ViewPoints: Differential String Analysis for Discovering Client- and Server-Side Input Validation Inconsistencies

Muath Alkhalaf*, Shauvik Roy Choudhary†, Mattia Fazzini†
Tevfik Bultan*, Alessandro Orso†, Christopher Kruegel*

*Department of Computer Science
University of California
Santa Barbara, CA
{muath | bultan | chris}@cs.ucsb.edu

†College of Computing
Georgia Institute of Technology
Atlanta, GA
{shauvik | mfazzini | orso}@cc.gatech.edu

ABSTRACT

Since web applications are easily accessible, and often store a large amount of sensitive user information, they are a common target for attackers. In particular, attacks that focus on *input validation vulnerabilities* are extremely effective and dangerous. To address this problem, we developed ViewPoints—a technique that can identify erroneous or insufficient validation and sanitization of the user inputs by automatically discovering inconsistencies between client- and server-side input validation functions. Developers typically perform redundant input validation in both the front-end (client) and the back-end (server) components of a web application. Client-side validation is used to improve the responsiveness of the application, as it allows for responding without communicating with the server, whereas server-side validation is necessary for security reasons, as malicious users can easily circumvent client-side checks. ViewPoints (1) automatically extracts client- and server-side input validation functions, (2) models them as deterministic finite automata (DFAs), and (3) compares client- and server-side DFAs to identify and report the inconsistencies between the two sets of checks. Our initial evaluation of the technique is promising: when applied to a set of real-world web applications, ViewPoints was able to automatically identify a large number of inconsistencies in their input validation functions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods*; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Web security, input validation, differential string analysis, web testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA '12, July 15-20, 2012, Minneapolis, MN, USA
Copyright 12 ACM 978-1-4503-1454-1/12/07 ...$10.00.

1. INTRODUCTION

Web applications are important and increasingly widespread. In fact, nowadays, it is common to use web applications for daily activities such as reading the news, doing online banking, and watching movies. Together with the growth of their user base, the complexity of web applications has also grown. Whereas early web applications were mostly static, today's applications are highly dynamic, with complex logic on both the client and the server sides. In general, web applications follow a three-tier architecture that consists of a front-end component (typically, a web browser running on the user machine), a back-end component (a remote web server), and a back-end data store (a database).

Because of their nature, web applications present a specific set of security challenges. First, unlike traditional desktop applications, web applications can be accessed by any user who is connected to the Internet, which exposes a web application to a large base of potential attackers. Second, the back-end database contains data that is often sensitive and confidential, holding information about a potentially large number of users. Finally, the communication between the different layers of a web application occurs through directives (commands) that often embed user input and are written in many languages, such as XML, SQL, and HTML. For these reasons, it is of paramount importance to properly validate and sanitize user input, to make sure that malicious input cannot result in unintended execution of harmful commands and, as a consequence, provide attackers with access to sensitive information. Unfortunately, it is not uncommon for developers to perform either faulty or incomplete input checks, which can leave the web application susceptible to *input validation vulnerabilities*. Popular examples of attacks that leverage such vulnerabilities are interpreter injection (e.g., cross-site scripting (XSS)), locale and unicode attacks, file system attacks, and buffer overflows, which are among the most common and dangerous attacks for web applications [21,22]. There are also attacks that exploit application-specific vulnerabilities and that are even harder to identify because they depend on the particular input validation logic performed on the target application.

One way to make sure that the input validation performed by a web application is adequate is to find a characterization of the potential attacks. It is often possible, for instance, to encode well-known attacks in the form of attack patterns and use these patterns to suitably sanitize inputs. In other cases, however, the checks to be performed on the inputs are specific to the functionality of the web application, and the input validation may be tightly coupled with and dependent on the application logic. Because they are specific to individual applications, there are no pre-defined patterns that can be used to assess these types of input checks. In these cases, to make

sure that the input validation is adequate, it would be necessary to produce a different specification for each different application, which is a tedious and error-prone task.

To address this problem, we present ViewPoints, a novel approach for automatically identifying input validation issues in web applications. ViewPoints is based on the observation that developers often introduce redundant checks both in the front-end (client) and the back-end (server) component of a web application. Client-side checks are mainly introduced for performance reasons, as they can save one network round-trip and the additional server-side processing that would be incurred when invalid input is sent to and subsequently rejected by the web application. Therefore, to improve the user experience and provide instant feedback, many web applications validate inputs at the client side before making the actual request to the server. On the other hand, since client-side validation can often be circumvented by malicious users, the server cannot trust the inputs coming from the client side, and all input checks performed on the client side must be repeated on the server side before user input is processed and possibly passed to security sensitive functions.

The key insight behind ViewPoints is that, because checks performed at the client and server sides should enforce the same set of constraints on the inputs, we can leverage the redundancy in these checks to automatically identify issues with input validation. If client-side checks are more restrictive, the web application may accept inputs that legitimate clients can never produce, which is problematic because malicious users can bypass client-side checks. If server-side checks are more restrictive, conversely, the client may produce requests that are subsequently rejected by the server, which will result in poor performance and reduce the responsiveness of the web application. Moreover, correctness of the client-side input validation functions is also important for addressing the emerging class of client-side input-validation vulnerabilities [19]. Based on this insight, ViewPoints compares the checks performed on client and server sides against each other and identifies and reports inconsistencies between them. More precisely, the ViewPoints technique consists of three main steps:

Extracting and mapping input validation operations at the client and server sides. ViewPoints automatically extracts client- and server-side input validation functions and maps corresponding input validation operations on the client and on the server.

Modeling input validation functions as deterministic finite automata (DFAs). After extracting and mapping them, ViewPoints uses string analysis techniques to model client- and server-side input validation functions as DFAs that encode the set of constraints that the functions impose on the inputs.

Identifying and reporting inconsistencies in corresponding input validation functions. ViewPoints compares the client- and server-side DFA models of corresponding inputs to identify inconsistencies between them, that is, checks that are performed only on one of the two sides. It then reports checks performed only on the client as potential server-side vulnerabilities, and checks performed only on the server as opportunities for improving the efficiency of the web application and as potential client-side vulnerabilities.

To evaluate the effectiveness and practicality of our approach, we implemented a prototype of ViewPoints that can analyze web applications developed using Java EE frameworks and used the prototype on seven real-world web applications. The results of our evaluation, albeit preliminary, are promising and motivate further research. Overall, ViewPoints was able to identify inconsistencies in five out of the seven applications considered. More precisely, ViewPoints identified 2 server-side and over 200 client-side input-validation inconsistencies. Additionally, Viewpoints was able to generate counterexamples for the inconsistencies it identified, which guarantees that the issues found are true positives, that is, they correspond to actual problems in the applications.

The main contributions of this paper are:

- A novel technique, ViewPoints, that leverages redundant client- and server-side input checks in web applications to automatically identify, through differential automata-based string analysis, security and efficiency issues in input validation code.
- An implementation of the ViewPoints approach that can be used on web applications based on Java EE frameworks.
- An empirical study performed on several real-world web applications that provides initial, yet clear evidence that ViewPoints is able to identify actual input validation problems in real code effectively and efficiently.

