CAVEAT: Facilitating Interactive and Secure Client-Side Validators for Ruby on Rails applications

Timothy Hinrichs*, Michael Cueno*, Daniel Ruiz*, V.N. Venkatakrishnan* and Lenore Zuck*
*Dept. of Computer Science
University of Illinois at Chicago
Chicago, IL 60607 USA
Email: hinrichs, mcueno2, druiz22, venkat, zuck@uic.edu

***Abstract*—Modern web applications validate user-supplied data in two places: the server (to protect against attacks such as parameter tampering) and the client (to give the user a rich, interactive data-entry experience). However, today's web development frameworks provide little support for ensuring that client- and server-side validation is kept in sync. In this paper, we introduce** CAVEAT[†]**, a tool that automatically creates client-side input validation for Ruby on Rails applications by analyzing server-side validation routines. The effectiveness of** CAVEAT **for new applications is demonstrated by developing three custom apps, and its applicability to existing applications is demonstrated by examining 25 open-source applications.**

Keywords*—*Web applications, Data validation, Frameworks

I. Introduction

Interactive processing and validation of user input are increasingly becoming the de-facto standard for Web applications. With the advent of client-side scripting there has been a rapid transition in recent years to validate user input in the browser itself, before it is submitted to the server. This client-side validation offers numerous advantages, among which are faster response time for users and reduction of load on servers. Yet, client-side validation exposes new vulnerabilities since malicious users can circumvent it and supply invalid data to the server. To defend against these so-called *parameter-tampering* attacks, the server must therefore perform data validation that is at least as strict as that of the client. The practical problem with this situation is that the client and server are often written in different programming languages, thereby requiring the development team to maintain two separate code bases that implement similar but not identical functionality, a notoriously difficult problem.

Several recent studies [1], [2], [3], [4] have uncovered parameter tampering vulnerabilities in both open source and commercial websites, most notably in websites for banking and on-line shopping, as well as those accepting payments through third party cashiers (such as PayPal and AmazonPayments.) These vulnerabilities enable takeovers of accounts and allow a malicious user to perform unauthorized financial transactions.

Ruby on Rails (RoR) has recognized the importance of server-side validation and includes special machinery to make the development and maintenance of server-side data validation especially simple. RoR includes several built-in routines that perform common data validation (e.g., checking that a field is numeric), and a developer only needs to declare which of those routines ought to be applied and to which data elements. For any validations not covered by these built-in routines, the developer extends the validators available in RoR by writing custom code. However, RoR fails to provide machinery that makes client-side validation as easy as server-side validation. Developers wanting to implement both client- and server-side validation must still write and maintain those validation routines in two separate code bases.

One popular extension to RoR, called `client_side_validations` [5], simplifies the creation of client-side validation by analyzing the built-in validations of a RoR application and automatically replicating those validations on the client. Each time a user enters a piece of data on one of these `client_side_validations`-enabled web forms, the validation routines immediately check for errors and signal them to the user, despite the fact that the developer implemented no client-side validation at all. This paradigm of copying server-side validation to the client is advantageous because it incentivizes the developer to spend time and energy perfecting the server-side validation code (which is necessary to protect the server against parameter-tampering attacks), achieves the desired client-side validation, and avoids the problem of manually maintaining two separate code bases with similar functionality. The main limitation of `client_side_validations` is that it fails to move the custom validators written by the developer over to the client.

In this paper, we present the design and implementation of CAVEAT[†], a tool for analyzing and translating custom Ruby on Rails validators from the server to the client automatically. Developers who plan to employ CAVEAT write their custom validators in a fragment of Ruby that admits stronger compile-time analysis than the full Ruby language. The developer also follows several new conventions to guarantee that CAVEAT is sound. The developer can also provide hints to CAVEAT to control which server-side validations to move to the client, thereby leveraging application-specific information known only to the developer.

This paper makes the following contributions:

- Identification of a fragment of RoR custom validation that admits strong compile-time analysis.
- Design and implementation of a tool for replicating server-side validation on the client.
- Conventions for RoR that ensure the tool's soundness.

