Verifying Client-Side Input Validation Functions Using String Analysis

Muath Alkhalaf Tevfik Bultan Jose L. Gallegos
Computer Science Department, University of California Santa Barbara, CA, USA
$\{$*muath,bultan,jlgallegos*$\}$*@cs.ucsb.edu*

***Abstract*—Client-side computation in web applications is becoming increasingly common due to the popularity of powerful client-side programming languages such as JavaScript. Client-side computation is commonly used to improve an application's responsiveness by validating user inputs before they are sent to the server. In this paper, we present an analysis technique for checking if a client-side input validation function conforms to a given policy. In our approach, input validation policies are expressed using two regular expressions, one specifying the maximum policy (the upper bound for the set of inputs that should be allowed) and the other specifying the minimum policy (the lower bound for the set of inputs that should be allowed). Using our analysis we can identify two types of errors 1) the input validation function accepts an input that is not permitted by the maximum policy, or 2) the input validation function rejects an input that is permitted by the minimum policy. We implemented our analysis using dynamic slicing to automatically extract the input validation functions from web applications and using automata-based string analysis to analyze the extracted functions. Our experiments demonstrate that our approach is effective in finding errors in input validation functions that we collected from real-world applications and from tutorials and books for teaching JavaScript.**

I. Introduction

A crucial problem in developing dependable web applications is the correctness of the input validation operations. One of the main forms of interaction between a user and a web application is through text fields, where the user types a text as input which is then parsed by the web application and converted to some specific type of data such as a date, a credit card number, or an e-mail address. A web application needs to separate valid user input from inputs that do not match the expected input type and prompt the user to re-enter an appropriate input if necessary.

Web applications typically use a three-tier architecture which consists of client-side code (executing at the user's machine that is running the browser), server-side code (executing at the web server) and the back-end database (storing the persistent data on a separate database server). In recent years, in order to improve efficiency and usability, web applications have started to migrate many of the computational tasks to the client-side code. This makes applications more responsive by reducing the need to send a request to the web server from the user's machine and wait for the response. Nowadays, many web applications include client-side input validation functions that check the user input and warn the user if the input is invalid without requiring any interaction with the web server.

This research is funded by the NSF grants CCF-0916112 and CNS-1116967 and by King Saud University.

In this paper, we focus on automated verification of client-side input validation functions for three main reasons: 1) *Security:* Client-side input-validation vulnerabilities are an emerging class of vulnerabilities that are due to errors in the client-side input validation functions [17]. 2) *Correctness:* Errors in the client-side input validation functions can cause valid inputs to be rejected without reaching the server. 3) *Performance:* Errors in the client-side input validation functions can degrade the performance by creating unnecessary communication between the client and the server.

In order to verify client-side input-validation functions in an existing web application, we first have to extract the input validation functions from the application. We do this by executing the application using both valid and invalid input values and track accesses to the input values in order to identify the input validation operations. We use dynamic slicing to find all the statements that are influenced by the input values that we track. We output the resulting slice in the form of an input validation function that returns true or false based on the input value.

We check the correctness of the automatically extracted input-validation functions using an automata-based string analysis. We use deterministic finite automata (DFA) to represent values that string expressions can take. At each program point, each string variable is associated with a DFA. We use a forward symbolic reachability analysis that computes an over-approximation of all possible values that string variables can take at each program point. Since convergence of the symbolic reachability analysis is not guaranteed without approximation, we use an automata based widening operation in the presence of loops. Our analysis is path-sensitive and handles the branch conditions by appropriately restricting the values that string variables can take at the true and false branches of a conditional branch. Our analysis is conservative in the sense that the set of string values we compute correspond to a superset of possible string values that a string expression can take at runtime.

We assume that each input validation function takes a string value as input and returns true if the input is valid (i.e., if the input is permitted by the input validation policy) or returns false otherwise. When we automatically extract an input validation function we extract it in this form. Using our string analysis we compute all possible input values that allows the program to reach to the "return true" statement. Then we check if this set of values are a subset of the language defined by the regular expression characterizing the input validation policy. If it is, then we know that the application implements the input validation policy correctly.

One easy way to conform to every input validation policy would be to reject all the inputs. In order to make sure that an input validation function is allowing at least some reasonable set of inputs, we write the input validation policies using two regular expressions, one (maximum policy) corresponding the largest set of strings that should be recognized as valid (i.e., everything outside this set should be definitely rejected) and one (minimum policy) corresponding to the smallest set of strings that should be recognized as valid (i.e., everything within this set should be definitely accepted). Having a minimum and a maximum validation policy helps us in two ways: 1) It provides more flexibility where an application does not have to match to a single policy exactly, but, instead, has to do validation that is at least as strict as the maximum policy and at least as permissive as the minimum policy. 2) Having a minimum policy allows us to check for the cases where an application erroneously disallows some valid user inputs. In order to check that the application is at least as strict as the maximum policy we look at the input values that reach the "return true" statement as we described above. However, in order to check that the application is at least as permissive as the minimum policy we use our string analysis to compute all possible input values that allows the program to reach to the "return false" statement. Then we check if the intersection of this set of values and the minimum policy is empty. If it is, then we conclude that the application is not rejecting any user input that should be valid according to the minimum policy.

II. An Overview

In this section we give an overview of how we model the client-side input validation function verification problem.

Client-side input validation functions: A client-side input validation function takes a string value from a web (HTML) form field and checks it against a certain policy. If none of the validation functions used for a form report a violation, then the form data is submitted to the server, otherwise it is not submitted. We make the following assumptions about the validation functions: A validation function takes at least one string input and checks it with respect to a certain policy and returns either true or false. True indicates that the string input conforms to the policy and false indicates otherwise.