The rest of the paper is organized as follows. Section 2 presents a motivating example for our work. Section 3 illustrates our approach. Section 4 discusses our current implementation of ViewPoints. Section 5 presents our empirical results. Section 6 discusses related work. Finally, Section 7 provides concluding remarks and sketches future research directions.

2. MOTIVATING EXAMPLE

To motivate our work, we use an example taken from a real web application called JGOSSIP (`http://sourceforge.net/projects/jgossipforum/`), a message board written using Java technology. Figure 1 provides a high-level, intuitive view of the web application, for illustration purposes. The parts of the web application shown allow users to register their email address by entering it into a form and submitting it to a web back-end hosted on a server at `site.com`.

Typically, in order to access functionality on the server side, the client side issues an HTTP request that contains a set of input elements expressed as $\langle name, value\rangle$ pairs. When this happens, the different input elements are marshaled (i.e.,packaged together), and the resulting bundle is passed to the server-side code as an input. The server code then accesses these inputs by name by invoking library functions provided by the language or framework used. These functions, which we call *input functions*, parse the HTTP request containing the inputs and return the values of the requested input. The server then processes the retrieved input values, normally performs some form of input validation, and then uses them in possibly critical or sensitive functions.

To illustrate, Figures 2 and 3 show two snippets of client- and server-side validation code, respectively, from JGOSSIP (we slightly simplified the code to make it more readable and self-contained). The user fills the client-side form, shown on lines 18–22 of Figure 2, by providing an email address to the HTML input element with name "email" and by clicking on the "Submit" button. When this button is clicked, the browser invokes the JavaScript function `validateEmail`, which is assigned to the `onsubmit` event of the form. This function first fetches the email address supplied by the user from the corresponding form field. It then checks if this address has zero length and, if so, accepts the empty address on line 6. Otherwise, on lines 9 and 10, the function creates two regular expressions. The first one specifies three patterns that the email address should not match: a single space character, a string with the `@` symbol on both ends, and the string "`@.`". The second one

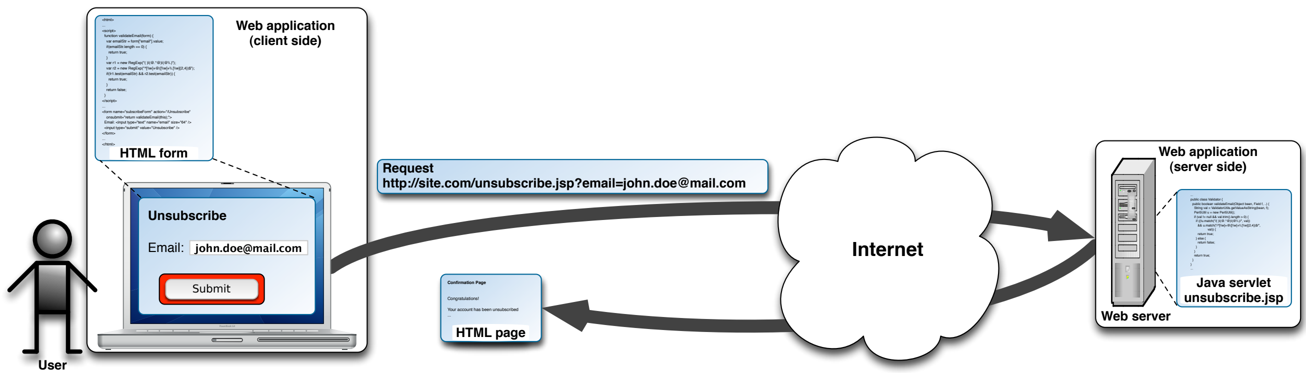

Figure 1: Example web application.

```
1  <html>
2  ...
3  <script>
4  function validateEmail(form) {
5    var emailStr = form["email"].value;
6    if(emailStr.length == 0) {
7      return true;
8    }
9    var r1 = new RegExp("( )|(@.*@)|(@\\.)");
10   var r2 = new RegExp("^[\\w]+@([\\w]+\\.[\\w]{2,4})$");
11   if(!r1.test(emailStr) && r2.test(emailStr)) {
12     return true;
13   }
14   return false;
15 }
16 </script>
17 ...
18 <form name="subscribeForm" action="/Unsubscribe"
19     onsubmit="return validateEmail(this);">
20   Email: <input type="text" name="email" size="64" />
21   <input type="submit" value="Unsubscribe" />
22 </form>
23 ...
24 </html>
```

Figure 2: JavaScript and HTML code snippets for client-side validation.

```
1  public class Validator {
2    public boolean validateEmail(Object bean, Field f, ..) {
3      String val = ValidatorUtils.getValueAsString(bean, f);
4      Perl5Util u = new Perl5Util();
5      if (!(val == null || val.trim().length == 0)) {
6        if ((!u.match("/( )|(@.*@)|(@\\.)/", val))
7          && u.match("/^[\\w]+@([\\w]+\\.[\\w]{2,4})$/",
8                     val)) {
9          return true;
10       } else {
11         return false;
12       }
13     }
14     return true;
15   }
16 ...
17 }
```

Figure 3: Java server-side validation code snippet.

specifies a pattern that the email address should match: start with a set of alphanumeric characters, followed by symbol @, further followed by another set of alphanumeric characters, and finally terminated by a dot followed by two to four additional alphanumeric characters. If the email address does not match the first regular expression and matches the second one, this function returns true, indicating acceptance of the email address (line 12), and the form data is sent to the server. Otherwise, the function rejects the email address by returning false on line 14. This results in an alert message to inform the user that the email provided is invalid.

When the form data is received by the server, it is first passed to the server-side validation function. For the specific form in this example, the validation function used is method `validateEmail` from class `Validator`, which is shown in Figure 3. This method calls a routine on line 3 to extract the value contained in the email field from the form object (`bean`) and stores it in variable `val`. It then uses library `Perl5Util` to perform the regular expression match operations, which allows for using the same Perl style regular expression syntax used in the client. First, the method checks whether the email string is `null` or has zero length after applying the `trim` function, on line 5. If so, it accepts the string. Otherwise, it checks the address using the same regular expressions used on the client side. As shown on lines 6–12, the address is accepted if it satisfies these regular expression checks, and it is used for further processing on the server side (e.g., it may be sent as a query string to the database); otherwise, it is rejected on the server side, and the user is taken back to the form.

