut rather than the number of distinct alphabet symbols. 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 [10] 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}.

Figure 6 shows the mincut automaton M_{mc} for our running example in Figure 1 along with the mincut edges which correspond to the mincut alphabet $\Sigma_{mc} = \{<\}$.

Once we compute an alphabet-cut Σ_{mc}, we generate the sanitization patch F^S with a `replace` statement that deletes the symbols in Σ_{mc} from the input, making sure that the resulting string does not match M_{mc}. Although the function F^S is a sound repair that will guarantee that $POST^+(F_T \circ F^S \circ F^L \circ F^V, \Sigma^*) \subseteq POST^+(F_R, \Sigma^*)$, we apply two heuristics to generate more accurate repair functions.

The first heuristic is the *escape* heuristic. We look at the automaton M_1 generated in line 9 of the Algorithm 1 (which represents all the string values returned by F_R), and check if all the characters in the mincut alphabet Σ_{mc} are always preceded by the same single character e. If that is the case, we we call the character e the escape character. Formally speaking, given DFA $M_1 = \langle Q_1, q_0, \Sigma, \delta_1, F_R \rangle$, we check that $\forall q \in Q_1, \forall c \in \Sigma_{mc} : \delta_1(q, c) \neq sink \Rightarrow (\forall q' \in Q_1 : \delta_1(q', c') = q \Rightarrow c' = e)$. If this is the case, then the sanitization patch F^S we generate uses the `replace` operation to escape all the characters in the mincut alphabet Σ_{mc} (instead of deleting them) by prepending them with the escape character e.

The second heuristic we use is the *trim* heuristic. Here, if we get a mincut Σ_{mc} which contains space characters, we first check if M_1 accepts any string that contains a space character (which can be determined by checking if transitions on space characters always go to the sink). If not, then we generate a patch that deletes the space characters as in our basic mincut based patch generation algorithm. If M_1 does accept a string that contains a space character, then we check if it is the case that the strings accepted by M_1 do not start or end with space characters. Formally speaking, given DFA $M_1 = \langle Q_1, q_0, \Sigma, \delta_1, F_R \rangle$, we check that for all space character s $\delta_1(q_0, s) = sink$ and $\forall q \in Q_1 : \delta_1(q, c) \in F_R \Rightarrow c \neq s)$. If so, then we generate patch F^S which uses the `trim` function to delete the space characters from the beginning and end of each input string.

5. SYMBOLIC STRING ANALYSIS

In order to implement our differential repair algorithm we use forward and backward symbolic string analysis techniques to compute the post-image $\textsc{Post}^+(F, L)$, pre-image $\textsc{Pre}^+(F, L)$ and negative pre-image $\textsc{Pre}^+_\perp(F)$ of sanitizer functions. Before we apply the differential repair algorithm, we first extract the sanitizer functions from source code (as discussed in Section 6) and represent them in a language-agnostic intermediate representation. Our intermediate representation supports the string validation and sanitization operations that are used in validation and sanitization functions. Each extracted sanitization function can contain multiple *exit* statements which correspond to rejection of the input string, and a return statement that returns the output of the sanitizer. To conduct forward and backward symbolic string analyses on this intermediate representation, we implement the pre- and post-image computations for each validation and sanitization operation in our intermediate representation. As a symbolic representation we use Deterministic Finite Automata (DFA) where each string expression in the program is associated with a DFA, and the language accepted by the DFA represents the possible values that the string expression can take at any given execution. We use the symbolic DFA representation provided by the MONA DFA library [6], in which transition relation of the DFA is represented as Multi-terminal Binary Decision Diagrams (MBDDs). We implement forward and backward symbolic

analyses that correspond to flow-sensitive and path-sensitive symbolic reachability analyses, where we iteratively compute an over approximation of the least fixpoint that corresponds to the reachable values of the string expressions by using the pre- or post-image computations for the string validation and sanitization operations at each iteration [1, 33, 31]. Since DFAs can represent infinite sets of strings, the fixpoint computations are not guaranteed to converge. To alleviate this problem, we use the automata widening technique proposed by Bartzis and Bultan [5] to compute an over-approximation of the least fixpoint. Briefly, the widening operator merges those states belonging to the same equivalence class identified by certain conditions.