Figure 1 shows an email validation function taken form a popular JavaScript book [14]. The function checks that the email is not empty and that it conforms to the given regular expression. There is a subtle error in this function. In the regular expression in line 10 the author wanted to specify that the first part of the email (before the @) only contains alphanumeric characters and ".", "-", "_" and "+". Other special characters such as "[" are not allowed. However `\.-_` includes all ASCII characters in the range between "." and "_" which includes the special character "[". The developer forgot to escape the "-" character which indicates a range of characters if it is not escaped.

```
1 function validateEmail(inputField, helpText){
2    if (!/.+/.test(inputField.value)) {
3        if (helpText != null)
4             helpText.innerHTML = "Please enter a value";
5        return false;
6    }
7    else {
8       if (helpText != null)
9            helpText.innerHTML = "";
10      if (
        !/^[a-zA-Z0-9\.-_\+]+@[a-zA-Z0-9-]+(\.[a-zA-Z0-9]
        {2,3})+$/.test(inputField.value)) {
11            if (helpText != null)
12                helpText.innerHTML = "Please enter an"
                     +" email address";
13            return false;
14      }
15      else {
16            if (helpText != null)
17                helpText.innerHTML = "";
18            return true;
19      }
20   }
21 }
```

Figure 1. A JavaScript input validation function.

Validation policies: We use two types of validation policies: *Max* and *Min*. The *Max* policy specifies the maximal set of strings that should be accepted while the *Min* policy specifies the minimal set of strings that should not be rejected. We use regular expressions for specification of the *Max* and *Min* policies. This is a natural choice since regular expressions are well-known and developers implementing input validation functions commonly use regular expressions in string manipulation functions. Figures 2 and 3 show a number of maximum and minimum input validation policies that we have used in our analysis. Each policy has two entries: the type of the input field that is checked against this policy and the specification of the policy as a regular expression. The standard syntax we use for specifying regular expressions is a subset of `preg` syntax that is used in JavaScript and other languages. For some simpler input types we have the same maximum and minimum policies.

For example the *Email* policy in Figure 2 specifies the correct value for an email address. It is a more restrictive policy than RFC5322 which specifies valid email addresses. This is due to the fact that some major email providers such as Hotmail are much more restrictive in their email address policy than the previous standard. Although some of these policies look simple we were surprised to find that there are many validation functions that do not adhere to them.

```
Email      →  /^[a-zA-Z0-9]+[.a-zA-Z0-9_\-]*@
              [.a-zA-Z0-9_\-]+\.[a-zA-Z]{2,6}$/
Date       →  /^(([0-9]{1,2})|[A-Za-z]{3})[\/\-]
              [0-9]{1,2}[\/\-][0-9]{2}([0-9]{2})?$/
Phone      →  /^(\(?[0-9]{3}\)?)?[\- ]?[0-9]{3}
              [\- ]?[0-9]{4}$/
Time       →  /^[0-9]{1,2}:[0-9]{2}([ap]m)?$/
Zip Code   →  /^[0-9]{5}([. ][0-9]{4})?$/
NotEmpty   →  /^.*[^ \n\t].*$/
```

Figure 2. Maximum input validation policies.

```
Email      →  /^[a-zA-Z0-9]+@[a-zA-Z]+\.[a-zA-Z]{3}$/
Date       →  /^[0-9]{1,2}\/[0-9]{1,2}\/[0-9]{4}$/
Phone      →  /^\([0-9]{3}\) [0-9]{3}-[0-9]{4}$/
Time       →  /^[0-9]{2}:[0-9]{2}$/
Zip Code   →  /^[0-9]{5}$/
NotEmpty   →  /^.*[^ \n\t].*$/
```

Figure 3. Minimum input validation policies.

For example, four out of five validation functions that we found in JavaScript tutorials and textbooks that check for emptiness miss the fact that a form field with only spaces should be considered to be an empty field.

Input validation function verification: If the set of strings accepted by a validation function is not a subset of the *Max* policy or not a superset of the *Min* policy then we consider the validation function to be faulty. In other words, let us assume that the set of strings considered to be valid by a validation function F is $L(F_{true})$, i.e., these are the string values for which the validation function returns true. Let us assume that the language of the *Max* policy that specifies the maximal valid set of inputs for the given field type is $L(P_{max})$ and the language of the *Min* policy that specifies the minimal valid set of inputs for the given field type is $L(P_{min})$. Then the input validation function verification problem is to check if $L(P_{min}) \subseteq L(F_{true}) \subseteq L(P_{max})$.

Since the string analysis is an undecidable problem in general, it is not possible to compute $L(F_{true})$ precisely. However, using our automata-based string analysis approach we can compute an over-approximation $L(F_{true})^+$ such that $L(F_{true}) \subseteq L(F_{true})^+$. Note that, if $L(F_{true})^+ \subseteq L(P_{max})$, then we can be sure that F conforms to the *Max* policy. If $L(F_{true})^+ \not\subseteq L(P_{max})$, on the other hand, we cannot definitely say that F violates the *Max* policy. Since $L(F_{true})^+$ is an over-approximation, we may have a false positive. In this case, in order to figure out if the input validation function is really faulty, we generate a string value $s \in L(F_{true})^+ \setminus L(P_{max})$ and execute the input validation function on that value. If the input validation function returns true for this input, then we are sure that the input validation function violates the *Max* policy and the generated string s serves as a counter-example demonstrating the policy violation.

We cannot use the over-approximation $L(F_{true})^+$ to check conformance to the *Min* policy since $L(P_{min}) \subseteq L(F_{true})^+$ does not imply that $L(P_{min}) \subseteq L(F_{true})$. In order to check conformance to the *Min* policy we need an under-approximation of $L(F_{true})$. However, since our string analysis is a sound analysis technique, it can only generate over-approximations. We solve this problem by using our string analysis to compute an over-approximation of the set of values for which the input validation function F returns false. Let us call this set $L(F_{false})$. Using our string analysis we compute $L(F_{false})^+$ such that $L(F_{false}) \subseteq L(F_{false})^+$. Then, we check if $L(F_{false})^+ \cap L(P_{min}) = \emptyset$. If the intersection of $L(F_{false})$ and $L(P_{min})$ is empty, then we can be sure that the validation function F does not reject any value that is allowed by the *Min* policy, i.e., it conforms to the *Min* policy. If, on the other hand, $L(F_{false})^+ \cap L(P_{min}) \neq \emptyset$, we cannot be sure that F violates the *Min* policy since it can be a false positive. In this case we generate a string $s \in L(F_{false})^+ \cap L(P_{min})$ and execute the input validation function on input s. If the input validation function returns false for this input, then we can be sure that F violates the *Min* policy and s is a counter-example demonstrating this policy violation.