As shown in this example, the regular expression checks are similar on both ends, which emphasizes that validations on both ends should allow or reject the same set of inputs. Otherwise, there would be mismatches that may create problems for the application. As we stated in the Introduction, if the server side is less strict than the client side, this would be considered a vulnerability (even when such a vulnerability is not exploitable) since it violates a common security policy that server-side checks should not be weaker than the client-side checks: a malicious user could bypass the client-side checks and submit to the server an address that does not comply with the required format, which may result in an attack. For example, an attacker could inject SQL code in the email that may result in an SQL injection attack [11]. In general, server-side checks that are less strict than the client-side checks could lead to two types of undesirable behaviors: (1) the server side allows some wrong or malicious data to enter the system, leading to failures or attacks; (2) the client side rejects legitimate values that should be accepted, resulting in the user being unable to access some of the functionality provided by the web application.

In our example, the client-side validation code shown in Figure 2 rejects a sequence of one of more white space characters (e.g.," "),

for which the condition on line 6 evaluates to false and the regular expression check on line 11 fails, thereby resulting in the function returning false. However, for the same input, the second condition on line 5 of the server-side validation method (Figure 3) evaluates to false, due to the `trim` function call, and the string is therefore accepted by the server. This would lead to white spaces being accepted as email addresses by the server, which might in turn lead to failures (e.g., the web application might try to send an email to the user, which would fail due to an invalid email address) or attacks, such as a denial-of-service attack.

In the following sections, we show how our ViewPoints approach can automatically identify such inconsistencies, thus detecting and preventing this kind of issues.

3. OUR APPROACH: DIFFERENTIAL STRING ANALYSIS

As we discussed in the Introduction, the basic intuition behind ViewPoints is that we can leverage redundant checks in the client- and server-side components of web applications to detect inconsistencies between the two that may lead to security vulnerabilities, inefficiencies, and general misbehaviors. More precisely, we use the checks performed by the client (resp., server) as a specification to verify the checks performed by the server (resp., client).

Server-side input checks should never be less strict than their client-side counterparts, as client-side checks can be bypassed by malicious users. In fact, less strict server-side checks may lead to common web application vulnerabilities that the Open Web Application Security Project (OWASP) considers to be of *very high severity* and with a *very high likelihood of being exploited* [23].

Obviously, it is also possible for client-side checks to be less strict than server-side checks. This can potentially lead to client-side vulnerabilities [19]. Moreover, missing checks on the client side may result in unnecessary network communications and server-side computation; adding such checks can therefore improve the responsiveness and performance of the overall web application.

To identify input-validation inconsistencies between the client and the server, ViewPoints (1) extracts and maps input validation operations at the client and server sides, (2) models input validation functions as deterministic finite automata (DFAs), and (3) identifies and reports inconsistencies in corresponding input validation functions. Figure 4 provides a high-level view of the three steps of our approach. In the following sections, we describe these three steps in detail.

3.1 Input Validation Extraction

The goals of this step are to (1) automatically extract the snippets of input validation code from a web application, on both the client side and the server side, and (2) map client- and server-side input validation operations for the same input to each other. Because web applications can be developed using a number of languages and technologies, and both extraction and mapping highly depend on the specific languages and technologies used, it is not possible to devise a general technique for this step that would work for all applications. Therefore, in the input validation extraction and mapping step of ViewPoints, we focus on a specific class of web applications.

As far as languages are concerned, we target web applications that use Java on the server side and JavaScript on the client side, as these are among the most commonly used languages for the development of web applications. As for technologies, we focus on web applications built using any Java Enterprise Edition (EE) web framework, such as Google Web Toolkit (`http://code.google.com/webtoolkit/`), Struts (`http://struts.apache.org/`), and Spring MVC (`http://www.springsource.org/`). We focused on these technologies because they are the most popular application development frameworks for web applications.

3.1.1 Input Validation Identification

ViewPoints starts the extraction process by identifying entry points of the web application, that is, points where user input is read. At the client side, such points correspond to input fields in web forms. Although in theory ViewPoints would need to analyze all dynamically generated web pages and corresponding web forms to do that, modern web application frameworks let developers specify in a configuration file the input fields of a web application, together with the JavaScript validation functions to be applied to each field. By leveraging this information, Viewpoints can identify (1) the complete set of validated inputs on the client side, and (2) the corresponding set of JavaScript functions that are used for validating such inputs.

The identification of the input validation code on the server side is analogous to that of the client side, with the difference that validation is performed using a different language—Java instead of JavaScript—and that parameters are read through calls to input functions, as we discussed in Section 2. This second difference is almost irrelevant because, analogously to the client side, web application frameworks also allow developers to specify server side inputs and corresponding validation functions. Also for the server side, therefore, an analysis of the web application's configuration file can provide ViewPoints with (1) the complete set of validated inputs for the server side and (2) the set of Java functions that are used for validating each of these inputs.

It is worth noting that web applications could also perform input validation checks directly in the code, without explicitly specifying inputs and corresponding validating functions in a configuration file. However, we did not find any occurrence of this situation in the web applications we analyzed. Moreover, if these cases were found to be relevant, it would always be possible to perform, in addition to the configuration file analysis that we are currently performing, an analysis of the dynamically generated JavaScript code on the client side and an analysis of the Java code on the server side, as we did in previous work [1, 9].

3.1.2 Input Validation Analysis

After identifying the input validation functions for each input on the server side and client side, ViewPoints analyzes these functions to build a summary validation function for each input. Intuitively, given an input i, its *summary validation function* consists of the concatenation of all the relevant statements in all the validation functions that are applied to i. More precisely, ViewPoints identifies, in the validation functions for i, all and only the statements that operate on i, directly or indirectly, such as string manipulation operations on i and conditional statements affected by i's value. The rest of the code is disregarded because it is irrelevant for the validation of i. Summary validation functions are built differently for the client side and the server side.

On the *client side*, input validation is performed using JavaScript code, which is notoriously difficult to analyze statically due to its highly dynamic and loosely-typed nature [17]. Therefore, ViewPoints extracts the relevant client-side input validation code using dynamic slicing [24].

Specifically, ViewPoints (1) executes, in sequence, all of the validation functions associated with i and (2) collects the traces produced by the resulting executions. ViewPoints performs these steps several times using different values for i chosen from a pool of

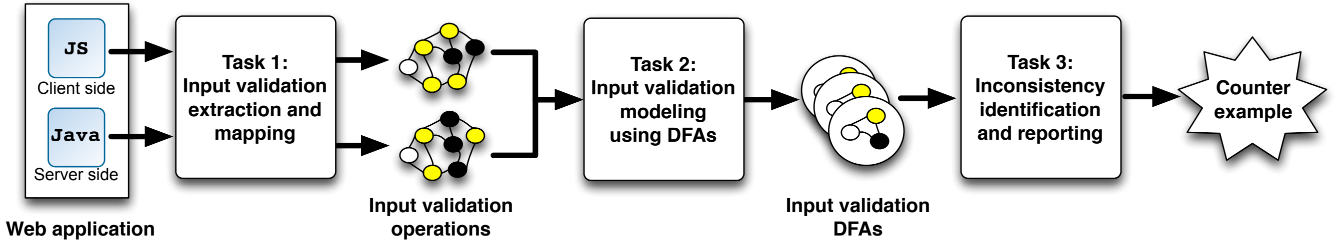

Figure 4: High-level view of ViewPoints.

representative values, generated using heuristics that are based on the type of the input field, and then uses the collected traces to compute dynamic slices for the executions. The trace collection is performed using a modified JavaScript interpreter, as discussed in Section 4.1. In particular, the modified interpreter converts all accesses to objects and arrays to accesses to specific memory locations, which avoids imprecision due to the use of objects, arrays, and aliasing. While slicing along a set of traces, ViewPoints handles internal function calls by inlining the code of the callees,[1] while external calls are treated as uninterpreted functions and are passed to the subsequent string analysis without further expansion (see Section 3.2). At the end of this phase, the slice for a given input is a sequence of statements that contains all of the relevant validation code for that input and is thus the summary validation function for it.

