WAVES: Automatic Synthesis of Client-side Validation Code for Web Applications

Nazari Skrupsky
University of Illinois
Chicago, USA
nskroups@cs.uic.edu

Maliheh Monshizadeh
University of Illinois
Chicago, USA
mmonsh2@uic.edu

Prithvi Bisht
University of Illinois
Chicago, USA
pbisht@cs.uic.edu

Timothy Hinrichs
University of Illinois
Chicago, USA
hinrichs@uic.edu

V.N. Venkatakrishnan
University of Illinois
Chicago, USA
venkat@cs.uic.edu

Lenore Zuck
University of Illinois
Chicago, USA
lenore@cs.uic.edu

Abstract

The current practice of web application development treats the client and server components of the application as two separate but interacting pieces of software. Each component is written independently, usually in distinct programming languages and development platforms — a process known to be prone to errors when the client and server share application logic. When the client and server are out of sync, an "impedance mismatch" occurs, often leading to software vulnerabilities as demonstrated by recent work on parameter tampering. This paper outlines the groundwork for a new software development approach, WAVES, where developers author the server-side application logic and rely on tools to automatically synthesize the corresponding client-side application logic. WAVES employs program analysis techniques to extract a logical specification from the server, from which it synthesizes client code. WAVES also synthesizes interactive client interfaces that include asynchronous callbacks whose performance and coverage rival that of manually written clients while ensuring no new security vulnerabilities are introduced. The effectiveness of WAVES is demonstrated and evaluated on three real-world web applications.

I INTRODUCTION

Current practices in mainstream web development isolate the construction of the client component of an application from the server component. Not only are the two components developed independently, but they are often developed by different teams of developers, some projects going so far as to outsource the client development [1]. Partly this is just good project management, but it is also a byproduct of the fact that the client component is often written using a different programming language and platform (HTML and JavaScript in a web browser) than the server (e.g., PHP, Java, ASP), therefore necessitating developers with different skill sets. When the client and server are supposed to share application logic but do not, an "impedance mismatch" occurs.

In this paper we are concerned with a specific kind of application logic: the input validation logic. Examples of input validation include input character validation ("username does not contain special characters"), required fields ("phone number is required") and logical checks ("credit card expiry date in past"). Input validation on the client improves the user experience because it provides the user immediate feedback about errors; furthermore, it often reduces network and server load. Input validation on the server is necessary for security. For if the server assumes all the data it has been sent has been validated by the client, a malicious user can circumvent the client, submit invalid data to the server, and exploit the lack of server-side validation. Recent work on parameter tampering [2–5] has uncovered impedance-mismatch vulnerabilities that enable takeovers of accounts and unauthorized financial transactions in commercial and open-source websites as well as third-party cashiers (such as `PayPal` and `Amazon Payments`).

Recently web development frameworks have begun to address the problem of application logic shared across client and server. For example, the Google Web Toolkit allows a programmer to put shared code into a special directory that is then replicated on both the client and server. For legacy applications, however, new programming language idioms are of little help, and manually retrofitting an application with extensive client-side and server-side validation is error-prone and expensive, especially since the client

and server validation must be synchronized every time the application is updated.

In this paper, we address the impedance mismatch problem for legacy web applications that have no interactive client-side input validation. We first note that these applications that were commonplace before Web 2.0 paradigm of development began and are still in deployment today. Our approach is to automatically examine the source code of a web application, identify the server-side input validation logic, and replicate that logic on the client. While designed for legacy applications, our approach can also be deployed in modern web development frameworks, thereby enabling a developer writing a new application to author only the server-side validation code while the framework automatically installs the corresponding client-side validation code. While such technology is most obviously beneficial because it simplifies a web developer's job, it can also help to improve the security of newly written applications. If the developer can focus all her energies on writing the input validation logic for the server (instead of splitting her time between the client and server), she is more likely to include all the validation necessary for the security of the application. Thus our approach has several high-level benefits:

- *Improved Usability.* Applications whose client input validation has been automatically generated provide end users with all the input validation expected of today's web apps.
- *Greater Development Efficiency.* Developers no longer write the same validation code twice since the client code is automatically synthesized from the server code.
- *Improved Security.* The development team can devote more resources to the design and implementation of the server code, thereby being more likely to include all the input validation necessary for the application's security.

Our realization of this approach, WAVES, uses program analysis to automatically extract a logical representation of the input validation checks on the server and then synthesizes efficient client-side input validation routines. Of particular note is that WAVES also generates code for validation checks that involve server-side state by utilizing asynchronous requests (AJAX) to perform the required validation. Because such validation routines can increase server and network load, WAVES allows a developer to choose the extent to which such validation checks are generated.

This paper is organized as follows: Section II presents the problem by means of a running example and the challenges our approach must overcome. In Section III we present a high level overview of our approach. In Section IV we present a detailed description of the different components of our approach. Section V presents an evaluation. We evaluate our approach and tool over three real-world web applications. Our experience indicates that our approach offers a promising improvement to current mainstream web development practices. Section VI presents a discussion and security analysis of our approach. In Section VII we present related work, and in Section VIII we conclude.

II EXAMPLE AND CHALLENGES

Figure 1 presents the client side interface of a simple user registration application. We will use this application as the running example throughout the paper. In this application, a user supplies her user ID and her password twice (for confirmation purposes). There are three validation checks performed by this application:

1. The characters in user ID belong to a specified character set, which in this case is all alphanumeric characters along with a hyphen and underscore.
2. The two supplied passwords match.
3. The user ID is available for creating an account (i.e., it is not already taken by another user).

Suppose the developer authors the server component of the application and implements these checks in server code. Our goal is to *automatically synthesize* the corresponding client side input validation routines. The high-level challenges in achieving our goal include:

- *Automatic inference of server-side constraints.* While the client side validation constraints are expressed in terms of form fields, the server side validation may be performed in terms of server-side variables within deeply nested control flows of the application. The server-side constraints must be extracted and expressed in terms of the form fields.

Figure 1: Running example of a registration application

- *Validation involving the server.* Sometimes validation involves server-side state (such as the database), but moving that data to the client is often impractical because of performance, security, privacy, and/or staleness issues. For example, when a user ID is submitted to the server, the server checks if the ID is unique in the database. Moving all the user IDs to the client is impractical; thus, some clients asynchronously contact the server to check if the ID is unique. The code that is generated must allow the client to asynchronously contact the server (and for the developer to control which asynchronous validations are performed).

- *Preservation of application logic and security.* The code that is generated must neither compromise the security of the application nor disable existing functionality.