III. Validation Function Extraction

In this section we discuss how we extract a validation function for a given HTML form input field. We start with a brief discussion of validation of HTML forms using JavaScript.

Input validation with JavaScript: The first step in form validation is to register an event handler for some event of the input form or its fields. The event handler is then used to call the JavaScript validation code for some or all of the form fields. Based on the result returned by this validation code, either the form will be submitted or error messages will be shown to the user. Submission of the form data is done either by the browser itself or a JavaScript issued XHR (XmlHttpRequest) request when Ajax is used. The default event for handling form validation code is `onsubmit` event of the form itself. In the basic case, the browser will execute the `onsubmit` event handler (if found) when a user tries to submit an HTML form by clicking on an HTML element of type `submit`. If the handler returns true (or if there is no handler for `onsubmit`) then the browser will submit the form data using an ordinary HTTP get/post request. In this approach all the validation code goes inside the `onsubmit` handler and the functions it calls.

In websites that use Ajax and XHR to submit the forms, the situation is different. First of all, the `onsubmit` handler should return false when the element used to submit the form is of type `submit` (so that the browser does not submit the form itself). Furthermore, since the form will be submitted from within the JavaScript code, the element the user is supposed to click to submit the form does not have to be of type `submit`, and there is a large number of events besides `onsubmit` that can be linked to form submission such as `onclick`, `onmousedown`, and `onmouseup`. Finally, due to the capturing and bubbling of DOM events, it is possible to do the validation in an event handler for one of the events of one of the ancestors of the element used to submit the form. This happens especially when the area the user clicks on to submit the form consists of multiple elements overlaid on top of each other.

Validation function extraction: It is not feasible to statically find and extract the code that does the client-side input validation. First, it is difficult to find the event handlers that contain the form validation code due to the variety and complexity of the input validation process discussed above. Even in the basic case, where the `onsubmit` event handler is used, sometimes this event handler is registered dynamically from within the JavaScript code that loads the page instead of being statically linked in the HTML code of the webpage. Second, even if we succeed in statically locating and extracting the event handling code, the code itself is large and full of event handling, error handling and error message rendering functions which are hard to separate statically. Furthermore the validation code contains all the validation functions for all form fields mixed together instead of having one function per input field.

In our analysis we focus on one form input field at a time and break the verification process into two phases. In the first phase, we extract a dynamic slice [20] from the JavaScript code that is executed upon form submission. This slice represents the validation code for the field we are targeting. It contains all the statements that access the targeted field along with all the other statements they depend on. Then, in the second phase, we statically analyze the extracted slice using string analysis techniques (discussed in Section IV) to check if the validation code conforms to the minimum and maximum validation policies.

As discussed in Section II, in order to verify the input validation function in a sound manner with respect to the maximum and minimum policies, we need to over-approximate both the set of inputs that are accepted by the input validation function and also the set of inputs that are rejected by the input validation function, respectively. We do this by generating, based on our policies, two input values: a valid and an invalid one. We run the program for each input and extract two separate dynamic slices. We record the last control branch that accesses the input value that we provided. If the input value we provided at the beginning is a valid input value, we insert the statement "return true" to the beginning of the branch that the execution takes. If the input value we provided at the beginning is an invalid input value, then we insert the statement "return false." We use the slice extracted using the valid input value to obtain an over-approximation of the values that the input validation function accepts by computing an over-approximation of the input values that reach the statement "return true". We use the slice extracted using the invalid input value to obtain an over-approximation of the values that the input validation function rejects by computing an over-approximation of the input values that reach the statement "return false".

We implemented the dynamic slicing on top of HtmlUnit [6] which is a browser simulator written in Java that uses Rhino [15] JavaScript interpreter. Using HtmlUnit, we simulate the process of filling out a form and submitting it. We provide a profile of values that will be used to fill out the form including one value per each form input and two values - valid and invalid - for the target input field. During this simulation, we instrument the interpreter to track all the JavaScript statements that operate on or test the content of our target field. We then output these statements and all other statements in the execution trace that they depend on. If there are function calls, we inline them such that the final code consists of only one validation function for the target field which ends with either a "return true" statement or a "return false" statement. Since we have instrumented the JavaScript interpreter, we convert all accesses on objects and arrays to accesses on memory locations. This avoids imprecision in our static string analysis phase due to objects, arrays and aliasing.

IV. String Analysis

Given a JavaScript input validation function, we compute an over approximation of string variables' values at each program point using a flow- and path-sensitive, intra-procedural, symbolic string analysis algorithm. The possible values of a string variable at a program point is represented by a deterministic finite automaton (DFA). We use a symbolic automaton representation where the transitions of the automaton are represented as a Multi-terminal Binary Decision Diagram (MBDD).

Lattice: Since JavaScript is not statically typed, it is not possible to statically infer variable types before performing our analysis. Hence, during our string analysis we have to take into account all the variables in the validation function since they can all potentially hold string values. Moreover, we have to take into account that variables can change their types during execution. We use a value lattice that reflects this by initializing all variables in the code to a special value called *uninitialized value*, denoted as $\perp$, which corresponds to the bottom element of the lattice. This special value indicates that the variable type is unknown. As soon as we figure out that a variable is a string variable, we initialize its value to $\emptyset$ (which corresponds to the empty language, i.e., no string value).

If we find out through our analysis that a variable is actually not a string variable, we change its value to the top value of the lattice which corresponds to *unknown value* (and unknown type) and is denoted as $\top$. Just below the top value of the lattice we have Σ^* which corresponds to all possible strings, and this corresponds to the case where we know that a variable is a string variable but we do not know anything about its value, i.e., it could have any string value. If we exclude the top and bottom elements of the lattice (which are introduced to deal with non-string values), the remaining elements form a sub-lattice where $\emptyset$ is the least element and Σ^* is the greatest element, and all the other elements are regular languages over the alphabet Σ.

Algorithm: Our string analysis algorithm (Algorithm 1) starts by receiving the control flow graph of the given validation function along with the validation policies as input. Each node in the CFG represents a statement in the given validation function. In this discussion we will only concentrate on the types of nodes/statements that are crucial for our analysis.