On the *server side*, input validation functions are written in Java, which is a language considerably more amenable to analysis than JavaScript. Therefore, ViewPoints identifies summary validation functions on the server side using static, rather than dynamic slicing [24]. This way, it can avoid possible problems of incompleteness related to the fact that a dynamic analysis might not exercise all possible behaviors of the validation functions. Similar to what it does on the client side, ViewPoints computes a slice for each input by considering in sequence all of the input validation functions associated with that input, inlining internal calls, and treating external calls as uninterpreted functions.

The final result of this first step of the ViewPoints technique is the following. For each validated input of the web application, ViewPoints produces two summary validation functions, one for the client side and a corresponding one for the server side. These pairs of summary validation functions are the input of the subsequent phase of the technique, which we describe in detail in the following section.

3.2 Input Validation Modeling Using DFAs

In its second step, ViewPoints performs automata-based string analysis to model the input checks performed by the summary validation functions computed in the previous step. Given a summary validation function, ViewPoints computes an over approximation of the string variable values at each program point using a flow- and path-sensitive, intra-procedural, automata-based symbolic string analysis algorithm that we defined in previous work for JavaScript [1]. The algorithm we use in ViewPoints is an extended version of the original algorithm that handles both Java and JavaScript string manipulation operations. Our string analysis algorithm represents possible values of a string variable at a program point using a deterministic finite automaton (DFA). Since DFAs can accept infinite sets, the data flow lattice corresponding to our string analysis is an infinite lattice with infinite chains, and we achieve convergence using an automata widening operator [3]. We use a symbolic automaton representation where the transitions of an automaton are represented as a Multi-terminal Binary Decision Diagram (MBDD).

The input to our algorithm (Algorithm 1) is the control flow graph (CFG) of the given validation function. Each node in the CFG represents a statement in the validation function. In this discussion, we will only concentrate on the types of nodes/statements that are crucial for our analysis.

Algorithm 1 STRINGANALYSIS(CFG)

```
1:  initParams();
2:  queue WQ := NULL;
3:  WQ.enqueue(CFG.entrynode);
4:  while (WQ ≠ NULL) do
5:     node := WQ.dequeue();
6:     IN := ⋃_{node' ∈ PredNodes(node)} OUT_{node'};
7:     if (node ≡ IF pred THEN) then
8:        tmp_{on_T} := tmp_{on_F} := IN;
9:        if (numOfVars(pred) = 1) then
10:          var := getPredVar(pred);
11:          predVal := EVALPRED(pred);
12:          tmp_{on_T}[var] : = IN[var] ∩ predVal;
13:          tmp_{on_F}[var] : = IN[var] ∩ (Σ* - predVal);
14:       end if
15:       tmp_{on_T} := (tmp_{on_T} ∪ OUT_{on_T})∇OUT_{on_T};
16:       tmp_{on_F} := (tmp_{on_F} ∪ OUT_{on_F})∇OUT_{on_F};
17:       if (tmp_{on_T} ⊈ OUT_{on_T}) then
18:          OUT_{on_T} := tmp_{on_T}; OUT_{on_F} := tmp_{on_F};
19:          WQ.enqueue(Succ(node));
20:       end if
21:    else
22:       tmp := IN;
23:       tmp[var] := EVALEXP(exp, IN);
24:       tmp := (tmp ∪ OUT)∇OUT;
25:       if (tmp ⊈ OUT) then
26:          OUT := tmp;
27:          WQ.enqueue(Succ(node));
28:       end if
29:    end if
30: end while
31: for (node ≡ RETURN TRUE ) do
32:    return OUT_{node}[parameter1]
33: end for
```

Algorithm 1 computes the least fixed point that over-approximates the possible values that string variables can take at any given program point. The algorithm is worklist based, that is, it keeps track of the CFG nodes that still needs to be processed in a worklist.

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. Given variable v, and the `IN` array for a statement, `IN`$[v]$ is a DFA that accepts all string values

[1]Although inlining can be problematic in the case of recursion and in the presence of deep call graphs, validation functions tend to be simple and are not affected by these issues. If this were not the case, it would always be possible to stop the inlining at a given depth and introduce approximations.

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.

The algorithm starts by initializing all of the validation function parameter values in the IN array of the entry statement to Σ^*—indicating that the validation function can receive any string value as input—and by adding the CFG node for the entry statement to the worklist.

At each iteration of the algorithm, a CFG node is extracted from the worklist, and the transfer function for the corresponding statement is computed (as described below). After computing the transfer function using the IN array, the OUT array for the current statement is updated using the join (union) and widening operators. Note that assignment, join, and widening operations on IN and OUT arrays are performed as point-wise operations. The analysis converges when the worklist becomes empty, which means that reevaluating the transfer functions would not change any of the OUT arrays. After convergence, the OUT value for the input parameter (*parameter1*) at the `return true` statement (i.e., the statement that represents a successful validation) is the result of our analysis—a DFA that accepts an over-approximation of the set of input values that the validation function identifies as valid.

Below we describe the transfer functions used to compute the OUT array for the two main types of statements handled by the algorithm: *Assignment Statement* and *Conditional 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. 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. EVALEXP evaluates the expressions as follows:

- *variable*: The set of values for the *variable* in the IN array (i.e., the DFA IN[*variable*]) is returned.
- *string-literal*: A singleton set that only contains the value of the *string-literal* is returned (i.e., a DFA that recognizes only the *string-literal*).
- **concatenate**(*expression1*, *expression2*): In this case, EVALEXP computes the concatenation of the regular languages resulting from evaluating *expression1* and *expression2* and returns it as the result (using the symbolic DFA concatenation operation discussed in [28]).
- **replace**(*pattern, string-literal, variable*): EVALEXP computes the result of replacing all string values in IN[*variable*] that match the *pattern* (given as a regular expression) with the *string-literal*. There are two types of pattern matching: **partial match** and **full match**. The match operation used is chosen based on the *pattern* value as follows. 1) If the value starts with symbol "^" and ends with symbol "$", a full match must be performed, that is, string in IN[*variable*] should be replaced only if it fully matches the regular expression in the *pattern*. This is accomplished by taking the difference between the language in IN[*variable*] and the language L(*pattern*) and adding the *string-literal* to the result. 2) In all other cases, a partial match is performed, where the result is computed by using the language-based replacement algorithm described in [28].
- **call** *function*(...): Because our analysis is intra-procedural, we only analyze one function at a time without following function calls. However, for the most commonly used functions, such as REPLACE and its variations, we have constructed function models that can be used during our analysis. More precisely, in the case of a function call, our algorithm operates as follows: If possible, it inlines the function; otherwise, it uses the model for this function, if one is available; and if none of these two options is possible, it returns Σ^*, which indicates an unknown string value.