We implemented the automata-based pre and post-image computations for common string operations such as concatenation and language-based replacement that have been described in earlier work [33, 31], and we also implemented novel specialized algorithms for computing pre and post-image of specialized string sanitization operations such as `trim` and `addslashes`.

Specialized Replace Algorithms.

To increase the precision and performance of the differential repair, we developed a number of automata-based algorithms for computing the pre and post-images of frequently used string operations such as `trim`, `htmlspecialchars`, `addslashes`, `mysql_real_escape_string`, `tolower`, and `toupper`. These algorithms are more precise and more efficient than using the general replace algorithm to model these specialized operations [33]. The general replace algorithm consumes significantly more time and space since it relies on automata determinization which requires the use of subset construction algorithm which has an exponential complexity. On the other hand, the specialized algorithm we developed for the ESCAPE operation, for example, runs in linear time and its result is precise without any over-approximation. In the experimental results we demonstrate the improvement that we gain from these specialized algorithms.

Below we describe the post-image of four of the specialized replace operations that we implemented. We omit the description of pre-images and other operations due to space limitation.

POSTESCAPE(DFA M_1, char e, charset E): this automata operation escapes characters in E along with the escape character e itself in all strings in $\mathcal{L}(M_1)$ using the escape character e. It returns a DFA M such that $\mathcal{L}(M) = \{ w_1 e c_1 w_2 e c_2 \ldots w_k e c_k w_{k+1} \mid k > 0, w_1 c_1 w_2 c_2 \ldots w_k c_k w_{k+1} \in \mathcal{L}(M_1), \forall i : c_i \in E \cup \{e\}$ and $w_i \in \Sigma^* - (E \cup \{e\})^* \}$. An example of an escape function is PHP's `addslashes`. Notice that ESCAPE escapes all chars $c \in \{e\} \cup E$ without checking if they have already been escaped before. This may result in double escaping i.e. $w_1 e e w_2$ will become $w_1 e e e e w_2$.

POSTTRIMLEFT(DFA M_1, char s): this automata operation removes all the s characters from the beginning of strings in $\mathcal{L}(M_1)$ up to the first character that is not equal to s. It returns a DFA M such that $\mathcal{L}(M) = \{c_1 w \mid w_1 c_1 w \in \mathcal{L}(M_1), w_1 \in \{s\}^*$ and $w \in \Sigma^*$ and $c_1 \in \Sigma - \{s\}\}$.

POSTTRIMRIGHT(DFA M_1, char s): this automata operation removes all the s characters from the end of strings in $\mathcal{L}(M_1)$ up to the first character that is not equal to s. It returns a DFA M such that $\mathcal{L}(M) = \{w c_1 \mid w c_1 w_1 \in \mathcal{L}(M_1), w_1 \in \{s\}^*$ and $w \in \Sigma^*$ and $c_1 \in \Sigma - \{s\}\}$.

POSTREPLACECHAR(DFA M_1, char c, String S): this automaton operation replaces a single char c with a string s in all strings in $\mathcal{L}(M_1)$. It returns a DFA M such that $\mathcal{L}(M) = \{w_1 S w_2 S \ldots w_k S w_{k+1} \mid k > 0, w_1 c w_2 c \ldots w_k c w_{k+1} \in \mathcal{L}(M_1), w_i \in (\Sigma - \{c\})^*\}$. This operation can be used to model string operations such as PHP's `htmlspecialchars` efficiently.

We would like to discuss the automata algorithms that we developed to compute the pre- and post-images of the ESCAPE operation that we mentioned above as an example.

POSTESCAPE(M_1, e, E) implementation: Given $M_1 = \langle Q_1, q_{10}, \Sigma, \delta_1, F_1 \rangle$ the result DFA $M = \langle Q, q_0, \Sigma, \delta, F \rangle$ is constructed as follows: For each state q_i that has at least one out transition $(q_i \xrightarrow{c} q_j)$ on a character $c \in \{e\} \cup E$ (1) we mark each transition $(q_i \xrightarrow{c} q_j)$ out from q_i to a state q_j, (2) we add a new state q_k and a new transition $(q_i \xrightarrow{e} q_k)$ on escape character e, (3) we move each marked transition $(q_i \xrightarrow{c} q_j)$ to become a transition $(q_k \xrightarrow{c} q_j)$. The resulting automaton does not have nondeterminism which means that we avoid the use of subset construction algorithm for determinization.