There are two main types of statements we are dealing with in the algorithm which are *Assignment Statement* and *Conditional Statement*. Each statement is associated with two arrays of DFAs: `IN` and `OUT`. Both `IN` and `OUT` have one DFA for each variable and input parameter in the validation function.[1] Given variable v, and the `IN` array for a statement, `IN`$[v]$ is a DFA that accepts all string values that variable v can take at the program point just before the execution of that statement. Similarly, `OUT`$[v]$ is a DFA that accepts all string values that variable v can take at the program point just after the execution of that statement. The `tmp` array is used to store the temporary values (i.e., DFAs) computed by the transfer function before joining these values with the previous ones. As shown in Algorithm 1, the algorithm starts by initializing all the validation function parameter values in the `IN` array of the entry statement to Σ^*. This indicates that the validation function can receive any string value as input. At each program point, we update the DFAs in the `OUT` array based on the DFAs in the `IN` array and the transfer function of the statement at that point. Below we describe the transfer function used to compute the `OUT` array for each of the two basic types of statements mentioned above. Notice that assignment, join, and widening operations on `IN` and `OUT` arrays are carried out as point-wise operations.

Algorithm 1 STRINGANALYSIS(*CFG*,*Policy*$_{\min}$,*Policy*$_{\max}$)

```
initParams();
queue $WQ := NULL$;
$WQ$.enqueue($CFG.entrynode$);
while ($WQ \neq NULL$) do
    $node := WQ$.dequeue();
    $IN := \bigcup_{node' \in PredNodes(node)} OUT_{node'}$;
    if ($node \equiv$ IF $pred$ THEN) then
        $tmp_{on\_T} := tmp_{on\_F} := IN$;
        if ($numOfVars(pred) = 1$) then
            $var := getPredVar(pred)$;
            $predVal :=$ EVALPRED($pred$);
            $tmp_{on\_T}[var] := IN[var] \cap predVal$;
            $tmp_{on\_F}[var] := IN[var] \cap (\Sigma^* - predVal)$;
        end if
        $tmp_{on\_T} := (tmp_{on\_T} \cup OUT_{on\_T}) \nabla OUT_{on\_T}$;
        $tmp_{on\_F} := (tmp_{on\_F} \cup OUT_{on\_F}) \nabla OUT_{on\_F}$;
        if ($tmp_{on\_T} \not\subseteq OUT_{on\_T}$) then
            $OUT_{on\_T} := tmp_{on\_T}$; $OUT_{on\_F} := tmp_{on\_F}$;
            $WQ$.enqueue($Succ(node)$);
        end if
    else
        $tmp := IN$;
        $tmp[var] :=$ EVALEXP($exp, IN$);
        $tmp := (tmp \cup OUT) \nabla OUT$;
        if ($tmp \not\subseteq OUT$) then
            $OUT := tmp$;
            $WQ$.enqueue($Succ(node)$);
        end if
    end if
end while
for ($node \equiv$ RETURN TRUE ) do
    if ($L(OUT_{node}[param1]) \not\subseteq$ POLICY$_{max}$) then
        $s :=$ pick($L(OUT_{node}[param1]) \setminus$ POLICY$_{max}$)
        return "COUNTER-EXAMPLE: $s$"
    else
        return "OK"
    end if
end for
for ($node \equiv$ RETURN FALSE ) do
    if ($L(OUT_{node}[param1]) \cap$ POLICY$_{min} \neq \emptyset$) then
        $s :=$ pick($L(OUT_{node}[param1]) \cap$ POLICY$_{min}$)
        return "COUNTER-EXAMPLE: $s$"
    else
        return "OK"
    end if
end for
```

Exp $\rightarrow$ **replace** (*ptrn, strlit, var*)
| **call** *func(...)* | **concat** (*Exp, Exp*)
| **var** | **strlit** | **numlit** | **boollit** | **null** | **undefined**

Figure 4. The abstract grammar for the right hand side expressions in an assignment statement.

Assignment Statement: In this type of statement a variable on the left hand side is assigned a value of an expression on the right hand side. Figure 4 shows the syntax for the type of expressions on the right hand side that we handle with our analysis. We use the function EVALEXP to compute the set of string values that an expression can take. This function takes two inputs: an expression on the right hand side of an assignment and an `IN` array (which is the `IN` array of the assignment statement where the expression is). It evaluates the expressions as follows:

- var: The set of values for the variable var in the `IN` array (i.e, the DFA `IN`$[var]$) is returned.
- $strlit$: A singleton set that only contains the value of the string literal *strlit* is returned (i.e., a DFA that recognizes only $strlit$).
- $numlit$, $boollit$, $null$: Since our analysis is concerned only with analyzing string values, in these cases we return $\top$ indicating that the value (and type) of the expression is not known.

[1]In our implementation we have a wrapper around the DFA representation in order to represent the bottom $\perp$ and top $\top$ elements, but we will refer to `IN` and `OUT` as DFA arrays to simplify the discussion.

- $undef$: This represents an uninitialized variable so we return $\perp$.
- **concat**$(exp1, exp2)$: Here we compute the concatenation of the regular languages resulting from evaluating $exp1$ and $exp2$ and return it as the result (using the symbolic DFA concatenation operation discussed in [23]).
- **replace**$(ptrn, strlit, var)$: Here we compute the result of replacing all string values in $\texttt{IN}[var]$ that match the pattern $ptrn$ (given as a regular expression) with the string literal $strlit$. There are two types of pattern matching *partial match* and *full match*. The match operation used is chosen based on $ptrn$ value as follows: 1) If the value starts with the symbol ˆ and ends with the symbol $ this means that we have to do a full match where we have to replace a string in $\texttt{IN}[var]$ only if it fully matches the regular expression given by $ptrn$. This is done by taking the difference between the language in $\texttt{IN}[var]$ and the language $L(ptrn)$ and then adding the replace string str to the result. 2) Otherwise we use partial match where we compute the result by using the language-based replacement algorithm described in [23].
- **call** $func(...)$: Since our analysis is intra-procedural we only analyze one function at a time without following any function call. However, for the commonly used functions (such as replace and its variations) we have constructed function models that can be used during our analysis. So, in case of a function call, there are three options: 1) We inline the function if possible. 2) We use the model that we have for this function if it is available. 3) If the first two options are not available then we return Σ^* indicating an unknown string value.