III OUR APPROACH

This paper presents an approach for improving the web application development process that alleviates the problem of inconsistent client and server input validation: WAVES (Web Application Validation Extraction and Synthesis). Unlike traditional approaches that require developers to write and maintain the input validation routines in both the client and the server codebases, WAVES requires developers to only maintain the input validation code on the server. WAVES then automatically synthesizes the corresponding validation code for the client.

Figure 2 shows the desired transformation of the running example[1]. The non-interactive version of the web application is shown on the left and is comprised of the client-side code (`register.html`) and server-side code (`register.php`). Guided by validation checks in `register.php`, WAVES generates the interactive version of this application shown on the right (newly added code in bold font). The retrofitted client validates each of its three fields as soon as the data in any field changes. For instance, when the user changes `uid`, the client checks that only alphanumeric characters, hyphens, and underscores appear in the user ID; additionally, the client asks the server if the user ID is unique in the database.

WAVES breaks this transformation into four conceptually distinct phases:

(1) Server analysis. WAVES performs dynamic program analysis—submitting form inputs to the server and inspecting the sequence of instructions that the server executes. The key insight is that when the server is given an input it accepts, the sequence of if-statements it executes contain all the input validation constraints it checks. So after submitting form field inputs that the server accepts and rewriting the if-statements in terms of the original form field inputs, we have a list of potential input validation constraints. We then analyze each one to determine if it is truly an input validation constraint—a constraint that when falsified causes the server to reject the input. Once the list has been reduced to the set of actual input validation constraints, we identify which constraints are dependent on the server's environment (the *dynamic* constraints) and which are not (the *static* constraints).

In our running example, we first submit legitimate values for `uid` and the two passwords. The server checks if the `uid` contains only the permitted characters, that the `uid` is unique in the database, and that the passwords match. The server will undoubtedly execute other if-statements, which must be an-

[1]For concreteness, the example shows the client implemented in JavaScript, and AJAX, and the server implemented in PHP. While these languages are the ones addressed by our current prototype, the underlying techniques used by our approach are agnostic to programming languages. Our implementation can be easily extended to other server platforms (e.g., JSP, .NET) and client platforms (e.g., ActionScript).

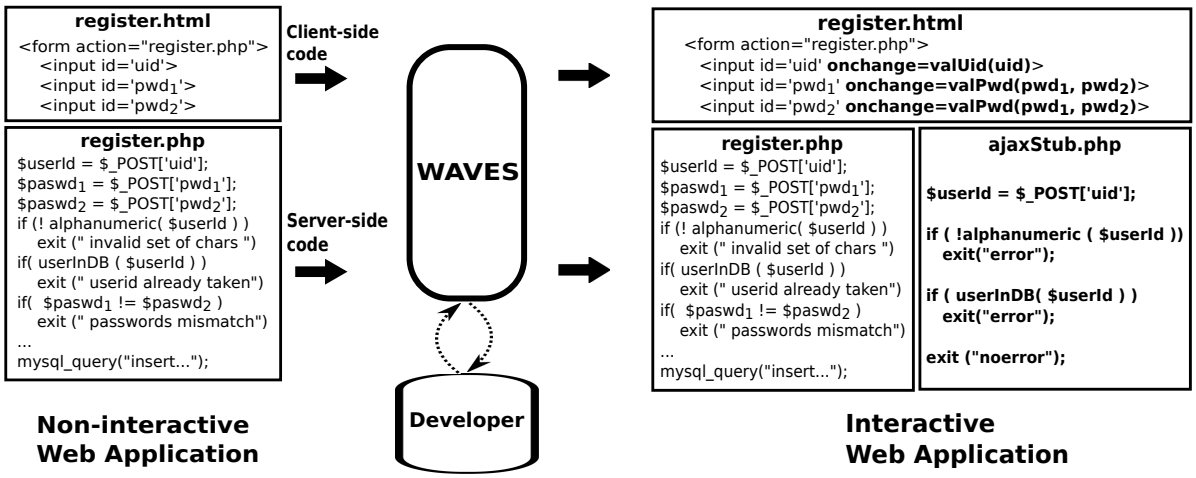

Figure 2: WAVES: synthesizing client-side validation code.

alyzed to find the constraints of interest. Finally, we separate the static constraints (the alphanumeric constraint on `uid` and the password equivalence constraint) from the dynamic constraints (the fact that the `uid` is unique in the database).

(2) Client-side code generation. In WAVES, once the static and dynamic constraints have been extracted from the server, we synthesize client-side code to check those constraints. The static constraints can be checked directly by JavaScript code, but the dynamic constraints can only be checked after communicating with the server. So for each form field, we generate code that performs two tasks: checking if any errors arise because of static constraints and if not, checking if any errors arise because of dynamic constraints by asynchronously contacting the server. In our example, the client code for the `uid` field first checks if the value contains only the permitted characters and if so asks the server if the value is unique in the database. It also checks that the two passwords are the same (whenever there are values for both fields).

(3) Server-side code generation. The asynchronous messages sent by the client to check the dynamic constraints for a form field can only be responded to by special-purpose server-side code. (The original code assumes the user provided values for all form fields, but the client's asynchronous messages aim to check constraints even before the user completes the form.) These server stubs behave the same as the original server code but operate properly when data for only one or two form fields is provided. Different techniques can be used to generate server stubs, but we recommend code slicing. To minimize server communication, we also recommend checking all of the dynamic constraints for a form field via one asynchronous message. In our example, we generate a single server stub that checks if the given `uid` is unique in the database. Unlike the original server code, that stub does not perform any checks on the password fields.

(4) Integration. Once the new client and server code has been generated, it must be integrated into the existing client and server codebases. In this step, the developers can choose to disallow some generated code parts to be integrated into the application since there are some constraints which may reveal information about the server state or data. How the integration is done depends on the programming languages for client and server, but ideally regenerating client and server code to reflect changes in the application will require minimal additional integration effort.

IV TECHNICAL DESCRIPTION

In this section we describe each of the four phases of our approach in more detail.

1 SERVER ANALYSIS

The server analysis phase of WAVES aims to discover all of the constraints on form fields that the server enforces (Algorithm 1). Besides the URL of the web form, WAVES is given inputs for the form that the web server accepts, i.e., a single error-free

input. WAVES begins by submitting this initial input (the *success* input) to the server, which returns a trace of the instructions that the server executed in response (Algorithm 1 Line 1). Instrumenting the server to return such a trace is done offline and was described in prior work [3]. Since the success input is accepted by the server, those inputs satisfy all of the constraints the server enforces, and consequently all the input validation constraints will appear as if-statements in the resulting server trace. By rewriting those if-statements in terms of the original inputs (using taint analysis of [3]), WAVES extracts the set of conditions that were true of the form field inputs: $\{C_1, \ldots, C_n\}$ (Line 2).