PREESCAPE(M_1, e, E) implementation: Given $M_1 = \langle Q_1, q_0, \Sigma, \delta_1, F_1 \rangle$ we first preprocess M_1 to partition all transitions $(q \xrightarrow{e} q')$ into two sets of transitions: Escaping transitions T_g and escaped transitions T_e such that: $\forall q, q' \in Q_1 : (q \xrightarrow{e} q') \in T_e \Rightarrow \exists q'' \in Q_1 \colon (q'' \xrightarrow{e} q) \in T_g$.

Notice that in M_1, a transition $(q \xrightarrow{e} q')$ can not be escaping and at the same time being escaped, i.e., $T_e \cap T_g = \emptyset$. Otherwise we will have a string $w_1 e e c w_2 \in \mathcal{L}(M_1)$ where $w_1, w_2 \in \Sigma^*$ which contradicts the definition of POSTESCAPE(M_1, e, E). We can formalize this condition as follows: There is no path $q_0, \ldots, q_{i-1}, q_i, q_{i+1}, q_{i+1}, \ldots, q_f$ in M_1 where $q_f \in F$, $\delta(q_{i-1}, e) = q_i$, $\delta(q_i, e) = q_{i+1}$ and $\delta(q_{i+1}, c) = q_{i+2}$ where $c \in \{e\} \cup E$. During the analysis, we enforce this condition by applying PREESCAPE on $M_1 \cap$ ESCAPE$(\mathcal{A}(\Sigma^*), e, E)$. Using the same reasoning we conclude that (1)$\forall (q \xrightarrow{e} q') \in T_g : q' \notin F$, (2) $\forall c \notin \{e\} \cup E$, $\forall (q \xrightarrow{e} q') \in T_g : \delta(q', c) = sink$, (3) $\forall q \in Q_1 (\forall c \in \{e\} \cup E : \delta(q, c) \neq sink \Leftrightarrow \forall c' \notin \{e\} \cup E : \delta(q, c') = sink)$, and (4)$\delta(q_0, e) \neq sink \Rightarrow (q_0 \xrightarrow{e} q') \in T_g$.

Using these results, we compute T_g and T_e using a depth first traversal starting from the start state q_0. We then compute the set of *escaped states* Q_e which is the set of states that has all input transitions in T_g. Due to the preconditions we stated earlier, all transitions on e must be either coming into an escaped state or going out from it. Also all transitions on escaped characters $c \in \{E\}$ must be going out from an escaped state. Finally, given Q_e, we construct the new DFA by removing all states $q_k \in Q_e$ such that: (1) all transitions $(q_i \xrightarrow{e} q_k)$ are removed, and (2) each transition $(q_k \xrightarrow{c} q_j)$ is added, as an out transition, to all states q_i where a transition $(q_i \xrightarrow{e} q_k)$ was removed. Based on the conditions we discussed above on M_1, this last step can be done without determinization and subset construction.

6. IMPLEMENTATION & EXPERIMENTS

We have implemented our differential repair algorithm using our symbolic string analysis library Stranger. Our differential repair algorithm implementation is available online [26] along with the source code. In order to evaluate our repair algorithm we experimented on five open-source PHP applications 1) PHPNews v1.3.0 (news publishing software),

2) UseBB v1.0.16 (forum software), 3) Snipe Gallery v3.1.5 (image management system), 4) MyBloggie v2.1.6 (weblog system), 5) Schoolmate v1.5.4 (school administration software) along with a number of JavaScript sanitizer-function benchmarks from various websites [1]. We ran all the experiments on an Intel I5 machine with 2.5GHz X 4 processors and 32 GB of memory running Ubuntu 12.04.