Conditional Statement: This type of statement includes all program constructs that indicate a branching condition, such as *if statements*, *for loops*, *while loops*, and *do while loops*. Conditional statements consist of a predicate on variables and constants. Because they represent a branch in the program, unlike other statements, they are followed by two statements, one on the ON_TRUE branch and the other on the ON_FALSE branch. 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. OUT_{on_T} represents the values for the ON_TRUE branch, and OUT_{on_F} represents the values for the ON_FALSE branch. In order to compute these arrays, our algorithm first computes, using function EVALPRED, DFA_T—the DFA that accepts the set of string values that would make the predicate evaluate to true. Then, the algorithm computes the OUT_{on_T} array by intersecting the IN DFA with DFA_T. Conversely, to compute the OUT_{on_F} array, our algorithm intersects the IN DFA with the complement of DFA_T.

Function EVALPRED recursively traverses the predicate while constructing the DFA for each subexpression in the predicate. Logical operations are handled using automata union, intersection and complement operations, while all other expressions are mapped to regular expressions [1]. Note that our string analysis algorithm is not a relational analysis because we keep track of values of each string variable separately. This means that we cannot precisely represent predicates on multiple variables, and the function EVALPRED only handles predicates that contain a single variable. If there is a branch condition on multiple variables, our algorithm loses precision because it loses path sensitivity at that branch location (see line 9 in Algorithm 1). It is worth noting that, in our experiments, this did not cause precision loss since we have not encountered branch conditions with multiple variables in the programs used in our experimentation.

3.3 Inconsistency Identification and Reporting

At the end of the automata-based string analysis phase, we obtain two automata for each input field i: $A_c(i)$ (client side) and $A_s(i)$ (server side), where

- $L(A_c(i))$ (the language accepted by automaton $A_c(i)$) is an over-approximation of the set of string values that are accepted by the client-side input validation function for input field i, and
- $L(A_s(i))$ (the language accepted by automaton $A_s(i)$) is an over-approximation of the set of string values that are accepted by the server-side input validation function for input field i.

Using $A_c(i)$ and $A_s(i)$, we construct two new automata:

- $A_{s-c}(i)$ where $L(A_{s-c}(i)) = L(A_s(i)) - L(A_c(i))$, and
- $A_{c-s}(i)$ where $L(A_{c-s}(i)) = L(A_c(i)) - L(A_s(i))$.

We call $A_{s-c}(i)$ and $A_{c-s}(i)$ *difference signatures,* where:

- $L(A_{s-c}(i))$ contains strings that are accepted by the server side but rejected by the client side, and

- $L(A_{c-s}(i))$ contains strings that are accepted by the client side but rejected by the server side.

Let us now consider various scenarios for the difference signatures. If $L(A_{s-c}(i)) = L(A_{c-s}(i)) = \emptyset$, this means that our analysis could not identify any difference between the client- and server-side validation functions, so we have no errors to report. Note that, due to over-approximation in our analysis, this does not mean that the client and server-side validation functions are proved to be equivalent. It just means that our analysis could not identify an error.

If $L(A_{s-c}(i)) \neq \emptyset$, there might be an error in the server-side validation function. A server-side input validation function should not accept a string value that is rejected by the client-side input validation function—as we discussed earlier, this would be a security vulnerability that should be reported to the developer. Due to over-approximation in our analysis, however, our result could be a false positive. To prevent generating false alarms, we validate the error as follows. We generate a string $s \in L(A_{s-c}(i))$ and execute both the client and server-side input validation functions by providing s as the input value for the input field i. If client-side function rejects the string, and server-side function accepts it, then we are guaranteed that there is a problem with the application and report the string s as a counter-example to the developer. If we cannot find such a string s, then we do not report an error.

We also check if $L(A_{c-s}(i)) \neq \emptyset$ and, if so, we again generate a string to demonstrate the inconsistency between the client and server-side validation functions. Note that client-side validation functions accepting a value that the server rejects may not be as severe a problem as their counterpart. It is nevertheless valuable to report this kind of inconsistencies because fixing them can improve the performance and response time of the web application and prevent client-side vulnerabilities [19].

4. IMPLEMENTATION

In this section, we present the details of our implementation of ViewPoints and describe the various modules of the tool that perform the steps of the technique discussed in Section 3.

4.1 Input Validation Extraction

Web Deployment Descriptor.

Each web application must provide a Web deployment descriptor file, `web.xml`, as specified in the Java EE specification [7]. In this first step, our tool analyzes this file to understand and store references to the different components used within the web application, along with the paths to various library and framework configuration files. It then performs framework specific analysis of this information to discover how input fields of application forms are validated on both the client side and the server side. Upon discovery of this information, our tool gets a reference to the client- and server-side validation functions and proceeds with the validation code extraction from both sides. Our current implementation handles two popular J2EE frameworks: Struts and Spring MVC. Based on our experience, it could be extended to handle additional frameworks with relatively low effort.

Server-side Extraction.

Once our tool knows the specific server-side Java functions that are used to validate each input (i.e., form field), it accesses the corresponding class files using the Soot framework (`http://www.sable.mcgill.ca/soot/`) in order to analyze such validation functions. Because of the limitations of our current implementation, we had to apply several semi-automated transformations to the validation functions before being able to analyze them in isolation. Here is the list of transformations that our tool applies to each validation method (note that most of this transformations could be eliminated with further engineering) :

Input parameter re-writing: The function is transformed to remove all the formal arguments that are not of interest for our analysis. This allows us to have a simpler function and ignore many of the indirections introduced by the framework.

Function inlining and modeling: Our analysis inlines string operations performed by library functions. For instance, the `validateEmail` routine in the motivating example originally used the `isBlankOrNull` method from the library class `GenericValidator`. This function was inlined and corresponds to the expression ($val == null \ || \ val.trim().length \ == \ 0$).

Parameter inlining: Some of the validation routines might use parameters that are read from a configuration file. For example, the developer can specify a regular expression in the configuration that the validation function matches against the input. For such functions, our tool plugged the value of the parameter at all the corresponding uses inside the function.