Not each of the resulting conditions, if falsified, leads to an error. For example, consider an additional condition in the `register.php` file that writes information to a log under certain conditions. Those conditions also appear as if-statements but are irrelevant for input validation purposes. Thus, WAVES next identifies which of the conditions (C_i) if falsified lead to an error. For each C_i, WAVES constructs inputs that satisfy $\neg C_i$ using a string solver [6] (Line 5) but is otherwise as similar to the original success input as possible (Line 6). The intent is that this *failure input*, if rejected by the server, demonstrates that $\neg C_i$ is an error condition. If the server rejects a failure input, we know that the conjunction of conditions in that trace (after rewriting them in terms of the original form field inputs) is an error condition: $D_1 \wedge \cdots \wedge D_m$ (Line 7-8). That is, every input satisfying $D_1 \wedge \cdots \wedge D_m$ contains at least one error. The constraints that WAVES extracts is a collection of such error conditions (Algorithm 1 Line 13).

Simplification. The algorithm described above is sound by construction (proof in Appendix IX) : if WAVES finds an error condition, then any input satisfying that condition will cause an error. But in practice each of these error conditions is usually too weak to be useful because it includes checks on all of the form fields. The only time the error condition is satisfied is therefore when all of the form fields have values. One of the design goals of WAVES is to give the user real-time feedback each time she enters a new form field value, a goal that the error conditions described so far fail to achieve. To illustrate the issue, consider a failure input where the user ID satisfies the necessary conditions but where the two passwords are unequal. The above algorithm would identify the following conjunction as an error condition.

$(uid \in$ `[0-9a-zA-Z_-]*`$) \wedge isUnique(\ uid\ ) \wedge (pwd_1 \neq pwd_2)$

The problem is that this constraint can only be evaluated once there are values for all three form fields. Moreover, this constraint only ensures that if `uid` is alphanumeric and not already present in the database then the passwords must be equal. While the correct simplification of this example is obvious from our description of the application ($pwd_1 \neq pwd_2$), in general we cannot soundly eliminate conjuncts from an error condition.

Simplification is therefore crucial to the practical utility of WAVES. The basic premise behind our simplification routine is that we have two kinds of server traces: those with errors and those without errors. The conjunction of conditions in a trace with errors is an error condition: any input that satisfies *all* the constraints is rejected by the server. The conjunction of conditions in a trace without errors is a *safe condition*: no input that satisfies a safe condition is rejected by the server. Thus, we can simplify an error condition by removing all safe conditions contained within it (Algorithm 1 Line 13).

Unfortunately, it is just as important and difficult to simplify a safe condition as it is to simplify an error condition. All we know is that no input satisfying all the conjuncts together causes an error. But if WAVES knows which form fields are independent of which others in terms of all control paths (the *indep* argument to Algorithm 1), it can break large safe conditions and error conditions into independent conjunctions of constraints (Lines 3, 9). WAVES then records each independent conjunction of constraints as either a safe condition (Line 11) or as an error condition (Line 13). Any error condition that is also a safe condition is eliminated as an error condition (Line 13). We found this independence information crucial to generating practically useful error conditions.

Static and Dynamic Constraints. The constraints WAVES extracts from the server are one of two kinds: static or dynamic. Dynamic constraints depend on the server's environment (e.g., file system or database), while static constraints do not. The difference is important because static constraints will never change and hence can easily be synthesized on the client, but dynamic constraints change each time the server's environment changes and hence for correctness can only be checked by the server. The way WAVES identifies dynamic constraints is straightfor-

Algorithm 1 WAVES (url, suc_input, indep)

Returns: Client validation code in JavaScript and server stubs in PHP.

```
1: trace := SUBMIT(url, suc_input)
2: C_1 ∧ ··· ∧ C_n := CONSTRAINTS(trace)
3: safe := PARTITION(C_1 ∧ ··· ∧ C_n, indep)
4: for all C_i do
5:   bl := SOLVER(¬C_i)
6:   bl := bl ∪ ELIMINATEVARS(suc_input, VARS(bl))
7:   trace := SUBMIT(url, bl)
8:   D_1 ∧ ··· ∧ D_m := CONSTRAINTS(trace)
9:   P = PARTITION(D_1 ∧ ··· ∧ D_m, indep)
10:  if SERVERACCEPTED(trace) then
11:    safe := safe ∪ P
12:  else
13:    errors := errors ∪ (P − safe)
14: (static, dynamic) := SPLITSTATICDYNAMIC(errors)
15: return (GENCLIENT(static),GENSERVER(dynamic))
```

ward: any constraint referencing the server's environment (e.g., the database, files, sessions, global variables, time, etc.) is a dynamic constraint; all others are static (Algorithm 1 Line 14).

Discussion. One of the limitations of server-side analysis is that if the constraints enforced by the server are complex enough, it may be that a single success input is insufficient to extract all of the constraints enforced by the server. While we did not encounter this limitation in the applications we evaluated, to address such forms we would apply the algorithms we developed in prior work to construct additional success inputs automatically [3].

2 CLIENT-SIDE CODE GENERATION

Generating the client code to check a collection of static and dynamic constraints is broken into two distinct components: generating code that checks the static constraints and generating code that checks the dynamic constraints. Recall that the static constraints can be checked directly on the client, and the dynamic constraints require communicating with the server. For each form field, WAVES generates an event handler that first checks the static constraints for an error and if none is found then checks the dynamic constraints.

Static constraints. Each static constraint is basically a conditional test on form fields that can include any number of string and integer manipulation functions (e.g., $len(trim(x)) > 6$ ensures the length of field x after removing whitespace from both ends is greater than 6). Formally, each constraint is represented in the logic of strings and integer arithmetic. Loosely, this means these are the usual boolean connectives ($\wedge$, $\vee$, $\neg$) together with a collection of string and integer operations that are found in many programming languages.

Given the static constraints that must hold of the form, we must identify which constraints are pertinent to each form field so that each time that form field changes we can check the right constraints. Choose too many, and the user may see error warnings for form fields that she has not even filled in; choose too few, and she will not be warned of errors when they exist. This identification is quite simple after converting the constraints to a canonical form (conjunctive normal form): for form field f collect all those constraints where f occurs.

There are some static constraints which may reveal secret information about the server. For example, the constraint `password == "secret"` (revealing the hard-coded password "secret", which is a poor security practice) should not be added to client-side code. These constraints occur rarely, and we have not encountered any warnings of this type. The string solver can recognize constraints in which a form field value is checked against a constant value, however it cannot identify whether this constant value is a server-related secret. Therefore, the developer should choose to allow these type of static checks to appear on the client-side or not. In practice, The number of these warnings is insignificant compared to the benefits of using WAVES.