Conditional Statement: This type of statement represents the branch conditions in a number of language constructs in JavaScript including *if statement*, *for loop*, *while loop* and *do while loop*. Conditional statement consists of a predicate on variables and constants. Since it represents a branch in the program, unlike other statements, it is followed by two statements, one after the `ON_TRUE` branch and the other one after the `ON_FALSE` branch.

The predicate in a conditional statement constrains the values of its variables in each of the two branches of execution following the conditional statement. If the predicate evaluates to true, the execution will continue in the `ON_TRUE` branch, otherwise it will take the `ON_FALSE` branch. This behavior is represented in our analysis by having two `OUT` arrays reflecting the possible future values on each of the two branches of execution. $\texttt{OUT}_{on\_T}$ represents the values for the `ON_TRUE` branch and $\texttt{OUT}_{on\_F}$ represents the values for the `ON_FALSE` branch. In order to compute these arrays we first compute the DFA that accepts the set of string values that a variable can take that would make the predicate to evaluate to true (we describe this in detail in the next section). Then we compute the `OUT` array of the `ON_TRUE` branch of the conditional statement by intersecting the `IN` DFA for the variable with the DFA for the set of strings that make the predicate to evaluate to true. On the other hand for the `ON_FALSE` branch we intersect the `IN` DFA with the DFA that is the complement of the DFA that corresponds to the strings that make the predicate to evaluate to true.

Fixed point computation: Algorithm 1 shows our string analysis algorithm that computes the least fixed point that over approximates the possible values that string variables can take at any given program point. At each iteration of the algorithm we compute the transfer function for a statement as described above. After computing the transfer function using the `IN` array, we update the `OUT` array for the current statement using the join (union) and widening operators. The widening operator we use here is taken from [1] and it is used to achieve convergence since the analysis lattice has an infinite height. Briefly, we merge those states in the two input automata belonging to the same equivalence class. Two states are equivalent if the languages accepted starting from the two states are equivalent or both states are reachable from the initial state via the same string.

The analysis converges when the work list becomes empty, which means that reevaluating the transfer functions will not change any of the `OUT` arrays. After the convergence, the `OUT` value for the input parameter ($param1$) at the `return true` statement (which corresponds to an over-approximation of the set of input values that the validation function identifies as valid) is checked against the maximum policy, and the `OUT` value for the input parameter at the `return false` statement (which corresponds to an over-approximation of the set of input values that the validation function identifies as invalid) is checked against the minimum policy. At `return true` statement we verify that all input values that are considered valid by the validation function conform to our maximum policy (i.e., are a subset of the maximum policy). If not, we generate a counter-example string that is not in the maximum policy but is considered to be valid by the input validation function (to compute the counter-example string we implement a function "pick" that returns a string accepted by a given DFA). At `return false` statement we verify that none of the input values that are considered to be invalid by the validation function are a member of the minimum policy. If not, we generate a counter-example string demonstrating the minimum policy violation.

A. Handling Predicates

We only handle the JavaScript predicates that are used for string manipulation. Figure 5 shows the syntax of the predicate language that we handle. Due to space limitations we only list a subset of the predicates that we can handle.

An important point that is not expressed in the abstract syntax is that we only handle predicates on a single variable

```
Pred   →  Pred && Pred  |  Pred || Pred  |  !Pred
       |  var RelOp strlit  |  var.length RelOp intlit
       |  regexp.test(var)  |  var.match(regexp)
       |  var.indexof(strlit) RelOp intlit
RelOp  →  < | <= | > | >= | == | !=
```

Figure 5. The abstract grammar for the branch conditions handled by our analysis.

which means that var in the above syntax must be the same variable throughout the whole predicate. So, each branch condition must be on a single string variable (although, of course, different branch conditions can be on different variables). The reason behind this restriction is mainly the limitations of the DFA representation we are using which can only store a single set of values for each program variable at each program point. Consider the following predicate on two different variables x and y: `x == "foo" || y == "bar"`. On the `ON_TRUE` branch of the condition there are three possible $\texttt{OUT}_{on\_T}$ states. One possibility is to restrict the value of x only. The second one is to restrict the value of y only. The third and last one is to restrict both values. These three possibilities reflect the fact that program execution may take the `ON_TRUE` branch when either `x == "foo"`, `y == "bar"` or `x == "foo"` and `y == "bar"`. Note that we cannot express these three scenarios using only two DFAs (one for x and one for y). Even when we handle a predicate with `and` boolean operator we will have a similar problem when handling the `ON_FALSE` branch of the conditional statement. We can consider using a set of DFAs for each variable, however, this problem gets worse when the conditional statement is part of a loop structure. In this case, for each iteration of the fixed point computation we will add more and more possibilities of `OUT` states and the number will grow exponentially requiring more and more DFAs to represent it. Hence, due to this problem we limit that the predicates should be on a single variable. Interestingly in our experiments we did not encounter any predicate with more that one variable, so this limitation was not significant in practice.

Algorithm 2 shows the algorithm we use to compute the DFA for a given predicate on a certain variable. The algorithm takes a predicate on one variable as input and returns the corresponding DFA (in the algorithm we represent the DFA using the regular expression for its language). The DFA is computed recursively on the given predicate following its recursive structure. Notice that wherever we use the notation Σ^i it means concatenating Σ i times where $\Sigma^0 = \Sigma$. We have omitted the discussion of some of the predicates due to the space limitation.

V. Experiments

The approach presented in this paper can be used both as a forward engineering approach (as an analysis used during the application development) or as a reverse engineering approach (by automatically extracting and analyzing input validation functions after deployment). We evaluated the forward engineering scenario on input validation functions collected from tutorials and books for teaching JavaScript.