[†]Compiler For Automated Validation Extraction Analysis and Translation

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-298-1

- A comparison of 25 existing Rails validators against our fragment of RoR validation.
- Demonstration of the tool on 3 RoR applications.

The rest of the paper is organized as follows. After an overview including a running example (Section II), we describe our approach and the challenges of replicating server-side validation on the client (Section III). Next we describe our implementation (Section IV) and discuss our evaluation (Section V). Finally we compare CAVEAT to related work (Section VI) and conclude (Section VII).

II. Background

Ruby on Rails is a web application development framework written in the Ruby programming language. The goal of the framework was to make web programming easier, faster, and more productive. It comes standard with various sane defaults, assumes various conventions that users must also follow, embraces the REST pattern and HTTP verb semantics, highly promotes DRY or ”Don't Repeat Yourself”, and at its core, is fundamentally based on the Model View Controller (MVC) architecture.

The MVC architecture is centered around separating the concerns between the representation of information (the Model), the interaction with that information (the Controller), and its presentation (the View). Models usually wrap an interface around a database table. This interface usually consists of methods that contain the application's business logic. Views are the user interface of the application that, in the context of a web application, get sent to the browser. They are usually written in HTML and some templating code such as embedded Ruby (ERB) or HAML, and contain only functionality needed to present the information. Controllers are what bridge the connection between a View and a Model. They handle the incoming requests, query the models as needed, pass the resulting data on to the view, and return it as the response.

A. Running Example

Many web applications manage user information, e.g., people create new user accounts, provide information like name, email, and address, edit profiles when they become out of date, and delete users along with their profiles from the system. Figure 1 is a diagram depicting the profile information associated with each user. (We use this as a running example that explains our ideas through the paper.)

Every time a new user is created, the application needs to ensure that the profile information meets the following conditions:

- the username has not already been chosen;
- the name must be no longer than 80 characters;.
- the birthdate must be a valid combination of month-day-year;
- the user must be at least 18 years old;
- the two passwords must match;
- the email address must be properly formatted.

These requirements are typical of web applications that require users to create their profiles.

Figure 1: New User Form

B. Example in Rails

Rails has built-in machinery for constructing a significant portion of the web application for creating, retrieving, updating, and deleting (also known as CRUD) the basic objects of the application, e.g., user profiles. It even has built-in machinery for checking that those objects satisfy certain constraints, such as the user being over 18 years old.

In Rails, building CRUD functionality for a User object begins with a definition for the User model, which is given in Listing 1. The developer writes down the fields for the User object (in the line `attr_accessible`) along with statements dictating that those fields must satisfy conditions.

```
class User < ActiveRecord::Base
  attr_accessible :name, :username, :born_on,
    :email, :password, :password_confirmation
end
```

Listing 1: The User Model

The developer also writes Views for displaying the User model to the user (both for creating and editing a User object). The developer also writes Controllers that accept HTTP requests for creating a new user or editing an existing user, validates the User data against the required constraints, and either saves the data to the model or rejects the data and sends the user back error messages. Rails simplifies the process of writing such controllers by automatically validating a User object's data when it is saved to the database, accumulates any error messages that describe why a validation failed, and displays those error messages to the user. For example, Figure 2 shows the response of the Rails app when the submitted data fails the validation. Since the errors are stored within the User model passed to the view, the view can then use that information to display error messages.