Extracting Sanitizer Functions. Before we run our differential repair algorithm, we need to extract sanitizer functions from the client and/or server side and map them to each other to generate target, reference sanitizer pairs. We built a crawler using HTMLUnit [11] to find input fields and corresponding sinks in a PHP web application. When the crawler hits a web page with an HTML form, it fills it out automatically using a set of pre specified profiles and submits it. Then, for each HTML input field, JavaScript validation and sanitization code is dynamically extracted as described in [1], along with the HTML constraints on the field, resulting with one sanitizer function per each input field. On the server-side, we also collect the execution traces to figure out the inputs and the sinks (where the inputs flow into) during crawling. We use that information later on to map server-side sanitizer functions to client-side sanitizer functions. Finally, we use the front end of Pixy [13] to statically extract the corresponding sanitizer functions from the server-side. We augmented Pixy code to add path sensitivity and support for PHP5. Client-client sanitizers are mapped to each other based on the type of data they operate on (i.e., email address, phone number, ... etc). Server-server mapping is done within the same application and across different applications based on field names that are extracted from the PHP `$_POST` and `$_GET` arrays.

Results. Table 1 shows the total number of target-reference sanitizer pairs we analyzed and the number of patches generated at each phase of the algorithm for four categories: Client-server (C-S) where the server (target) is patched against the client (reference), server-client (S-C) where the client (target) is patched against the server (reference), server-server (S-S) and client-client (C-C). Note that for the server-server and client-client cases we analyze each pair twice by considering each sanitizer function as target once and as reference once. The most commonly generated patches are validation patches which indicates that inconsistencies in validation policies are common. There are also a significant number of sanitization patches generated, except that the client-server pairs generated no sanitization patches. We checked the client-server pairs manually and confirmed that this is an accurate result (i.e., there are no cases where the sanitization at the client-side is stronger than the sanitization at the server-side). Among 49 server-server validation patches, 48 of them are generated for the pairs that are from different applications. We found 14 server-server sanitization patches within the same application, which indicates inconsistent sanitization policy within the application. For server-server and client-client, there are no length patches since the validation patches in these cases do not involve length restrictions.

Table 2 shows the details of sanitization patches and results of the mincut algorithm. As we can see our mincut heuristics were able to identify 41 trim operations and 10 escape operations. This identification is very helpful since applying sanitization patches that escape certain characters is not idempotent which is critically important to be avoided for server-client pairs. In the client-client sanitization patches there was an interesting case in which the mincut was Σ. The reason was that the languages of the post-image DFAs were disjoint which means that the two functions return completely different sets of strings (in this case the discrepancy was due to the presence/absence of the dash symbol in phone number fields).

Table 1: Number of Patches Generated

	#pairs	#valid.	#length	#sanit.
C-S	122	61	11	0
S-C	122	53	2	30
S-S	206	49	0	33
C-C	19	34	0	5

Table 2: Sanitization Patch Results

	mincut avg size	mincut max size	#trim	#escape	#delete
S-C	4	10	15	10	20
S-S	3	5	23	0	20
C-C	7	15	3	0	2

Table 3 shows the memory and time performance of our repair algorithm. Rows I, II, and III corresponds to validation, length, and sanitization patch generation phases, respectively. Memory performance is represented as number of BDD nodes that is needed to represent a DFA where the size of each BDD node is 16 bytes. In general our algorithm is efficient both in terms of time and memory. The average total time for the algorithm is 1.35 seconds and the maximum time is 186.22 seconds. During our experiments we had to abort the algorithm in 36 among 469 pairs (%7.6) due to MONA's limit on BDD size. The main reason for exceeding the BDD size limit is length constraints with large numbers. For example, in one of the cases in the experiments, validation patch restricts the length of the language of the input strings to 255. Then, sanitization function htmlspecialchars in the target increases the maximum length of the post-image to 1275. Since we use finite automata to represent sets of strings, the automaton has to count the length of the strings with its states. For this reason, we can see from Table 3 that the second phase of the algorithm which deals with the length constraints has the highest time and memory consumption.