After the above transformations are performed, our tool invokes the constant propagation and dead code elimination phases from the Soot framework to obtain a concise CFG for the validation function under analysis. These CFGs have two kinds of exit nodes—one that returns `true`, leading to the successful validation of the input, and the other one that returns `false`, leading to the rejection of the input. Our tool first uses this CFG to compute control and data dependences, which are then used to synthesize the PDG for the function. Upon the creation of the PDG, forward slicing is performed from the variable representing the input parameter being validated. This static slice contains all of the operations that are performed on the input variable and marks those that are string operations. Then, our tool performs backward slicing (on this forward slice) starting from the accepting nodes (i.e.,`return true` statements) to capture the string operations involved in the successful validation. The resulting slice is a CFG with only string operations performed on the validated input and is saved in an intermediate XML format later used by the differential string analysis phase.

Client-side Extraction.

The client-side validation functions extracted by our tool from the framework configuration files pose a different challenge to our analysis. These functions do not take as input the value of the single field being validated, but rather a bundle of the values of all of the fields in a web form. This creates a problem for our analysis, as we want to extract the summary validation function for a single field, not the whole web form. As discussed in Section 3.1.2, we solve this problem by executing the validation function using different input values and extracting the validation code for the target field using dynamic slicing.

To collect the traces later used for dynamic slicing, our tool uses HtmlUnit [8], which is a browser simulator based on Rhino [15]—a JavaScript interpreter. Using HtmlUnit, our tool can simulate the process of filling out a form (using values selected from a pool based on the type of the targeted form fields) and submitting it. During this simulation, our tool instruments the interpreter to track all of the JavaScript statements that operate on or test the content of the target fields. The tool then outputs these statements and all

other statements on which they depend. If there are function calls for non-native JavaScript functions, our tool inlines them so that the final code consists of only one validation function for the target field that ends with a “return true” statement.

For data that is coming from outside the syntactic scope of the validation function—such as masks that are used in some of the validation functions we encountered—our tool takes their values from the application execution trace and plugs them into the validation function before performing dynamic slicing.

4.2 Input Validation Modeling Using DFAs

Our tool loads the slices extracted from both client- and server-side validation functions and builds DFAs of string operations for these, as discussed in Section 3. Because more than one validation function can be associated with one field in the validation configuration, all the associated functions will be called separately for validating that field: if any of them rejects the input, then the input will be rejected. To correctly model this situation, our tool computes the intersection of all of the DFAs for all such functions (i.e., the resulting DFA accepts the intersection of the languages of the individual DFAs).

All the string analysis operations on the DFAs are performed by invoking the StrangerAutomaton library (`http://www.cs.ucsb.edu/~vlab/stranger/`), which internally uses the Mona tool to represent the DFAs (`http://www.brics.dk/mona/`).

4.3 Inconsistency Identification

In this step, for every input field i, our tool gets the two DFAs, one from the client side, and the other from the server side, and generates two difference signatures $A_{c-s}(i)$ and $A_{s-c}(i)$, which are DFAs that accept languages that correspond to the set differences between the languages of the client and server-side DFAs (as explained in Section 3). For each of the generated signatures that are not empty, our tool then tries to produce a counterexample. If the counterexample is found, the mismatch is confirmed and reported to the developers.

5. EMPIRICAL EVALUATION

To assess the usefulness of our approach, we used our implementation of ViewPoints to perform an empirical evaluation on a set of real-world web applications. In our study, we investigated the following two research questions:

RQ1: Can ViewPoints identify inconsistencies in client- and server-side input validation functions (or establish equivalence between them otherwise)?

RQ2: Is ViewPoints efficient enough to analyze real-world web applications within acceptable time and memory usage limits?

In the rest of this section, we will first describe the details of the experimentation performed for investigating RQ1 and RQ2, and then discuss our results.

5.1 Experimental Subjects

For our experiments, we selected seven real-world web applications from two open source code repositories: Sourceforge (`http://sourceforge.net`) and Google Code (`http://code.google.com`). Because our current implementation can handle web applications written using Java EE frameworks, we searched the two repositories for web applications with these characteristics. In addition, we discarded projects with a small user base or with a low activity level, so as to privilege web applications that were more likely to be widely used and well maintained.

Table 1: Web applications used in our empirical evaluation.

$Name$	URL
JGOSSIP	http://sourceforge.net/projects/jgossipforum/
VEHICLE	http://code.google.com/p/vehiclemanage/
MEODIST	http://code.google.com/p/meodist/
MYALUMNI	http://code.google.com/p/myalumni/
CONSUMER	http://code.google.com/p/consumerbasedenforcement
TUDU	http://www.julien-dubois.com/tudu-lists
JCRBIB	http://code.google.com/p/jcrbib/

Table 1 shows the list of web applications that we used for our experimentation and the URL from which they were obtained. The first four applications in the list are written using the Struts framework (`http://struts.apache.org/`): JGOSSIP is a messaging board application; VEHICLE is an application to manage vehicles owned by a company; MEODIST is an application for managing information about a club members; and MYALUMNI is a social network application for school alumni. The last three applications are written using the Spring MVC framework (`http://www.springsource.org/`): CONSUMER is a customer relationship management application; TUDU is an on-line application for managing todo lists; and JCRBIB is a virtual library application that supports user collaboration. Based on their descriptions, these web applications cover a wide spectrum of application domains. Moreover, because of the way they were selected, most of these applications are popular and widely used in practice. JGOSSIP, for instance, has been downloaded almost 30,000 times from its Sourceforge page.

5.2 Experimental Procedure and Results

For conducing our experiments, we used a Ubuntu Linux machine with an Intel Core Duo 2.4Ghz processor and 2GB of RAM running Java 1.6. To collect data to answer RQ1 and RQ2, we ran ViewPoints on the web applications considered. For each web application, ViewPoints first analyzed the application's configuration to identify its inputs and corresponding client- and server-side validation functions. It then built client- and server-side summary validation functions for each input.

Table 3 shows relevant data for this part of the analysis. The first column in the table lists the application name, followed by the number of forms extracted (Frm) and the total number of inputs across all forms ($Inputs$). Column VI_C (resp., VI_S) lists the number of inputs for which a client-side (resp., server-side) validation function is specified in the configuration. Similarly, column ET_C (resp., ET_S) lists the time taken, in seconds, to extract the summary validation functions for these inputs on the client side (resp., server side). For example, web application CONSUMER contains 3 forms, for a total of 21 inputs. Of these inputs, 14 are validated on the client side, whereas all of 21 of them are validated on the server side. It took 68.4 and 1.1 seconds to extract the summary validation functions on the client side and server side, respectively. Note that the time required to compute the client-side summary validation functions is much higher than the time to extract server-side summary validation functions. This difference is due to the additional time required to perform dynamic slicing on the client side, which in turn requires ViewPoints to load and run JavaScript functions in the browser.

After building client- and server-side summary validation functions for each input, ViewPoints constructed the corresponding DFAs, as described in Section 3. Table 2 shows details about this

Table 2: Relevant data on the input validation modeling step of the technique.