Generating client-side code that checks the constraints for a given form field is a linear time and space procedure, assuming the client has implementations of all the string and integer functions. Each form field reference is translated into a lookup of that field's value (e.g., *uid* becomes `document.getElementById('uid')`). Each boolean connective is translated to the client's version of that connective (e.g., $\wedge$ becomes &&). Each string or integer function is translated to a client implementation of that function (e.g., *uid* $\in$ `[0-9a-zA-Z_-]*` becomes *uid* $=\sim$ / `[0-9a-zA-Z_-]*`/).

Dynamic constraints. A dynamic constraint is essentially a static constraint, which is additionally deemed to be volatile. More precisely, constraints which directly involve the server's environment (e.g., session data, database and file operations) are classified dynamic. Nested constraints are also considered dynamic when present within the scope of a dynamic condition. Because the server's environment may change from the time a form is generated to the time it is submitted (e.g., the set of available user names changes), dynamic constraints can only be checked by consulting the server.

To this end, WAVES generates and makes use of *server-side stubs*, which check dynamic constraints on the server (described in §3). When the client needs to check a form field with a dynamic constraint, it communicates with the server asynchronously. The client-side code for checking dynamic constraints consists of sending requests with form field values to the server and processing status changes from the server's responses into real-time feedback for the user.

Triggering Validation. Once the client-side code is generated, we must instruct the client to execute that code at the appropriate time and inform users when constraints have been violated. For modern web clients, it is usually a simple matter to provide snippets of code to be executed for each of a fixed number of events (e.g., each time the user changes the *uid*). Thus it is a simple matter to tell the client to run the code that checks the appropriate constraints each time a form field changes and provide error messages when appropriate.

Figure 3 gives the JavaScript validation generated by WAVES for the running example. `valUid` is called when `uid` data is supplied and `valPwd` is called when $\texttt{pwd}_2$ is supplied.

3 SERVER-SIDE CODE GENERATION

The main goal in this step is to create server code that responds to an asynchronous client request to check the dynamic constraints for a given form field. That code invokes a stub for each of the dynamic constraints extracted by WAVES. If any of the stubs produces an error, the server returns an error. Stub generation is a three-step process, which we explain below with our running example.

Dependency Analysis. Given a dynamic constraint in the server code, WAVES first performs a data and control dependency analysis to compute the set of all program variables (not just form fields) on which the dynamic constraint depends (either implicitly or explicitly). We call these the *related variables*. We do this via backward analysis, starting from the dynamic constraints and working backwards in the server code. In the running example for the dynamic constraint `userInDB( $userId )`, the set of related program variables includes `$userId` and `$_POST['uid']`.

Program Slicing. WAVES then employs off-the-shelf program slicing techniques [7] to generate the server stubs. More precisely, we begin at the top of the code and prune out any instructions not relevant to the related variables, stopping once we reach the dynamic constraint. The efficiency of the resulting stubs is a direct consequence of how effective our pruning of the server code is. Prune too little, and the stub is inefficient; prune too much, and the stub is unsound. Our pruning process was guided by the following three criteria.

First, the server stub includes all those instructions that the result of the dynamic constraint depends on. All assignments that have a related variable on the left hand side are retained in the server stub. For our running example, this ensures the assignment `$userId = $_POST['uid'];` is not pruned from the stub.

Second, the server stub includes environment variables and functions that affect these variables, such as functions that read or write session values. These functions and variables may indirectly change the control flow of the server code.

Third, some instructions change the state of the server while executing, e.g., inserts and updates in database operations, database schema changes, writing to files, as well as changing and setting session and cookie variables. Including statements with side-effects can lead to inconsistent server state, since the user has not actually submitted the form, but exclud-

```
<script type="text/javascript">
function valUid( ) {
 var uid = document.getElementById('uid').value;
 var {textRE} = /(a-zA-Z0-9_-)*/;
 var bReturn = textRE.match(uid);
 if(!bReturn)
   alert("Error: No special characters in user id.");
 else
   checkExistingUser( uid );
 return bReturn;
}
function valPwd( ) {
 var pass1 = document.getElementById('pwd1').value;
 var pass2 = document.getElementById('pwd2').value;
 var bReturn = pass1 != pass2;
 if(bReturn){
   alert("Error: Passwords don't match.");
 return !bReturn;
}
function checkExistingUser( uid ) {
  xmlhttp.open("GET","unamecheck.php?uid="+uid,true);
  xmlhttp.onreadystatechange = function () {
      ... recvStat(xml http response); };
  xmlhttp.send();
}

function recvStat( uidStatus ) {
  if(uidStatus == "error")
    alert("Error: user id not available.");
}
</script>
```

Figure 3: JavaScript validation for running example.

ing such statements can lead to security vulnerabilities (e.g., an application outfitted to defend against denial-of-service attacks by logging IP addresses and dropping large bursts of requests from a single IP). Thus, WAVES allows a developer to choose whether statements with side-effects are allowed in stubs or not. If side-effects are not allowed, and a stub includes a side-effect after pruning, that stub is eliminated and the dynamic constraint is not checked. Note that failure to check a dynamic constraint is a source of incompleteness, not unsoundness. In addition, none of our test applications (§V) required allowing the use of side effects.

Simplification and Optimization. There are some cases in which constraints on *unrelated* form fields may appear in a server stub. This happens because of control dependencies introduced by if-else constructs in the server code, which will cause unwanted errors. As discussed in Section 1, we can alleviate this problem by using independence information for the form fields.

4 INTEGRATION

WAVES is designed to incorporate client side validation code in new as well as legacy applications. In the previous steps, WAVES generated the code necessary to enable client-side validation of user inputs. The integration of this generated code in an application requires minimal changes to the application's codebase. Installing the server code only requires uploading it to application's directory on the server. Installing the client code is almost as easy—it simply requires augmenting the client's source code to include the JavaScript file containing the generated code. Thus when that file is loaded by the browser, it attaches

all the event handlers to appropriate fields to perform validation.

V EVALUATION

Implementation. The server-side analysis is implemented in Java and Lisp and builds upon our prior work WAPTEC [3] as well as the state-of-the-art SMT solver Kaluza [6]. The client-side code generation is implemented in LISP and Java and builds on Plato [8] (a web form generator), php.js [9] (a library of PHP functions implemented in JavaScript), and the jQuery validation module [10]. The server-side code generation is implemented in Java and builds on Pixy [11] (a tool for PHP dependency analysis).

Test suite. We selected three medium to large and popular PHP applications. Table 1 provides background information on these applications (number of files, effective lines of code – without comments and empty lines, and functionality). The application test suite was deployed on a Mac Mini (1.83 Ghz Intel, 2.0 GB RAM) running the MAMP application suite, and the WAVES prototype was deployed on an Ubuntu virtual machine (2.4 Ghz single core Intel, 2.0 GB RAM).