Algorithm 2 EVALPRED(*Pred*)

```
if $Pred \equiv Pred1$ && $Pred2$ then
    return EvalPred($Pred1$) $\cap$ EvalPred($Pred2$);
else if $Pred \equiv Pred1$ || $Pred2$ then
    return EvalPred($Pred1$) $\cup$ EvalPred($Pred2$);
else if $Pred \equiv$ !$Pred1$ then
    return $\Sigma^*$ - EvalPred($Pred1$);
else if $Pred \equiv var == strlit$ then
    $retVal := \{strlit\}$;
    return $retVal$;
else if $Pred \equiv var$ != $strlit$ then
    $retVal := \Sigma^* - \{strlit\}$;
    return $retVal$;
else if $Pred \equiv var.length == intlit$ then
    return $\Sigma^{intlit}$;
else if $Pred \equiv var.length > intlit$ then
    return $\Sigma^* - \bigcup_{l=0}^{intlit} \Sigma^l$;
else if $Pred \equiv var.length >= intlit$ then
    return $\Sigma^* - \bigcup_{l=0}^{intlit-1} \Sigma^l$;
else if $Pred \equiv var.length < intlit$ then
    return $\bigcup_{l=0}^{intlit-1} \Sigma^l$;
else if $Pred \equiv var.length <= intlit$ then
    return $\bigcup_{l=0}^{intlit} \Sigma^l$;
else if $Pred \equiv var.length$ != $intlit$ then
    return $\Sigma^* - \Sigma^{intlit}$;
else if $Pred \equiv regexp.test(var)$ | $var.match(regexp)$ then
    if check_regexp($regexp$) = partial_match then
        return CONCAT(CONCAT($\Sigma^*$, $L(regexp)$), $\Sigma^*$);
    else
        return $L(regexp)$;
    end if
else if $Pred \equiv var.indexof(strlit) == intlit$ then
    if $intlit$ = -1 then
        return $(\Sigma - \{strlit[0]\})^*$;
    else if $intlit \geq 0$ then
        return CONCAT($\Sigma^{intlit-1}$, CONCAT ($\{strlit[0]\}$ , $\Sigma^*$));
    end if
else if $Pred \equiv var.indexof(strlit) >= intlit$ then
    if $intlit$ = -1 then
        return CONCAT( $\Sigma^*$, CONCAT($\{strlit[0]\}$, $\Sigma^*$)));
    else
        return CONCAT($(\Sigma - \{strlit[0]\})^{intlit}$, CONCAT( $\Sigma^*$, CONCAT($\{strlit[0]\}$, $\Sigma^*$)));
    end if
else if $Pred \equiv var.indexof(strlit) > intlit$ then
    return CONCAT($(\Sigma - \{strlit[0]\})^{intlit+1}$, CONCAT( $\Sigma^*$, CONCAT($\{strlit[0]\}$, $\Sigma^*$)));
else if $Pred \equiv var.indexof(strlit) <= intlit$ then
    if $intlit$ = -1 then
        return $(\Sigma - \{strlit[0]\})^*$;
    else
        return $\bigcup_{l=0}^{intlit}$ CONCAT($\Sigma^l$, CONCAT($\{strlit[0]\}$, $\Sigma^*$));
    end if
else if $Pred \equiv var.indexof(strlit) < intlit$ then
    if $intlit$ = 0 or $intlit$ = -1 then
        return $(\Sigma - \{strlit[0]\})^*$;
    else
        return $\bigcup_{l=0}^{intlit-1}$ CONCAT($\Sigma^l$, CONCAT($\{strlit[0]\}$, $\Sigma^*$));
    end if
else if $Pred \equiv var.indexof(strlit)$ != $intlit$ then
    if $intlit$ = -1 then
        return CONCAT($\Sigma^*$, CONCAT($\{strlit[0]\}$, $\Sigma^*$));
    else if $intlit$ = 0 then
        return CONCAT($\Sigma - \{strlit[0]\}$, $\Sigma^*$);
    else
        return CONCAT($\Sigma^{intlit-1}$, CONCAT($\Sigma - \{strlit[0]\}$, $\Sigma^*$));
    end if
else
    return $\Sigma^*$;
end if
```

We evaluated the reverse engineering scenario on several real-world applications by extracting and analyzing their input validation functions. In our experiments we used a MacBook Pro with a 2.53 GHz core 2 due processor and 4 GB of memory.

A. Verifying Stand-Alone Input Validation Functions

In this section we show the results of verifying 23 JavaScript validation functions collected from a JavaScript book [14] and several JavaScript input validation tutorials on the internet. The book on JavaScript and the tutorials represent a set of validation functions that should have been written very carefully given that these are examples used for teaching JavaScript programming. These functions also include a wide variety of string operations and predicates that one expects to see in a JavaScript application and hence form a good benchmark for evaluating the effectiveness of our string analysis techniques. The functions we analyzed cover all the policies mentioned in Section II. Five of the validation functions were changed slightly to conform to our assumptions for validation functions, so that the modified function return `true`/`false` values instead of empty/nonempty error messages in case of acceptance/rejection of input.

Results. The total time it took for analyzing the 23 functions was 3.05 seconds during which 488 lines of code have been analyzed. Table I shows the individual results for each validation function using our analysis. The first column is the validation function name. The second one is the source for this function. The third column shows the type of data that is validated by this validation function which is also the type of policy used to verify the function itself. Columns four and five show the performance of our string analysis in terms of time and memory. Last column shows the result for verification against both the maximum and the minimum policies. Failing the maximum policy means that the function accepts some values that are invalid according to our policy. For example, the first function does not satisfy the maximum policy which means that it allows email addresses that are considered to be invalid by our policy. On the other hand failing the minimum policy means that the function rejects some values that are correct according to our policy. For example, function number 13 does not satisfy the minimum policy which means that it rejects some time inputs that we consider to be correct.

Discussion. Among the 23 functions we have analyzed, 10 of them violated a maximum policy while 3 of them violated a minimum policy. We tested these results using the counter-examples generated by our analysis and did not find any false positives due to over-approximation. It is interesting to see that all the functions that validate the email addresses failed to comply with our maximum email policy (which is the most complicated policy we used). It is even more interesting to see that four of the five functions that validate non emptiness failed to comply with our maximum non emptiness policy although this check seems very simple.

The most subtle error in email validation functions is the one that we discussed at the beginning of section II where the developer forgot to escape the dash character inside a character class. This results in accepting email addresses with invalid characters such as "[". Another problem that we have found is the usage of a black list to block invalid characters in an email address rather than a white list where only valid characters are accepted. All of these black lists miss at least one invalid character. We think that the white list approach that we used in our policy is much simpler and less error prone.

The phone validation function number 16 failed to comply with our minimum policy because it rejects a phone number that has two parentheses around its area code. This is the most common format to write a phone number in US and, hence, in our minimum policy we specify that this should be accepted as valid input.