C. Rails Validations

Rails includes a variety of built-in validation methods but allows the developer to write custom validators as well. The built-in validators since Rails version 3.0.0 are listed in Table I and can be configured to only be triggered for certain states in

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-298-1

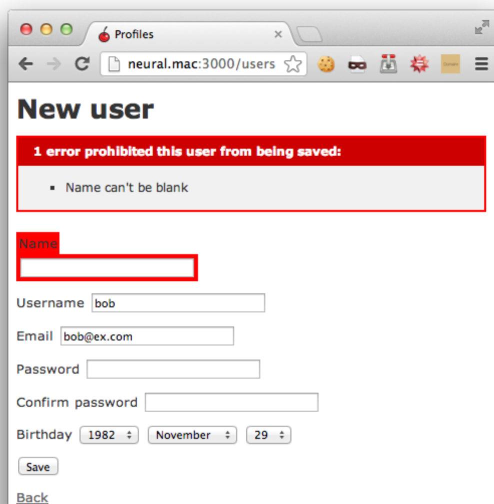

Figure 2: Example of Validation Failure

the record's lifecycle, such as creation, update, or save. Some built-in validators can even be configured to trigger only when a given function returns true.

In our running example, we can use the `presence` validator to ensure that `:name`, `:username`, and `:email` are always provided by the user. A developer would add the following code to the User model.

```
validates :name, :username, :email, presence
    : true
```

Listing 2: Presence Validation on User attributes

While convenient, Rails' built-in validators are not sufficient for every situation and hence Rails allows the developer to write custom validators that are treated similarly to its built-in validators. Just like built-in validators, custom validators are executed each time an object is saved to the database. For example, to ensure that all users are at least 18 years old, a developer could include the code in Listing 3 in the User model.

```
validate :ensure_not_minor

def ensure_not_minor
  unless born_on <= 18.years.ago
    errors.add :born_on, 'Must be at least
        18 years old to use this site.'
  end
end
```

Listing 3: User Model :born_on validation

Rails allows a developer to write validators for individual fields of an object or for the entire object and provides specialized machinery for both cases (`ActiveModel::EachValidator` and `ActiveModel::Validator`, respectively).

III. Approach

CAVEAT transforms a Rails application with server-side validation into a Rails application with both client- and server-side validation. To use CAVEAT, a Rails developer writes her application while obeying a few additional conventions, and CAVEAT automatically analyzes that application, generates JavaScript to implement client-side validation, and installs that JavaScript into the appropriate views. The CAVEAT-instrumented web forms provide the user with real-time feedback and ensure that every form submission passes all the client validations. That is, each time the user edits a form field, the JavaScript runs all the validations pertinent to that field, and alerts the user to any errors; each time the user submits the form, it allows the submission only if there are no outstanding errors. Below we outline the main challenges in transforming web applications in this way as well as the CAVEAT architecture that addresses those challenges.

A. Challenges

Identifying validation code. Web applications can choose how and where to validate input data. In PHP for example, validation code can be interspersed with code for generating HTML, manipulating the database, and changing the session. Ruby on Rails makes the task of identifying which code is related to validation much simpler because developers are expected to write validation code in one of a handful of ways. Thus one of the reasons CAVEAT targets Ruby on Rails is that identifying the code responsible for validating input data only requires understanding the Rails infrastructure, as opposed to performing sophisticated source-code analysis.

Translating server-side validation code into client-side validation. This challenge is by far the most complex. Most obviously, the language the server-side validations are written in, Ruby, is different from the language the client-side validations are written in, JavaScript, and hence there must be translation at the basic level of the programming language. But there is a more fundamental problem in the translation that a Ruby-to-JavaScript compiler would not itself be able to solve: Rails validators are written assuming that the user has completely filled out the web form—that all the data the user will provide has already been provided. In contrast, the client-side validations are often run before the user has completed filling out the form and thus must account for the fact that some data may be unknown at the time the validation is run. Clients that ignore the possibility that some data is unknown can produce surprising and/or unsatisfactory behavior. For example, suppose the server requires values for all of the fields on the form. If the client simply replicated this validation check, then as soon as a user entered a single value in a single field, all the remaining fields would be highlighted as errors.