Table 3: Time and Memory Performance of Analysis

repair phase	DFA size (#bddnodes)		peak DFA size (#bddnodes)		time (seconds)	
	avg	max	avg	max	avg	max
I	997	32,650	484	33,041	0.14	4.37
II	129,606	347,619	245,367	4,911,410	9.39	168.00
III	2,602	11,951	4,822	588,127	0.17	14.00

Figure 7 shows a comparison in terms of time and memory (represented using the number of BDD nodes) between the performance of the generic replace algorithm and the specialized replace algorithms we presented in this paper. In our setup we computed the post-image of the two PHP functions `addslashes` (which does 3 replace operations) and `htmlspecialchars` (which does 5 replace operations) on the language $\bigcup_{i=0}^{l} \Sigma^l$. This setup is identical to a sanitizer function that restricts the length of its input i to a certain value l through branch condition `len(i) <= l`, and then sanitizes the input that passes the branch condition using one of the two functions. The x axis shows the length while the y

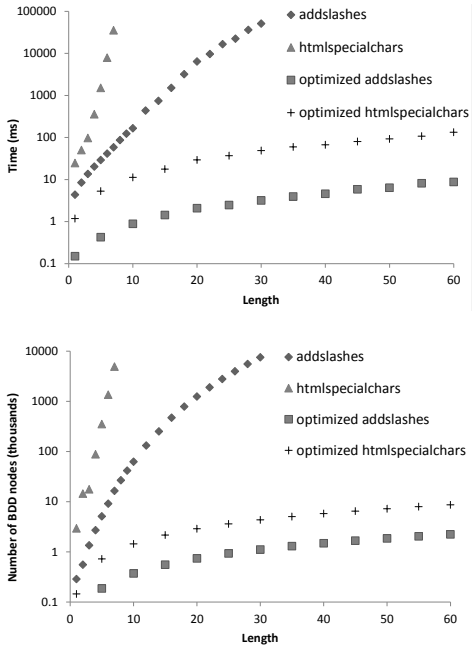

Figure 6: Time and Memory performance for generic replace and optimized/specialized replace algorithms.

axis shows the time and memory and uses a log scale. We notice that the generic replace grows exponentially as we increase the length while the specialized ones do not. For the generic replace, the analysis reaches MONA limit on BDD size at length 31 for `addslashes` and at length 8 for `htmlspecialchars`. On the other hand, for the specialized replace operations it took less than 5 seconds to run with length 1000 with negligible memory overhead.

7. RELATED WORK

Automatic program repair [2, 30, 29, 24, 25, 20, 22] became an active topic recently. Weimer et. al. [30, 29] repair programs using genetic programming by randomly mutating the abstract syntax tree (AST) of the program. Son et. al. [24, 25] find access control problems in PHP by comparing a possibly buggy AST with one that is considered to be correct and then patch the difference by inserting statements from the latter into the former. Unlike these syntax based approaches, our differential repair algorithm uses a semantic approach which enables us to generate precise repairs in multiple languages. In [20, 22] test suites are used to find bugs then symbolic execution is used to find constraints on variables that result in such bugs. Using the solution to the negation of these constraints, a patch is synthesized for the program such that it passes all test suites. Unlike this approach, our approach does not require a test suite and it can handle unbounded loops using fixpoint computation. Livshits et. al. [17] automatically place a set of sanitizers in a sanitizer free program based on a user defined policy and a flow graph. The sanitizers are placed in the flow graph such that they satisfy the specified policy and at the same time avoid idempotency problems. In our case we take into account the previous sanitization code and the way we generate the repair allows us to place it at the beginning of the code that is under repair without requiring a placement policy. Placing the repair before the original code, instead of changing the code, allows us to avoid interference with the original sanitization code that may have side-effects.

Differential analysis techniques [21, 16, 14, 15] typically stop after finding differences between different pieces of code without trying to repair it. In [21] differential symbolic execution is used to find differences between original and refactored code by summarizing procedures into symbolic constraints and then comparing different summaries using an SMT solver. SYMDIFF [14] computes the difference between two different functions in a language agnostic way by reducing both functions to Boggie [4] intermediate language then finds semantic differences using the Z3 SMT solver [18]. There are several specialized differential analysis techniques that focus on web applications. In NoTamper [7] the authors analyze client-side script code to generate test cases that are subsequently used as inputs to the server side of the application. Since the approach relies on dynamic (black-box) testing, it can suffer from limited code coverage. In a recent follow up paper [8], the same authors propose WAPTEC, which uses symbolic execution of the server code to guide the test case generation process and expand coverage. In addition to finding semantic differences, our work also generates a fix for such difference.