The most surprising result is the errors in non emptiness checks. The reason behind the four failures is that these four functions accept a field value that only consists of white space characters while such value should be considered empty. Such a validation error will likely cause an unnecessary interaction with the web server, and if the same validation check is also erroneous at the server side it can lead to fatal errors in the application.

Our results demonstrate that 1) writing input validation checks in JavaScript is an error-prone task and even the sample validation functions shown in tutorials and books on JavaScript contain errors. 2) Using the string analysis techniques we presented, we can efficiently check the conformance of a JavaScript input validation function to a given input validation policy.

B. Verifying Input Validation Operations in Deployed Web Applications

We applied our verification technique to a number of real world websites that use JavaScript to validate their HTML form inputs. For each of these websites we pick an HTML form, fill it out, and submit it. During submission we automatically extract the validation code for one target field and analyze this code statically to see if it violates our predetermined policies for the type of that field. We applied our technique on fields with common input format such as email, phone number, etc. Our analysis can also be applied to other fields that need specific policies chosen by the organization running the website such as username and password fields.

Results. Table II shows the results of applying our analysis to a number of websites. Each row represents the results for extracting and verifying the validation code for a single input field in a form in the given website. Since this is done twice for a valid and invalid input we show two sub-columns for

Table I
RESULTS OF OUR ANALYSIS ON INPUT VALIDATION FUNCTIONS COLLECTED FROM JAVASCRIPT BOOKS AND TUTORIALS

	FuncName	Source	Type	Time (seconds)	Memory (MB)	Result	
						Max Policy	Min Policy
1	validateEmail	Book - HeadFirst JavaScript	Email	0.640	9.6	X	√
2	validateEmail	Tut - http://www.webcheatsheet.com/	Email	0.150	24.3	X	√
3	emailValidator	Tut - http://www.tizag.com/	Email	0.140	44.7	X	√
4	checkEmail	Tut - http://developer.apple.com/	Email	0.560	8.3	X	√
5	isEmailAddress	Tut - http://www.devshed.com/	Email	0.140	19.9	X	X
6	isAlphabet	Tut - http://www.tizag.com/	Alphabet	0.014	24.1	√	√
7	isAlphabetic	Tut - http://www.devshed.com/	Alphabet	0.014	25.0	√	√
8	isAlphanumeric	Tut - http://www.tizag.com/	AlphaNumeric	0.018	26.1	√	√
9	isAlphaNumeric	Tut - http://www.devshed.com/	AlphaNumeric	0.009	27.2	√	√
10	isNumeric	Tut - http://www.tizag.com/	Numeric	0.013	28.0	√	√
11	isNumber	Tut - http://www.devshed.com/	Numeric	0.011	28.7	√	√
12	validateDate	Tut - http://www.the-art-of-web.com/	Date	0.041	32.0	√	√
13	validateDate	Book - HeadFirst JavaScript	Date	0.507	5.2	√	X
14	validatePhone	Tut - http://www.webcheatsheet.com/	Phone	0.075	15.6	√	√
15	checkPhone	Tut - http://developer.apple.com/	Phone	0.058	27.4	√	√
16	validatePhone	Book - HeadFirst JavaScript	Phone	0.076	37.5	X	X
17	validateTime	Tut - http://www.the-art-of-web.com/	Time	0.031	43.1	√	√
18	validateZipCode	Book - HeadFirst JavaScript	ZipCode	0.040	48.1	√	√
19	validateEmpty	Tut - http://www.webcheatsheet.com/	NotEmpty	0.448	0.8	X	√
20	notEmpty	Tut - http://www.tizag.com/	NotEmpty	0.013	1.8	X	√
21	isEmpty	Tut - http://developer.apple.com/	NotEmpty	0.017	2.9	X	√
22	isEmpty	Tut - http://www.devshed.com/	NotEmpty	0.014	4.2	√	√
23	validateNonEmpty	Book - HeadFirst JavaScript	NotEmpty	0.021	5.9	X	√

Table II
RESULTS OF OUR ANALYSIS ON DEPLOYED WEBSITES

	Source	Type	Code Size (LOC)		Execution Trace Size (LOC)		Extraction Time (s)		Analysis Time (s)		Result	
			Valid	Invalid	Valid	Invalid	Valid	Invalid	Valid	Invalid	Max	Min
1	www.google.com	Email	407	407	103	138	17.82	13.42	0.57	0.47	X	√
2	www.bloomberg.com	Email	2138	2133	776	769	14.85	12.84	0.48	0.44	X	√
3	www.bloomberg.com	Phone Number	2138	2138	808	808	18.81	16.92	0.45	0.44	X	√
4	www.stc.com.sa	Email	124	102	93	62	67.88	54.23	0.65	0.50	X	X
5	www.kooora.com	Email	35	35	10	11	10.50	7.67	0.60	0.45	X	√
6	www.multiply.com	Email	171	171	230	271	8.74	9.22	0.50	0.47	X	√
7	www.acm.com	Email	1644	1644	867	871	10.19	11.41	0.61	0.43	X	X
8	www.netflix.com	Email	518	518	2411	2411	24.27	24.27	-	-	X	√
9	www.btjunkie.org	Email	104	104	20	21	12.16	12.01	0.50	0.52	X	√
10	www.pcmag.com	Email	514	514	31808	31808	32.13	32.13	-	-	X	√
11	www.pcmag.com	Zip Code	570	570	31850	31850	33.48	31.81	0.33	0.43	X	√
12	www.apple.com	Email	1925	1925	1783	1783	14.15	13.95	0.51	0.42	X	√
13	www.acm.com	Zip Code	1666	1644	1052	1047	12.70	10.42	0.49	0.66	X	√

each. The first column is name of website where we got the form from. The second column shows the type of data that is validated by this validation function which is also the type of policy used to verify the function itself. Column three shows the size (in lines of code) of the form submission handling code including the validation code from which we extracted the validation function. Column four shows the number of lines of code that has been executed as part of submitting the form. Column five shows the time it takes to dynamically extract the validation function while column six shows the time to statically analyze the extracted function. In two cases there was no validation code in the application and the extracted validation function was empty. Hence, there was no string analysis done for these two cases and the corresponding column is left empty. Last column shows the result for verification against both the maximum and the minimum policies where an X means a policy violation.