To provide satisfactory, real-time feedback to the user, the client must be careful to validate only those fields with values. Given a collection of Ruby validations and a Ruby-to-JavaScript translator, the naïve approach to generating client-side validation is to generate one JavaScript validation routine for each Ruby routine and add a guard to each JavaScript validation routine so that it runs only if the form fields mentioned in the validator have values. For forms where the

Table I: Rails built-in validators

Validator	Description	Example
acceptance	Validates that checkbox on UI was checked when form submitted.	`validates :age_requirement, acceptance: true`
validates_associated	Validates model's other associated models.	`has_many :posts` `validates_associated :posts`
confirmation	Validates that two attributes have the exact same values	`validates :password, confirmation: true` `validates :password_confirmation, presence: true`
exclusion	Validates that attribute's value is not part of given set.	`validates :username, exclusion: { in: %w(admin manager user) }`
format	Validates that the attribute's value conforms to given regex.	`validates :email, format: { with: /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,}\z/i }`
inclusion	Validates that the attribute's value is part of a given set.	`validates :tag inclusion: { in: %w( security development rails ) }`
length	Validates that the attribute's value conforms to some length constraint	`validates :username, length: { maximum: 80, minimum:3 }` `validates :password, length: {in: 10...100}`
numericality	Validates that the attribute's value is numerical.	`validates :favorite_number, numericality: true`
presence	Validates that the attribute's value is not blank or nil.	`validates :username, presence: true`
uniqueness	Validates that the attribute's value is unique immediately before being saved.	`validates :email, uniqueness: true`

validation for each field is independent of all the other fields, this approach works; however, if the fields interact this basic approach fails to detect all of the validation failures in real-time. For example, consider a validation for three boolean fields: a, b, c, which the server requires to all have values. In addition, suppose the following validations are written in Ruby.

```
if a and !b then error
if b and !c then error
```

It is straightforward to translate these two validators to the client and add guards that ensure each validator is only run when the fields occurring within it have values. Now suppose the user sets a to True and c to False but sets no value for b. Since b appears in both validators, neither one is executed and the web form signals no error to the user. Yet there is actually already an error between a and c because as soon as either value for b is chosen, one of the validations will fail.

While this naïve approach is sound, it is incomplete, and incomplete validation leads to unsatisfactory clients. For example, if there were 10 a's and 10 c's, but a single b upon which all the a's and c's depended, an unlucky user who happens to set b last might end up with every one of those fields highlighted with an error with no idea how to fix the errors or even how many errors there actually were (at most 10 independent errors). While it is tempting to describe such examples as "corner cases", the heavily-studied problem of configuration management is replete with such examples [6]; moreover, such examples become more frequent as the complexity of validation increases. And the more complex the validation the more a tool like CAVEAT is useful: when writing and maintaining the validation code is hard enough to do just on the server, replicating that code for the client while taking unknowns into account is something a developer may not even attempt.

Plato [7] is a tool designed to build validation routines for the client that are both sound and complete; thus, in the example above, as soon as a is set to True and c to False, Plato's client would signal an error and highlight a and c for the user. The challenge to applying Plato is that its input is a fragment of first-order logic. Thus instead of translating Ruby validations to JavaScript validations, the use of Plato reduces the problem to translating Ruby validations to first-order logic. To address this challenge, we identify a fragment of Ruby that is commonly used and can be translated into first-order logic.

Installing client-side validation. Once we have JavaScript implementations of the server-side validation routines, those functions must be installed on the appropriate web page. In Rails, a developer writes Views that describe how to construct web pages; thus, the JavaScript implementations must be installed into the appropriate Rails Views. To do so, we must be able to track how each View's form fields are handled by Rails controllers when data for those fields is submitted by a user. We must also be able to trace how each controller uses form field data to instantiate Ruby objects and whether the controller runs server-side validations on that data.

Instead of attempting to apply program analysis algorithms to extract the necessary information from the Rails application automatically, we introduce a number of simple Rails conventions that if followed by the developer make installing client-side validation relatively straightforward. We believe that the conventions we propose are ones already followed in many applications.