Experiments. We chose one form in each of the three applications. Two of the chosen forms (`B2Evolution` and `WeBid`) do not contain any client-side validation; the other form (`WebSubRev`) already includes client-side validation. The first two forms allowed end-to-end testing of our prototype tool while the third form allowed us to compare WAVES synthesized code with validation code written manually by developers. We discuss our experiments and experiences below.

1 EFFECTIVENESS

For each of the selected forms, we first manually analyzed the server-side code for processing the chosen form and identified the constraints being checked — we call this the "ideal" synthesis and use it to assess effectiveness of WAVES. For each application, Column 2 of Table 2 shows the ideal number of constraints (static + dynamic). Static constraints, those that do not rely on server-side state, dominated the total number of constraints synthesized by WAVES (27 / 35). As shown in the next column, WAVES was able to synthesize over 83% of the constraints identified by the ideal synthesis.

False Negatives. WAVES suffered from a small number of false negatives due to missed constraints (Column 4 of Table 2). Constraints that WAVES failed to synthesize were those it failed to extract during the server analysis phase. One of the problems encountered was that WAVES generated form field inputs intended to detect whether or not a particular constraint leads to an error, but the form field inputs happened to falsify a different constraint, hence WAVES never inferred the original constraint that caused an error. For example, a constraint in `WeBid` required the e-mail field to include the @ character while another constraint required the e-mail field to satisfy a regular expression. WAVES was unable to uncover the regular expression constraint, because the input used to test if the regular expression constraint was actually an error condition so happened to include no @, therefore, the server rejected due to the first constraint and not the second. We attempted to avoid this problem by generating inputs that satisfy the combination of the two constraints, where one was negated and the other was not, but found that such constraint sets were often too complex for Kaluza to solve efficiently.

The second reason for missing constraints was a fundamental mismatch between the constraints we needed to solve and the language supported by Kaluza. For example, the PHP function `explode` takes a string and splits that string into an array of strings. Since Kaluza does not implement the theory of arrays, we could not encode `explode` into its constraint language, and hence simply ignored any constraint with `explode`. We expect that as SMT solvers that support the theory of strings mature (there have only been two developed to date), many of these issues will be overcome, and the results for WAVES will improve as a consequence. At this point, the important observation is that our basic approach is successful for a majority of the constraints, and there is no fundamental reason those results cannot be improved in practice.

False Positives. Cases where the synthesized client ends up rejecting inputs that the server actually accepts are considered to be false positives (Column 5 of Table 2). In our experiments, we did not encounter any false positives; however, we discuss at least one conceivable case that could cause false positives. When input validation is performed inside a loop, the number of iterations can influence the constraint that gets extracted from a particular trace. For example, the constraint extracted from a loop that iterates over the characters of an input of length

Application	Files	Size(eLoc)	Use
B2Evolution v0.8.6	127	20.4k	Blog
WeBid v0.5.4	403	51.5k	Auction
WebSubRev v0.63	114	12.7k	Conference Mgmt

Table 1: Application testsuite for evaluating WAVES.

Application	Ideal Synthesis	WAVES Synthesis	False Negatives	False Positives	Existing Validation
B2Evolution	10+1	7+1	3	0	0
WeBid	17+8	16+6	3	0	0
WebSubRev	5+1	4+1	1	0	5+0

Table 2: WAVES synthesized over 83% constraints successfully.

n will check exactly n characters each time regardless of the subsequent lengths. In this case, any input whose length is not the same would be rejected by the client. Properly handling this type of validation contained within loops would require assistance from developers in the form of loop invariants. An automatable approach is to discard constraints that are derived from within loops. We would like to note that such a solution would decrease false positives at the expense of increasing false negatives – an advantageous tradeoff which would produce all the benefits of client validation without any impedance of usability. In practice, however, we did not observe inputs being validated in this way but instead by built in library functions, which we handle correctly.

Form Interactivity. One of the benefits from using WAVES is that forms retrofitted with interactivity should improve the overall usability of the application. A synthesized client provides instant feedback as the user interacts with the form. For example, when the user inputs valid data, a green check mark will appear next to the form field; conversely, invalid data will appear next to a red X, and an error message will convey the mistake.

Applications that rely solely on the server to validate form input can be discouraging for the end-user. For example, in the `WeBid` application, we noticed that the server sends a single error message at a time. This particular form contains 25 constraints, so the user may need to resubmit that many times–correcting a single invalid value each time. In addition, the values of 2 password fields within this form are not saved when the user goes back to correct the form, making it necessary to re-enter passwords each time. This problem is eliminated when WAVES introduces validation into the client, because by the time the user submits the form, the values will already be error-free. Our approach helps hide the drawbacks of poorly written web applications that improperly deal with errors.

Improved Performance. The above `WeBid` example also illustrates that insufficient client-side validation can cause repeat submissions, which result in additional server workload and bandwidth use. In the original form submission logic, whenever the user commits an error she needs to retransmit all form data to the server, and the server needs to reprocess the input. Since WAVES effectively offloads validation onto the client, the server spends less resources on form processing, and the overall performance of the application improves. In general, the reduction of resource consumption at the server is expected when most of the constraints are static, but if there are many dynamic constraints, our approach could have the opposite effect. In our experiments, we observed over 75% of form fields have no dynamic constraints; moreover, WAVES allows the developer to choose which form fields to outfit with dynamic constraint checks.

Application	Offline				Online	
	Static Complexity	**Dynamic Complexity**	**Synthesis Time (sec)**	**Avg Stub Size (eLOC)**	**Avg Server RT (ms)**	**Avg Stub RT (ms)**
B2Evolution	52	9	522	27 (26%)	65	43
WeBid	17	18	281	40 (16%)	373	164
WebSubRev	25	5	12921	29 (25%)	633	76

Table 3: Offline and online performance measures

2 SYNTHESIZED CODE VS. DEVELOPER WRITTEN CODE

We also compared the code WAVES synthesized with code written manually by application developers. The third application in our test suite, `WebSubRev`, rejected invalid inputs by employing JavaScript. For this form, the server-side code checked 6 constraints (Column 2 Table 2), and the developer written client-side code checked 5 constraints (all of which were static). WAVES generated 4 static constraints and 1 dynamic constraint, therefore synthesizing 80% of the static constraints and 100% of the dynamic constraints.

The one static constraint that WAVES could not synthesize was a regular expression check on an array obtained from the `explode` function, which as described previously was problematic for Kaluza. The one dynamic constraint discovered by WAVES but not included in the manually written client dictates which filename extensions are accepted by the server. This constraint was not included in the manually written client because (i) the list of permitted extensions is stored in the database and (ii) the constraint is only checked by the server when the administrator has configured the application so that the file field is mandatory. Checking this constraint dynamically can yield a potentially large savings since before a potentially large file is transmitted to the server, the form can warn the user about an improper file type, thereby saving a potentially lengthy wait for the user while the file is transmitted over the network. The server and network also benefit from decreased loads.