Policy Violations. We have found a policy violation in each of the websites that we tested. Some of these policy violations are a result of subtle bugs in the validation code, some of them are a result of writing light weight validation code or even not writing any, and some of them are due to minor differences between our policies and websites' policies.

Two subtle bugs that we have found are in 7 and 4. In 7, a condition in the validation code was supposed to reject any email that ends with "**@csta.acm.org**". This was written as `if(ckEmailVal.match("@csta.acm.org")){..}` The programmer forgot to escape the dot as a special character in the regular expression so when JavaScript converts this string into a regular expression, it will interpret dot as any character. Our analyzer output "**A@cstaAacm.org**" as an example for an email that should not be rejected. We changed our minimum policy to reflect the website developers intention and still got a policy violation with the same example.

In 4 (a website for a large telecom company) the developers claim that they follow the RFC standard for email addresses. We found that they disallowed capital characters from emails with no obvious reason and our analyzer gave the following example that should not be rejected "**A@A.AAA**".

Some of the other websites have lightweight validation code that will accept incorrect input. For example, 2 only checks for presence of '@' and '.' in an email and our analyzer gave "**.@n**" as an invalid email that is accepted. 1 accepts any email that does not have space, ' or " in it. Our analyzer gave "**0x1f@0x1f.0x1f**" as an invalid email that is accepted. This latter example was randomly generated and happened to be not printable but for this case there are counter examples with printable characters. Finally, two websites 8 and 10, had no validation code at all and our slicer returned an empty validation function, meaning that all input values are accepted. In this corner case there is no need to run the string analysis and we only report a maximum policy violation.

VI. Related work

Client side input validation is an important problem that has been studied before. FLAX [17] uses dynamic analysis techniques to discover client side validation vulnerabilities. The authors use dynamic taint analysis to extract validation code related to a certain sink and then use random fuzzing to test this sink. In our technique we use a similar approach to extract the validation function but then we statically analyze the extracted code to see if it violates the given policies.

In [16] authors developed a symbolic execution framework for JavaScript. At the core of their framework there is a string constraint solver called KUDZU that is built on top of the bounded string solver HAMPI [11]. Their approach is able to handle a larger set of string operations and predicates in JavaScript compared to our approach. However, their approach bounds the lengths of the execution paths (by bounding loops) and uses a bounded string solver whereas our approach handles unbounded paths (using widening) and handles unbounded strings (using automata). For the verification problem we are addressing, a bounded string solver can only find policy violations but it can not prove the conformance to a given policy. Our static string analyzer (i.e., the second phase of our analysis) is sound (with respect to the restricted set of JavaScript string operations and predicates that we can handle) and can prove that a validation function conforms to a given policy.

NoTamper [2] uses dynamic symbolic execution to discover constraints on HTML form inputs at the client-side. Then, it uses these constraints to generate input values to test the server-side input validation. In contrast, we are focusing on finding input validation errors at the client-side with respect to a given policy.

MiTV [19] uses dynamic symbolic execution engine Pex [12] to test the correctness of user input validation functions for .NET web applications. These functions are first classified according to the type of input they validate. Then each validation function is tested by comparing it to a subset of the functions under the same class. As we have seen in our experiments it is possible for many or even all functions in a specific class to fail to correctly validate a user input. So we believe that it is worthwhile to develop validation policies and then use these policies as a reference for verification of different implementations.

GATEKEEPER [9] uses static analysis to verify the enforcement of security policies written in Datalog on JavaScript widgets. These policies are different than ours and they are not related to input validation.

Due to its importance for establishing dependability of web applications, 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 [3] which has also been used to check for various types of errors in the server side of Web applications [8], [13], [21], [10]. In [13], [21], multi-track DFAs, also known as *transducers*, are used to model replacement operations.

DFA based symbolic string analysis has been used to verify the correctness of string sanitization operations in PHP programs [23], [22]. Recently, foundations of relational string analysis using multi-track automata (as opposed to single-track automata used in our analysis) were investigated in [24]. In the future we plan to investigate integration of relational string analysis to our JavaScript string analyzer which would allow us to analyze branch conditions on multiple variables. Another future research direction would be automatically synthesizing fixes to validation functions that violate a given policy using techniques similar to the vulnerability patching techniques presented in [4].

There are also several other string analysis tools that use symbolic string analysis based on DFA encodings [18], [5], [23]. Some of them are based on symbolic execution and use a DFA representation to model and verify the string manipulation operations in Java programs [18], [5].

VII. Conclusions

We have presented an approach for verifying client-side input validation functions. Given maximum and minimum policies identifying the largest and smallest set of input values that should be accepted as valid, our analysis verifies if a given website conforms to the given policies by, first, extracting an input validation function from the given website using dynamic slicing, and then, checking the conformance of the extracted function to the given policies using automata-based static string analysis. When there is a policy violation, our analysis generates a counter-example string demonstrating the violation. We applied our analysis to a number of validation functions and websites and our results indicate that it can effectively find subtle errors in client-side input validation code.

References

[1] C. Bartzis and T. Bultan. Widening arithmetic automata. In *In Computer Aided Verification04*, pages 321–333, 2004.

[2] 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.

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

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

[5] 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.

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

[7] Google Labs. Closure Compiler. http://code.google.com/closure/compiler/.

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

[9] S. Guarnieri and B. Livshits. Gatekeeper: mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[10] G.Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 171–180, 2008.

[11] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, 2009.

[12] Microsoft Research. Pex. http://research.microsoft.com/en-us/projects/pex/.

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

[14] M. Morrison. *Head First JavaScript*. O'Reilly Media, 2007.

[15] Mozilla Foundation. Rhino: Javascript for Java. http://www.mozilla.org/rhino/.

[16] 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.

[17] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

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

[19] K. Taneja, N. Li, M. R. Marri, T. Xie, and N. Tillmann. Mitv: multiple-implementation testing of user-input validators for web applications. In *ASE*, pages 131–134, 2010.

[20] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

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

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

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

[24] F. Yu, T. Bultan, and O. H. Ibarra. Relational string verification using multi-track automata. In *CIAA*, pages 290–299, 2010.