Server-crucial validations. The discussion so far has focused on transforming server-side validation routines into client-side validation routines. But some server-side validation is conceptually more difficult to move to the client than others. Validation that relies on information the server has but the client does not can be difficult or even ill-advised to move to the client. For example, a user registration page that asks for a login ID, a password, first name, last name, address, etc. may require that the login ID is unique in the backend database. Moving such a validation to the client could be accomplished either by moving the entire list of current user IDs to the client or creating client-validation code that asynchronously communicates with the server (e.g., through AJAX).

To address this challenge we simply recognize which server-side validations can be implemented entirely on the client and which cannot, and then we transform only those validations that can be implemented entirely on the client. In addition to technical convenience, our choice to move just these validations to the client is motivated by a desire to preserve the overall security of the web application. Since many Rails applications are open-source, translating server-side code to the client tells users no more about the application than they could have learned from simply looking at the server code. But if we were to create client-side validations that moved some of the server's information to the client or communicated with the server asynchronously, we could potentially lessen the overall security of the application.

B. Architecture

Conceptually CAVEAT can be broken down into the three components shown in Figure 3, which together address the challenges discussed above: Validation identifier, Validation translator, and Validation installer.

- **Validation identifier**: Takes as input a Rails application and identifies all the server-side validations that the developer has written.
- **Validation translator**: Takes all of the validators from the Validation identifier, eliminates those that cannot be implemented entirely on the client, and generates one JavaScript validation function for each form field f that embodies all the validation that must be performed on the client when f's value changes. When field f's function is executed, it returns a list of form fields that fail to validate because of f's new value. If the list is empty, f's new value caused no validation failures.
- **Validation installer**: Augments each web form View in the Rails application so the resulting web page invokes the JavaScript functions created by the Validation translator. More precisely, the installer attaches JavaScript functions to the form fields' event handlers so that each time a user edits a form field, the validation routines run, and when a validation fails, those form fields that are the cause of the failure are highlighted for the user (e.g., by coloring the offending fields red).

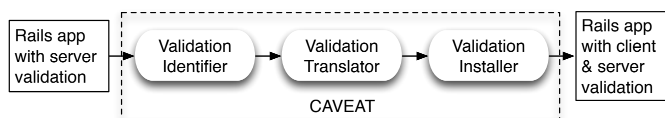

Figure 3: CAVEAT architecture

IV. Implementation

A. Validation Identifier

The first challenge in moving server side validation code to the client is identifying which server-side code is performing input validation. Ruby on Rails was chosen as a deployment target for CAVEAT because it has built-in constructs that make identifying server-side input validation straightforward. Each of Rail's MVC models includes a method called `_validate_callbacks` that returns all of the validation code for that model. The resulting list of validations includes built-in validators, custom validators, and Rails associations. CAVEAT is only interested in the custom validators, which can be identified based on the type hierarchy of Ruby. Thus identifying the validation code in a given Rails model amounts to a few lines of Ruby code.

B. Validation Translator

The Validation Translator takes as input a collection of Ruby validators and outputs JavaScript code that implements client-side validation for the given Ruby validators. The first step is converting each Ruby validator into an abstract syntax tree (AST), a task that CAVEAT carries out using the Ruby gem live_ast. Next CAVEAT converts each Ruby AST into a logical formula that Plato accepts as input or recognizes that the AST is not one that CAVEAT ought to move to the client. Finally, CAVEAT runs Plato once on the set of all logical formulae generated from the ASTs; the resulting JavaScript code implements the client-side data validation code.

The novel part of the Validation Translator is the second step: converting a validator (represented as an AST) into a logical formula or recognizing that the validator ought not be moved to the client. CAVEAT uses a whitebox approach to identifying which validators ought to be moved to the client. CAVEAT was designed to convert a specific fragment of Rails validators to the client; any validator that does not belong to this fragment is ignored. Figure II details the fragment of Ruby that CAVEAT moves to the client, which we call Ruby$_v$. Ruby$_v$ eliminates from full Ruby those language fragments that are perennially problematic for deep, semantic analysis of programming languages, e.g., loops, recursion, reflection, evaluation of dynamically-constructed code fragments, side effects, database operations, network connections.