$Subject$	$Client - side\ DFA$							$Server - side\ DFA$						
		min		max		avg			min		max		avg	
	Avg size (mb)	S	B	S	B	S	B	Avg size (mb)	S	B	S	B	S	B
JGOSSIP	6.03	4	10	35	706	6	39	6.05	4	24	35	706	6	41
VEHICLE	4.83	4	24	7	41	5	26	4.84	4	24	7	41	5	26
MEODIST	5.67	5	25	5	25	5	25	5.67	5	25	5	25	5	25
MYALUMNI	3.17	4	10	4	10	4	10	3.16	3	24	5	25	5	25
CONSUMER	5.34	4	10	17	132	5	25	5.34	4	24	17	132	7	41
TUDU	6.12	4	10	4	10	4	10	6.12	3	24	23	264	8	68
JCRBIB	5.37	4	10	4	10	4	10	5.38	5	25	5	25	5	25

Table 3: Relevant data on the input validation extraction step of the technique.

$Subject$	Frm	$Inputs$	VI_C	ET_C(s)	VI_S	ET_S(s)
JGOSSIP	25	83	74	329.80	83	4.38
VEHICLE	17	41	41	155.48	41	2.04
MEODIST	18	62	62	192.20	62	1.93
MYALUMNI	46	141	0	0.00	141	4.28
CONSUMER	3	21	14	68.40	21	1.10
TUDU	3	11	0	0.00	11	0.78
JCRBIB	21	45	0	0.00	45	1.51

part of the analysis. For each application, and both for the client side and the server side, the table shows: the average size of the DFAs in megabytes, followed by the minimum, maximum, and average number of states (column S) and BDD nodes (column B). (The number of BDD nodes represents the size of the symbolic representation of the DFA's transition relation.) As an example, the application TUDU has DFAs with an average of 4 states and 10 BDD nodes for the client side, whereas it has DFAs with an average of 8 states and 68 BDD nodes for the server side. Note that, when client-side validation is absent for an input, the DFA for that input accepts the language Σ^*. Hence, TUDU has a client-side DFA even though it has no client-side validation code (see Tables 2 and 3).

Finally, ViewPoints compared client- and server-side DFAs to identify possible inconsistencies among them. The results of this comparison for our subjects is shown in Table 4. For each application, the table reports the time it took ViewPoints to perform differential string analysis, in milliseconds, and the number of inputs with identified (and confirmed) inconsistencies. Specifically, column A_{C-S} shows the number of inputs for which the client side accepts strings that would be rejected by the server side, whereas column A_{S-C} shows the opposite. For JGOSSIP, for instance, the differential string analysis took around 3 seconds and identified nine client-side inconsistencies and two server-side inconsistencies.

5.3 Discussion

As the results in Table 4 show, ViewPoints was able to find both types of inconsistencies: client checks that are more strict than server checks and vice versa. We manually checked all the results and 1) verified that all identified inconsistencies correspond to actual inconsistencies (i.e.,our tool did not generate any false positives), and 2) confirmed that there are no inconsistencies other than those found by our automated analysis (i.e., our tool did not generate any false negatives). For JGOSSIP, in particular, ViewPoints found two instances of the inconsistency that we presented in our motivating example. As we explain in Section 2, such inconsistencies represent actual vulnerabilities in the code that a malicious user may be able to exploit. For the remaining applications, four out of six contain input validation inconsistencies on the client side.

Table 4: Data on the inconsistency identification step of the approach and overall results.

$Subject$	$Time$ (ms)	A_{C-S}	A_{S-C}
JGOSSIP	3220	9	2
VEHICLE	1486	0	0
MEODIST	1745	0	0
MYALUMNI	2853	141	0
CONSUMER	1019	7	0
TUDU	595	11	0
JCRBIB	1168	45	0

A special case is that of MYALUMNI, which has 141 inputs that are inconsistently validated at the client side and server side. For this application, the developers provided no validation whatsoever on the client side, and thus all the 141 inputs that are checked on the server side are inconsistently validated. Although these results are preliminary, and further experimentation would be needed to confirm them, they provide strong supporting evidence for answering RQ1: ViewPoint is indeed able, at least for the cases considered, to identify inconsistencies in client- and server-side input validation functions.

RQ2 relates to the efficiency and practicality of the analysis. From Table 3, we can see that the extraction phase of ViewPoints took between 0.78 and 4.38 seconds for the server-side validation functions. Although for the client-side functions the numbers are higher, due to the more expensive analysis performed on the client side (see Section 5.2), the maximum total time needed to analyze one of the web applications considered is less than six minutes.

Table 2 illustrates the space cost of ViewPoints. As the table shows, the space needed to store the DFAs is negligible, as it is less than seven megabytes in all cases. Finally, Table 4 shows the time needed to perform the comparison of two DFAs. Also in this case, the time it takes ViewPoints for the comparison is in the order of a few seconds and, thus, negligible. We can therefore provide a positive answer to RQ2 as well.

Overall, these results provide preliminary, yet clear evidence that ViewPoints can be both practical and useful.

6. RELATED WORK

Software vulnerabilities are a significant problem. Therefore, there exists a wide range of solutions that attempt to address the problem from different angles. In the following discussion, we mainly focus on the detection of web application vulnerabilities. Techniques to detect web application vulnerabilities can be divided into dynamic and static program analysis techniques. Dynamic analysis techniques (e.g., [10, 16, 20]) perform their computations while actually executing the analyzed code. As a result, these techniques do not generate false positives, since every detected error path corresponds to a true path that the program can take at run-

time. The disadvantage of dynamic scanners is that they experience problems regarding the coverage of all possible paths through the program. The number of these paths is generally unbounded, and grows exponentially with each branch in the program. Hence, it is easy to miss vulnerabilities due to program paths that were not taken into account.

In this paper, we focus on the detection of security flaws by means of (mainly) static code analysis. This has the advantage that the whole code base is checked, but the drawback that the analysis may report warnings for correct code (i.e., false positives). There are a number of approaches that deal with static detection of web application vulnerabilities. Xie and Aiken [27] addressed the problem of statically detecting SQL injection vulnerabilities in PHP scripts by means of a three-tier architecture. In this architecture, information is computed bottom-up for the intra-block, intra-procedural, and inter-procedural scope. As a result, their analysis is flow-sensitive and inter-procedural. This is comparable in power to Pixy [13]. Both systems use traditional data flow analysis to determine whether unchecked user inputs can reach security-sensitive functions (so-called sinks) without being properly checked. However, they do not calculate any information about the possible strings that a variable might hold. Thus, they can neither detect all types of vulnerabilities (such as subtle SQL injection bugs) nor determine whether sanitization routines work properly.

Static analysis of strings has been an active research area, with the goal of finding and eliminating security vulnerabilities caused by misuse of string variables. String analysis focuses on statically identifying all possible values of a string expression at a program point, and this knowledge can be leveraged to eliminate vulnerabilities such as SQL injection and XSS attacks. In [14], multi-track DFAs, known as *transducers*, are used to model replacement operations in conjunction with a grammar-based string analysis approach. The resulting tool has been effective in detecting vulnerabilities in PHP programs. Wassermann et al. [25, 26] 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 for encoding possible values that string expressions can take at each program point [2, 28]. In this approach, the values of string expressions are computed using a fixed point computation. Complex string manipulation operations, such as replacement, can also be modeled with the automata representation. In order to guarantee convergence in automata-based string analysis, several widening operators have been used [3,6,28]. Constraint-based (or symbolic-execution-based) techniques represent a third approach for static string analysis. Such techniques have been used for the verification of string operations in JavaScript [18] and the detection of security flaws in sanitization libraries [12].