3 OTHER EXPERIMENTAL DETAILS

We evaluated WAVES prototype on our test suite and recorded various performance measures during execution (Table 3). In the offline phase, when WAVES performs code analysis, client and server code generation, and installation, we measured the formula complexity of static and dynamic constraints. The second and third columns show static and dynamic formula complexities, which are the total number of boolean operators and atomic constraints. The total time taken by WAVES to extract the formula and synthesize the client is shown by the fourth column. We noted that WAVES spent most time in either analyzing traces or solving constraints. Because WAVES is designed as an offline program transformation tool, even if these numbers are not reduced via additional system engineering, they should be acceptable in many situations. For each dynamic constraint, WAVES synthesized an AJAX stub. As shown in the fifth column, the generated stubs were much smaller in size than the portion of the application relevant to validation – in most cases less than 25% of the original LOC (stub sizes measured in effective Lines of Code using CLOC [12]).

Once WAVES finishes execution and the results are installed, the application is ready for production. The seventh column of Table 3 shows average round trip time taken by stubs in responding to AJAX requests. The round trip time averaged in the range of 43 to 164 milliseconds. For comparison, the sixth column shows the average round trip time taken between client and server when users submit the full form. We believe that in real deployment scenarios such overheads are acceptable as user interactions typically last in the order of a few seconds and will overshadow delays associated with AJAX requests.

VI DISCUSSION

Security Analysis. Since our approach involves generating new code on the server and client, we must consider the security of the augmented web application. Our claim is that WAVES introduces no new security flaws into the application—that the augmented application is as secure (but no more secure) than the original.

It is easy to see that static checks do not change the security of the overall application, unless knowledge of the static constraints enforced by the server is a security vulnerability. It is of course a simple matter to allow the developer to choose which static constraints to enforce on the client. Dynamic constraints are more problematic because they introduce additional entry points (stubs) to the server, which can create security vulnerabilities if not properly protected [13]. Our stub generation includes all authorization checks (cookie validity and other server side checks) performed by the original server, ensuring that the generated stub code is *no weaker* in security than the server code it is derived from. Ultimately, the security of the original server code dictates the security of our approach. While the WAVES augmented application is indeed vulnerable to the same attacks as the original server code, we believe that WAVES decreases the likelihood of security flaws, by altering the web development process to allow the developer to specify all input validation constraints once on the server.

CSRF Protection (CSRF). Token-based CSRF protections involve the server adding a hidden token to each form and ensuring that a valid token is included with each form submission. CSRF defense is an important consideration for WAVES because it is adding code to the server (AJAX stubs) and introducing additional HTTP requests from the client to that server code. WAVES must ensure (i) the AJAX stubs are properly defended against CSRF attacks, (ii) the additional HTTP requests to those stubs include the right CSRF tokens, and (iii) the presence of additional HTTP requests does not change how the server processes the submission of the form's data. Here we distinguish two classes of token-based CSRF protections (proxy-based and app-based) and discuss how WAVES operates in the context of each.

Proxy-based CSRF protections perform both the token-addition and token-validation operations within a proxy that is deployed as a wrapper around a given web application. The proxy provides CSRF protection that is transparent to the underlying application by ensuring that the CSRF tokens are never seen by the application. The application is unaware that the tokens are being inserted and validated by the proxy. Proxy-based CSRF protection and WAVES do not interfere with one another as long as the proxy properly addresses AJAX requests, such as [14]. The fact that WAVES outfitted the original application to include additional JavaScript on the client and AJAX stubs on the server is irrelevant to the proxy; the developer could have added that code herself.

App-based CSRF protections are those where the token-addition and token-validation operations are implemented by the application itself. This means that when WAVES creates new AJAX stubs, those stubs must include the proper token-validation operations and when WAVES generates JavaScript code that makes AJAX calls, that code must include the appropriate token in the call. Here we distinguish app-based protections in terms of whether a token is unique for each session (session-tokens) or for each form (nonce-tokens). Session-specific tokens are far more common and handled properly by WAVES. (i) Each AJAX stub includes all the operations from the original server code that are dependent on server-state; hence, they include token-validation checks. (ii) Each AJAX call includes all the fields on the form (including fields for all the parameters passed in the target URL of the form). As long as the token is embedded directly in the HTML (and not inserted into the submission payload by JavaScript), the AJAX call will therefore include the appropriate token in every request. (iii) Since the token is valid for the duration of the session, using it in AJAX calls before form submission does not change its validity during form submission. In contrast, nonce-tokens expire after a single use; hence, even if the first AJAX request contains the proper token, no subsequent AJAX request will include the proper token, and more importantly the final form submission will not contain a valid token. Of course, a developer will know if the application has app-based, nonce-token CSRF protection and can simply tell WAVES to ignore all dynamic constraints during code generation.

Static vs Dynamic Analysis. WAVES uses dynamic analysis for extracting constraints and static analysis for stub generation. The choice of these different techniques is motivated by the original problems themselves. In the case of constraint extraction, we are interested in extracting each constraint in the server (expressed in terms of server side variables) precisely in terms of form inputs. Dynamic analysis is a good candidate for problems that require such precision. In the case of stub generation, we require that all instructions *ever* needed to check the constraint to be reported by the analysis, suggesting that we need to be *conservative* about the set of instructions that encompass the stub. Static analysis therefore is the natural choice, and the loss of precision in this case only results in slightly worse performance for stub execution (due to additional instructions). Our experi-

mental evaluation shows that the stub code generated for our examples is small, and the performance overheads due to any imprecision are relatively minor.

Other Client Functionality. It is worth noting that WAVES attempts to synthesize only the input validation code for the client—code that is derivable from the server code. We are not proposing to automate the entire client development process. The JavaScript code needed for displaying, organizing, and presenting rich client interfaces (e.g., menus, styles and similar UI elements) still needs to be developed through the usual process. It is worth noting that developing such display elements can be pursued through current development methods as they are tangential to server side development concerns.

VII RELATED WORK

We broadly divide the work related to WAVES into two categories: a) applicable to legacy applications, and b) applicable to newly written code. For each category we discuss the introduction of interactivity and its security implications.

1 LEGACY APPLICATIONS AND INTERACTIVITY

Legacy applications were (and still are) often written by developing the client-side and server-side codebases separately, many times using JavaScript to build interactivity on the client. Separate development of the client and server requires diligence in terms of writing proper server-side validation routines and ensuring that client-side and server-side validation are consistent. When the developer fails in these two tasks, the following two problems arise.