While simple, we show in our evaluation that a reasonable percentage of custom validators in existing applications can

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-298-1

Table II: Ruby fragment Ruby_v

Construct	Example
Control logic	`if-then-else`, `switch`
Boolean Logic operators	`and`, `not`, `or`
Comparison operators	$<, >, >=$
Basic Arithmetic	$+, -, *, /$
Miscellaneous functions	string manipulation
Method Calls	calls to other custom validators

be expressed in Ruby_v. Furthermore, CAVEAT's ability to address validators written in a more expressive language than Ruby_v is limited by Plato's input language, which is a heavily restricted fragment of first-order logic.

Converting a validator that is expressed in Ruby_v is performed by recursively walking the validator's AST (and the ASTs of any Ruby function called from that validator). If the recursive walk determines that the validator has not been expressed in Ruby_v, it halts the translation. At the top level, the translator is a recursive method with a large switch statement that is conditioned on the root of the AST it is given. It translates each Ruby language construct (e.g., `if`) into a logical version of it (e.g., $\Rightarrow$). The translator also recurses into any other function that is called by a validator.

In addition, the translator recognizes those fragments of Rails code that signal an error. Sometimes an `if` statement is used to check for the presence of errors, while other times an `if` statement is used to check for the lack of errors. Recognizing error-signaling Rails code requires understanding how a model record is validated by Rails. Every model record has an array named `errors` that specifies what validations have failed during validation. By definition, a model record is valid if and only if its `errors` array is empty. Therefore, validation methods that want to block a record from being saved, must insert something into the errors array. For our purposes, we only care whether the validation has returned $true$ or $false$. Thus, CAVEAT treats any statement that adds something to the `errors` array as a statement that signals an error. For the purpose of translation to logic, it suffices to replace any Ruby statement that adds to the `errors` array with $false$. Figure 4 gives pseudo-code for the main portion of the translator.

For example, the following Ruby validator would be represented by the AST shown in Figure 4. The translator would produce the logical formula $age < 16 \Rightarrow false$.

```
if age < 16 then
    errors[:age] << 'You are too young'
end
```

C. Validation Installer

After CAVEAT has generated JavaScript code implementing client-side validation, that JavaScript code must be installed on the client. Recall that each time the user changes a data element on a page, we want the CAVEAT-generated JavaScript to compute errors and for those errors to be communicated to the user via the client's graphical interface. Typical Rails clients already have the ability to display errors by highlighting offending fields in red, thus CAVEAT's main problem in terms of installing the JavaScript code is to ensure that it is invoked appropriately.

The JavaScript code generated by CAVEAT includes one function for each field of the MVC model that CAVEAT was invoked on. Thus each time the user modifies data entry element e, the client must run the CAVEAT code corresponding to the MVC model element e. We do this by leveraging jquery's bind() function to create a callback chain that runs the necessary CAVEAT functions for each field when it has changed value. In order to correctly bind these functions to the form fields we intend them for, we assume developers follow the best practice of labeling form fields using the exact name

Algorithm 1 TRANSLATE(ast, parents)

Input: *ast* is an abstract syntax tree
Input: *parents* is a static version of the function call stack
Returns: First-order logic representation of AST. If the formula includes $\perp$, translation is not possible.