Static analysis of strings has been an active research area, with the goal of eliminating bugs caused by inadequate use of string validation and sanitization operations. In [19], multi-track DFAs, known as *transducers*, are used to model replacement operations in conjunction with a grammar-based string analysis approach. The resulting tool was used in detecting vulnerabilities in PHP programs. Wassermann et al. [27, 28] propose grammar-based static string analyses to detect SQL injections and XSS, following Minamide's approach. A more recent approach in static string analysis has been the use of finite state automata as a symbolic representation [3, 33]. In this approach, complex string manipulation operations, such as string replacement, can be modeled using automata representation. In order to guarantee convergence in automata-based string analysis, several widening operators have been used [9, 5, 33]. Constraint-based (or symbolic-execution-based) techniques represent a third approach for string analysis. Such techniques have been used for the verification of string operations in JavaScript [23] and the detection of security flaws in sanitization libraries [12].

8. CONCLUSIONS

We presented a novel differential repair algorithm that automatically repairs the semantic difference between two input validation and sanitization functions. The algorithm takes a target and a reference function as input and fixes the target function with respect to the reference function automatically without the need for any input specification or manual intervention. The repaired function provides stronger validation and sanitization than both the target and the reference functions. The repair is constructed by composing the target function with three automatically generated patch functions which we call the validation patch, the length patch and the sanitization patch. Our experiments on several web applications that contain PHP and JavaScript code show that our differential repair technique is successful in repairing differences between sanitizer functions.

9. REFERENCES

[1] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 947–957, Piscataway, NJ, USA, 2012. IEEE Press.

[2] J. Andersen and J. L. Lawall. Generic patch inference. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 337–346, Washington, DC, USA, 2008. IEEE Computer Society.

[3] 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 *Proceedings of the Symposium on Security and Privacy*, 2008.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 364–387, Berlin, Heidelberg, 2006. Springer-Verlag.

[5] C. Bartzis and T. Bultan. Widening arithmetic automata. In *In Computer Aided Verification 04*, pages 321–333, 2004.

[6] M. Biehl, N. Klarlund, and T. Rauhe. Algorithms for guided tree automata. In *First International Workshop on Implementing Automata, WIA '96, London, Ontario, Canada, LNCS 1260*. Springer Verlag, 1997.

[7] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

[8] P. Bisht, T. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. Waptec: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 575–586, New York, NY, USA, 2011. ACM.

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

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

[11] Gargoyle Software. HtmlUnit: headless browser for testing web applications. `http://htmlunit.sourceforge.net/`.

[12] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with bek. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IN 2006 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 258–263, 2006.

[14] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 345–355, New York, NY, USA, 2013. ACM.

[16] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 201–204, New York, NY, USA, 2010. ACM.

[17] 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, New York, NY, USA, 2013. ACM.

[18] Microsoft Inc. Z3 SMT Solver. `http://z3.codeplex.com`.

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

[20] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.

[21] S. J. Person. *Differential Symbolic Execution*. PhD thesis, Lincoln, NB, USA, 2009. AAI3365729.

[22] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 277–287, Piscataway, NJ, USA, 2012. IEEE Press.

[23] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. *Security and Privacy, IEEE Symposium on*, 0:513–528, 2010.

[24] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

[25] S. Son, K. S. McKinley, and V. Shmatikov. Fix me up: Repairing access-control bugs in web applications. In *NDSS*, 2013.

[26] UCSB VLab. Stranger: Semantic Differential Repair tool. `http://cs.ucsb.edu/~vlab/stranger/`.

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

[28] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software*

Engineering, ICSE '08, pages 171–180, New York, NY, USA, 2008. ACM.

[29] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, May 2010.

[30] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

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

[32] F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *International Conference on Software Engineering (ICSE)*, pages 131–134, 2011.

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