Improper Input Validation. Improper input validation, where the server fails to reject malicious inputs, allows for the possibility of well known security vulnerabilities such as SQL-injection, Cross-site scripting, etc. Many existing works try to reason about missing and/or insufficient validation to detect as well as prevent these problems e.g., [15–20]. The goal of WAVES is orthogonal to these prior works, because it allows the developer to devote the entirety of her input validation development to the server and rest assured that the client validation code will be correct by construction.

Inconsistent Client- and Server-side Validation. Inconsistent client and server validation can lead to problems, such as the parameter tampering vulnerabilities (inputs the client rejects but the server accepts) that our recent work [2,3] established as pervasive in open source and commercial applications. WAVES avoids these inconsistencies for applications where the server validation code is correct by simply replicating that code for the client. Two related works also avoid these inconsistencies but for applications where the *client* validation is correct: Ripley [21] and [22]. These two classes of work are therefore complementary for legacy applications. In terms of techniques, other prior works have investigated analysis that spans multiple modules e.g., [23].

2 NEW APPLICATIONS AND INTERACTIVITY

The key goal of WAVES is to enable developers to write input validation routines once (on the server) and have them replicated elsewhere (on the client). The most germane work, Ripley [21] and [22], could seemingly be used to meet the same objective: write validation code once (on the client) and allow the system to automatically replicate it elsewhere (on the server). However, there is a crucial benefit to writing validation code on the server instead of the client: all constraints, whether static (not dependent on the server's database, file system, etc.) or dynamic (dependent on the server's state) can uniformly be written on the server, but only the static constraints can easily be written on the client. Implementing dynamic constraints on the client requires AJAX and server-side support; thus, dynamic constraints cannot be implemented solely on the client. Furthermore, even if they could be implemented on the client there may be privacy or security reasons to avoid doing so.

Outside the research arena, the most sophisticated tools to aid web development are found within web development frameworks like Ruby on Rails (RoR) [24], Google Web Toolkit (GWT) [25], and Django [26]. Google Web Toolkit allows a programmer to specify which code is common to the client and the server. However, it offers no support for a programmer in the problem of identifying and extracting static or dynamic checks that can be performed by the client. We are only aware of the following two tools that allow a developer to write validation in one place and have it enforced in other places: (a) Ruby on Rails with the SimpleForm plugin [27], and (b) Prado [28]. With RoR, a developer writes the constraints that data should satisfy on the server, and SimpleForm enforces those constraints on the client. The limitation, however, is that the constraints ex-

tracted are limited to a handful of built-in validation routines and are implemented on the client using built-in validation of HTML5. Prado's collection of custom HTML input controls allows a developer to specify required validation at server-side which is also replicated in the client using JavaScript. However, it also allows developers to specify custom validation code for server and client thus introducing avenues for inconsistencies in client and server validation. WAVES, in contrast, extracts any constraints checked by the server and implements them on the client using custom-generated JavaScript code. In the future we plan to investigate how to extend RoR so that developers write either custom Ruby validation code or validation constraints in formal logic, and RoR generates clients that automatically perform the requisite validation.

VIII CONCLUSION

In this paper, we introduced a new methodology for developing client validation code for web applications. Our approach allows the developer to improve security of the web application by focusing only on the server side development of validation. We developed novel techniques for automatic synthesis of the client side validation. Our experimental results are promising: they indicate that automated synthesis can result in highly interactive web applications that are competitive in terms of performance and rival human-generated code in terms of coverage.

ACKNOWLEDGEMENTS

This work was partially supported by National Science Foundation grants CNS-0845894, DGE-1069311, CNS-1065537, CNS- 1141863 and CCF-1018836 and US Air Force Research Laboratory grant FA-8750-12-C-0156.

References

[1] "Outsource JavaScript AJAX Development Services." http://outsourcejavascriptajax.webs.com.

[2] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. Venkatakrishnan, "NoTamper: Automatic Blackbox Detection of Parameter Tampering Opportunities in Web Applications," in *CCS'10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, (Chicago, IL, USA), 2010.

[3] P. Bisht, T. Hinrichs, N. Skrupsky, and V. Venkatakrishnan, "WAPTEC: Whitebox Analysis of Web Applications for Parameter Tampering Exploit Construction," in *CCS'11: Proceedings of the 18th ACM Conference on Computer and Communications Security*, (Chicago, IL, USA), 2011.

[4] R. Wang, S. Chen, X. Wang, and S. Qadeer, "How to Shop for Free Online – Security Analysis of Cashier-as-a-Service Based Web Stores," in *Oakland'11: Proceedings of the 2011 IEEE Symposium on Security and Privacy*, (Oakland, CA, USA), 2011.

[5] M. Alkhalaf, T. Bultan, S. R. Choudhary, M. Fazzini, A. Orso, and C. Kruegel, "ViewPoints: Differential String Analysis for Discovering Client and Server-Side Input Validation Inconsistencies," in *ISSTA'12: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, (Minneapolis, MN, USA), 2012.

[6] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, "A Symbolic Execution Framework for JavaScript," in *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, (Oakland, CA, USA), 2010.

[7] F. Tip, "A survey of program slicing techniques," *Journal of programming languages*, vol. 3, pp. 121–189, 1995.

[8] T. L. Hinrichs, "Plato: A Compiler for Interactive Web Forms," in *PADL'11: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, (Austin, TX, USA), 2011.

[9] "php.js project." `http://phpjs.org/`, 2011.

[10] "jQuery." http://jquery.com.

[11] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities," in *SP'06: Proceedings of the 27th IEEE Symposium on Security and Privacy*, (Oakland, CA, USA), 2006.

[12] "CLOC: Count Lines of Code." http://cloc.sourceforge.net.

[13] A. Guha, S. Krishnamurthi, and T. Jim, "Using Static Analysis for AJAX Intrusion Detection," in *WWW'09: Proceedings of the 18th International Conference on World Wide Web*, (Madrid, Spain), 2009.

[14] R. Pelizzi and R. Sekar, "A server- and browser-transparent CSRF defense for web 2.0 applications," in *Proceedings of the Annual Computer Security Applications Conference*, pp. 257–266, 2011.

[15] P. Saxena, S. Hanna, P. Poosankam, and D. Song, "FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications," in *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, (San Diego, CA, USA), 2010.

[16] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna, "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," in *SP'08: Proceedings of the 29th IEEE Symposium on Security and Privacy*, (Oakland, CA, USA), 2008.

[17] Y. Xie and A. Aiken, "Static Detection of Security Vulnerabilities in Scripting Languages," in *SS'06: Proceedings of the 15th USENIX Security Symposium*, (Vancouver, B.C., Canada), 2006.

[18] Y. Minamide, "Static Approximation of Dynamically Generated Web Pages," in *WWW'05: Proceedings of the 14th International Conference on World Wide Web*, (Chiba, Japan), 2005.