1: **if** ast[0] == :if **then**
2:     **return** TRANSLATE(ast[1]) $\Rightarrow$ TRANSLATE(ast[2])
3: **else if** ast[0] == :and **then**
4:     **return** TRANSLATE(ast[1]) $\wedge$ TRANSLATE(ast[2])
5: **else if** ast[0] == :or **then**
6:     **return** TRANSLATE(ast[1]) $\vee$ TRANSLATE(ast[2])
7: **else if** ast[0] == :not **then**
8:     **return** $\neg$ TRANSLATE(ast[1])
9: **else if** ast[0] == :call **then**
10:     **if** call represents a model variable **then**
11:         **return** ast[2]
12:     **else if** call adds to errors array **then**
13:         **return** $false$
14:     **else if** call represents arithmetic (e.g., in binary) **then**
15:         **return** TRANSLATE(ast[2]) ast[1] TRANSLATE(ast[3])
16:     **else if** call represents a method call **then**
17:         **if** ast[2].instance $\in$ parents **then**
18:             // callee is recursive
19:             **return** $\perp$
20:         **else**
21:             // callee is not recursive, so recurse and translate
22:             **return** TRANSLATE(get_ast(ast[2]))
23:     **else**
24:         **return** $\perp$
25: **else if** ast[0] == :$true$ **then**
26:     **return** $true$
27: **else if** ast[0] == :$false$ **then**
28:     **return** $false$
29: **else**
30:     **return** $\perp$

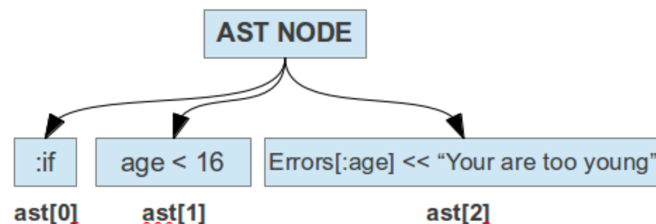

Figure 4: CAVEAT algorithms

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-298-1

of the attribute that the form corresponds to in the Ruby model.

V. Evaluation

We evaluated CAVEAT by (i) writing new Rails applications using CAVEAT and (ii) analyzing the the ease with which CAVEAT can be applied to existing, open-source Rails applications.

A. New Rails Applications

We created three pages with complex validations from different domains: a new user-registration form (the running example), a vacation planning web form, and a checkout form for a shopping cart application.

New user form. Our running example is a typical form that solicits basic information about a new user: name, username, email, password, and birthday. The user must provide information for all fields. The username must be one that has not already been chosen. The name must be no longer than 80 characters. The birthdate must be a valid combination of month-day-year. The user must be at least 18 years old. The password and password confirmation must be the same. The email address must be properly formatted.

When writing the Rails server-side implementation of this form, we used built-in validators for ensuring that all fields have values and that the username is no longer than 80 characters. We used a database query within a validator to check that the provided username is unique. We wrote custom validators to ensure that (i) the birthdate is a proper combination of month-day-year, (ii) the new user is at least 18 years old, and (iii) the two passwords match.

The custom validators that we wrote for birthdate, age, and password were all expressed naturally in Ruby$_v$, and CAVEAT moved those over to the client properly. For example, each time the user enters an invalid month-day-year combination for the birthdate (e.g., February 30) the form highlights the offending fields in red (e.g., month and day). As expected, the remaining validators were ignored by CAVEAT.

Vacation planner form. This example models what a user might see on a travel agency's website when trying to plan a vacation. It has form fields for the budget, number of travelers, number of children, number of adults, vacation package options and departure and return dates. The number of travelers must be the sum of the number of children and the number of adults. The departure date must be before the return date. There are three vacation package options that vary how many nights the vacation lasts; the longer the vacation the higher the budget must be.

When implementing the form processing code in Rails, all of the validators were custom validators, and all of them were naturally written in Ruby$_v$. For example, the budget check was implemented as a `switch` statement on the number of nights to stay, and each case included an `if` checking if the provided budget was sufficient for the chosen number of nights. CAVEAT properly moved all of these validations to the client.

E-Commerce checkout form. This example models a shopping cart checkout form. Here we assume that a subtotal has already been set for an order and that the user has some store credit with the company. The example contains fields for a shipping option, donation, and a payment option (either None or credit card number and expiration date). The donation, payment, and subtotal fields must all have numeric values, and if the payment option is None then subtotal + shipping costs + donation - store credit is less than or equal to zero.