Previous work on string analysis for finding bugs and security vulnerabilities in web applications, including our work in [1], concentrates on one side of the web application—either the client or the server—and assumes the existence of a specification that defines (models) the set of acceptable and malicious string values. The purpose of the analysis is then to determine whether the application code, and its sanitization functions, properly enforce these specifications. ViewPoints is different in that we do not assume that such specifications are available. Instead, we compare the input constraints enforced by both the client-side and the server-side code. Differences in these constraints are taken as indications of problems. A somewhat related idea was presented in a system called NoTamper [4]. In that paper, 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 [5], the same authors propose WAPTEC, which uses symbolic execution of the server code to guide the test case generation process and expand coverage. While the overall goal of WAPTEC is similar to ours, the techniques used to achieve this goal are quite different. For example, WAPTEC uses guided, dynamic analysis on the server side to generate a series of possible exploit inputs. ViewPoints, on the other hand, uses static analysis to model the server code. This potentially allows ViewPoints to find more vulnerabilities than WAPTEC, whose effectiveness depend on the coverage achieved by its dynamic analysis. In addition, ViewPoints uses an automata-based symbolic string analysis that over-approximates loops and does not bound the lengths of the strings; conversely, WAPTEC extracts client-side constraints from JavaScript code using Kudzu [18], an existing technique in which both loops and string lengths are bounded during the analysis.

7. CONCLUSION

Because many web applications store sensitive user information and are easily accessible, they are a common target of attackers. Some of the most insidious attacks against web applications are those that take advantage of input validation vulnerabilities—inadequate input checks that let attackers submit malicious inputs to an application, often with catastrophic consequences. Unfortunately, automatically checking validating functions would require a complete specification of the legal inputs for an application, which is rarely available. In this paper, we present ViewPoints, a novel technique that leverages differential string analysis to identify potentially erroneous or insufficient validation of user inputs in web applications. ViewPoints is based on the insight that developers typically perform redundant input validation on the client side and server side of a web application, and that it is therefore possible to use the validation performed on one side as a specification for the validation performed on the other side. ViewPoints operates by automatically extracting client- and server-side input validation functions, modeling them as deterministic finite automata, and comparing client- and server-side automata to identify and report inconsistencies between the two sets of checks. To assess the efficiency and effectiveness of ViewPoints, we implemented it for web applications built using Java EE frameworks and used it to analyze a set of real-world web applications. The results of this initial evaluation are promising and motivate further research in this direction: ViewPoints was able to automatically identify a number of inconsistencies in the input validation checks of the web applications we analyzed. In addition, the analysis was extremely fast, which demonstrates the practical applicability of the approach.

There are several possible directions that we are considering for future work. In the short term, we will extend our implementation so that it can handle a larger number of types of web applications (e.g., applications written in different languages or with validation functions that are not explicitly identified in the application's configuration) and perform a more extensive set of empirical studies. We will also investigate code synthesis techniques for automatically fixing the input validation inconsistencies identified by our approach; after ViewPoints has identified one or more inconsistencies, it could use the information in the string automata to guide the automatic construction of additional validation code to be added to the client, the server, or both. One more general direction for future work has to do with the solution space for the string analysis on which ViewPoints relies. There are many dimensions that characterize string analysis techniques: (1) static vs. dynamic, (2) grammar-based vs. automata-based, (3) bounded path vs. un-

bounded path, (4) bounded domain vs. unbounded domain, (5) relational vs. non-relational, (6) size based vs. value based, and so on. In our current approach, we consider only a specific solution in this space. In the future, we will study the many trade-offs between different approaches (e.g., between a sound approach, which generates no false negatives, and a complete approach, which generates no false positives) and investigate whether a different approach may be advantageous in terms of efficiency, precision, expressiveness, or ability to compare resulting models.

8. ACKNOWLEDGMENTS

This research was supported in part by NSF grants CCF-0916112 and CNS-1116967 to UCSB and NSF grants CCF-0964647 and CNS-1117167 to Georgia Tech. Muath Alkhalaf is funded in part by a fellowship from King Saud University.

9. REFERENCES

[1] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[2] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the Symposium on Security and Privacy (S&P)*, 2008.

[3] C. Bartzis and T. Bultan. Widening arithmetic automata. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer-Verlag, July 2004.

[4] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan. NoTamper: Automatic, Blackbox Detection of Parameter Tampering Opportunities in Web Applications. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan. WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[6] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh. A Practical String Analyzer by the Widening Approach. In *Proceedings of the 4th Asian Symposium on Programming Languages and Systems (APLAS)*, pages 374–388, 2006.

[7] N. Coward and Y. Yoshida. Java Servlet Specification Version 2.4. Technical report, Nov. 2003.

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

[9] W. Halfond, S. Anand, and A. Orso. Precise Interface Identification to Improve Testing and Analysis of Web Applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 285–296, 2009.

[10] W. Halfond, A. Orso, and P. Manolios. WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation. *IEEE Transactions on Software Engineering (TSE)*, 34(1):65–81, 2008.

[11] W. G. Halfond, J. Viegas, and A. Orso. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the International Symposium on Secure Software Engineering*, 2006.

[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 Security Symposium*, 2011.

[13] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[14] Y. Minamide. Static approximation of dynamically generated web pages. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, pages 432–441, 2005.

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

[16] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference (SEC)*, 2005.

[17] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 1–12, 2010.

[18] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland)*, 2010.

[19] 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.

[20] D. Scott and R. Sharp. Abstracting Application-Level Web Security. In *Proceedings of the 11th International World Wide Web Conference (WWW)*, 2002.

[21] The OWASP Foundation. Data Validation, 2010. `http://www.owasp.org/index.php/Data_Validation`.

[22] The OWASP Foundation. Top Ten Most Critical Web Application Vulnerabilities, 2010. `http://www.owasp.org/documentation/topten.html`.

[23] The OWASP Foundation. Validation Performed in Client, 2010. `http://www.owasp.org/index.php/Validation_performed_in_client`.

[24] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[25] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–41, 2007.

[26] 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.

[27] Y. Xie and A. Aiken. Static Detection of Security Vulnerabilities in Scripting Languages. In *Proceedings of the 15th USENIX Security Symposium (USENIX-SS)*, 2006.

[28] F. Yu, T. Bultan, M. Cova, and O. H. Ibarra. Symbolic String Verification: An Automata-based Approach. In *Proceedings of the 15th International SPIN Workshop on Model Checking Software (SPIN)*, pages 306–324, 2008.