[19] G. Wassermann and Z. Su, "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities," in *PLDI'07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, (San Diego, CA, USA), 2007.

[20] W. Xu, S. Bhatkar, and R. Sekar, "Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks," in

SS'06: Proceedings of the 15th USENIX Security Symposium, (Vancouver, B.C., Canada), 2006.

[21] K. Vikram, A. Prateek, and B. Livshits, "Ripley: Automatically Securing Distributed Web Applications Through Replicated Execution.," in *CCS'09: Proceedings of the 16th Conference on Computer and Communications Security*, (Chicago, IL, USA), 2009.

[22] D. Bethea, R. Cochran, and M. Reiter, "Server-side Verification of Client Behavior in Online Games," in *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, (San Diego, CA, USA), 2010.

[23] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna, "Multi-Module Vulnerability Analysis of Web-based Applications," in *CCS'07: Proceedings of the 14th ACM Conference on Computer and Communications Security*, (Alexandria, VA, USA), 2007.

[24] "Ruby on Rails." `http://rubyonrails.org/`.

[25] "Google Web Toolkit." http://code.google.com/webtoolkit/.

[26] "django: Python Web Framework." `https://www.djangoproject.com/`.

[27] "Simpleform website." `http://blog.plataformatec.com.br/2010/06/simpleform-forms-made-easy/`, 2011.

[28] "Component Framework for PHP5." http://www.pradosoft.com.

IX PROOFS

Definition 1 (Constraint Semantics)**.** *Each constraint over variables X describes a possibly infinite set of variable assignments to X. If C is the constraint, we denote the set of variables appearing in C as $Vars(C)$ and the set of assignments described by C as $VA(C)$. The semantics of a conjunction of constraints (which we also consider a constraint) is defined as usual.*

$$VA(C_1(\bar{x},\bar{y}) \wedge C_2(\bar{x},\bar{z})) = \{\bar{x}/\bar{a},\bar{y}/\bar{b},\bar{z}/\bar{c} \mid \bar{x}/\bar{a},\bar{y}/\bar{b} \in VA(C_1(\bar{x},\bar{y})), \bar{x}/\bar{a},\bar{z}/\bar{c} \in VA(C_2(\bar{x},\bar{z}))\}$$

Definition 2 (Input Semantics)**.** *The input semantics for a form is the (possibly infinite) set of variable assignments permitted by that form. A variable assignment X/A is consistent with the input semantics Δ if there is an extension of X/A that belongs to Δ. A variable assignment X/A is inconsistent if there is no extension of X/A belonging to Δ.*

Definition 3 (Error and Safe Conditions)**.** *A constraint C is an* error condition *for input semantics Δ if every $v \in VA(C)$ is inconsistent with Δ. A constraint C is a* safe condition *for Δ if every $v \in VA(C)$ is consistent with Δ.*

Definition 4 (Success and Failure Traces)**.** *The conjunction of constraints checked on a success trace is a safe condition, and the conjunction of constraints checked on a failure trace is an error condition.*

Definition 5 (Independence)**.** *A set of variables X is independent of the set of variables Y (where X and Y are assumed disjoint) for input semantics Δ if whenever the variable assignment X/A is consistent with Δ and the variable assignment Y/B is consistent with Δ then the assignment $\{X/A, Y/B\}$ is consistent with Δ. We say that a partitioning of variables $X_1 \cup \cdots \cup X_n$ is independent if X_i is independent for X_j for every $i \neq j$. We say a partitioning is strongly independent if X_i is independent of $\bigcup_{j \neq i} X_j$.*

Note that not all independent partitionings are strongly independent. Consider 3 variables x,y,z where Δ is all variable assignments except $\{x/a, y/b, z/c\}$. Then $\{x\},\{y\},\{z\}$ is an independent partitioning because any variable assignment for x,y can be extended to an assignment in Δ; any variable assignment for x,z can be extended; and any variable assignment for y,z can be extended, but $\{x/a, y/b, z/c\}$ cannot be extended to an assignment in Delta.

Theorem 1. *Suppose Δ is the input semantics for a web form. Suppose $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is the conjunction of constraints for some failure trace for that form, where $Vars(D_1) \cup \cdots \cup Vars(D_k) \cup Vars(C_{k+1}) \cup \cdots \cup Vars(C_n)$ is a strongly independent partitioning for Δ and for $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$. Suppose that for each D_i there is some E_i where (i) $E_i \wedge F_1 \wedge \cdots \wedge F_m$ are the constraints checked on a success trace, (ii) $Vars(E_i)$ is independent of the rest of the variables in the conjunction for $VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$, and (iii) $VA(D_i)$ intersects $VA(E_i)$. Then $C_{k+1} \wedge \cdots \wedge C_n$ is an error condition for Δ.*

Proof. Let $X = Vars(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$. Since $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is the conjunction of constraints for a failure trace, $D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is an error condition, ensuring that

each X/A in $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ is inconsistent with Δ.

Consider an assignment $Vars(D_1)/B$ such that $Vars(D_1)/B$ is in the intersection of $VA(D_1)$ and $VA(E_1)$. Since $E_1 \wedge F_1 \wedge \cdots \wedge F_m$ is on a success trace, it is a safe condition, ensuring that each assignment in $VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$ is consistent with Δ. Since $Vars(D_1)/B \in VA(E_i)$ and $Vars(E_i)$ is independent of the variables in $F_1 \wedge \cdots \wedge F_m$, we know that $Vars(D_1)/B \in VA(E_i \wedge F_1 \wedge \cdots \wedge F_m)$ and hence $Vars(D_1)/B$ is consistent with Δ.

By strong independence of $Vars(D_1)$ and $Vars(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$, we know that we can combine $Vars(D_1)/B$ and any assignment $(X - Vars(D1))/C$ in $VA(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ to produce an assignment in $VA(D_1 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$; thus, $\{Vars(D_1)/B, (X - Vars(D_1))/C\}$ must be inconsistent with Δ. By strong independence with respect to Δ, we see that either $Vars(D_1)/B$ or $(X - Vars(D_1))/C$ or both must therefore be inconsistent (since if both were individually consistent, their combination would be consistent. Since $Vars(D_1)/B$ is consistent by construction, we know that $(X - Vars(D_1))/C$ must be inconsistent, i.e., every element of $VA(D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n)$ is inconsistent, and thus $D_2 \wedge \cdots \wedge D_k \wedge C_{k+1} \wedge \cdots \wedge C_n$ is an error condition. Since we chose D_1 arbitrarily, the argument applies to all D_i and hence by straightforward induction we conclude that $C_{k+1} \wedge \cdots \wedge C_n$ is an error condition. $\square$