Table III: Breakdown of custom validators without system errors

Custom Validation Method Status	Count	Percent
Written in Ruby$_v$	11	18 %
Could be written in Ruby$_v$	17	28 %
Failed due to loops	7	11 %
Failed due to database interaction	25	41 %
Failed due to networking calls	1	1 %

We used the Rails built-in validators to ensure all fields have values and that the donation, payment, and subtotal fields are numeric. To enforce the condition on the payment option, we wrote a custom validator. That custom validator was naturally expressed in Ruby$_v$, and CAVEAT moved it over to the client properly. As soon as all the fields are given values, the client highlights an error if the payment option None is selected but the outstanding cost is greater than zero.

B. Existing Rails Applications

The second phase of our evaluation examined existing Rails applications. Our goal was to understand the prevalence of Ruby$_v$ validators in the wild. We gathered 25 open source applications from GitHub. We chose these applications based on the number of models that they contained, seeing this as a good indication on how many validations they would use. We also favored applications that were updated regularly and used a current version of Ruby and Ruby on Rails. Three notable applications in our sample were Redmine (a project management tool used regularly in industry), Diaspora (a social networking app) and Spree (a common e-commerce application). For each application, we ran a slightly modified version of CAVEAT that included additional logging capabilities on each of the 68 total custom validators across those 25 applications.

Of the 68 validators, 7 caused systems-level errors during our analysis. Most of the time these errors were due to conflicts with gem versions, specifically with the ast_live gem we use for generating the AST for the methods. Of the remaining validators, 11 validators were already expressed in Ruby$_v$, and 17 could easily have been written in Ruby$_v$. Thus overall, CAVEAT could be readily applied to 28 of 61 validators (46%). Those blocks not expressible in Ruby$_v$ included non-trivial and difficult to analyze loops, database interactions, and networking calls. See Table III for additional details. Although we are only able to support 46% of the analyzed legacy application validation code, new applications that can be written in Ruby$_v$ can be supported at 100%, as illustrated by the examples of the prior subsection.

VI. Related Work

Two of the most germane works, Ripley [8] and [9], could seemingly be used to meet the same objective as CAVEAT:

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 static constraints can easily be written on the client (implementing dynamic constraints requires AJAX and server-side support). Thus, even if all of a client's validation is moved to the server, the developer must write server-side validation.

The other relevant research work is WAVES [10], which automatically extracts server-side validation code from PHP and moves that validation code to the client. The benefit of CAVEAT compared to WAVES is that CAVEAT suffers from fewer analysis mistakes than WAVES. The first mistake WAVES can make is identifying which fragments of server-side code constitutes server-side validation—a mistake CAVEAT does not make because the MVC architecture of Rails facilities identification of validation routines. The second mistake WAVES can make is improperly moving a snippet of PHP validation code to the client—a mistake CAVEAT does not make because we have carefully identified the fragment of Ruby that CAVEAT moves to the client to ensure the resulting client-side code is correct.

Outside the research arena, the most sophisticated tools to aid web development are found within web development frameworks like Ruby on Rails (RoR) [11], Google Web Toolkit (GWT) [12], and Django [13]. Google Web Toolkit allows a programmer to specify which code is common to the client and the server. However, the programmer factors her code into the validation running on just the client, the validation running on just the server, and the validation running on both. Furthermore, GWT fails to provide the same robustness that CAVEAT does in the face of unknown values on the client as discussed in Section III.

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 `client_side_validations` plugin [5], and (b) Prado [14]. With RoR, a developer writes the constraints that data should satisfy on the server, and `client_side_validations` enforces those constraints on the client. The limitation, however, is that the constraints extracted 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. CAVEAT, in contrast, extracts constraints checked by the server and implements them on the client using custom-generated JavaScript code, thereby avoiding the possibility of inconsistency.

VII. CONCLUSION

CAVEAT obviates the need for developers to write and maintain two separate data validation code bases, a notoriously difficult problem in practice, and improves the overall security of the application by allowing developers to focus on the security-critical server-side data validation code. CAVEAT is implemented in Ruby on Rails, an especially attractive deployment target for CAVEAT since server-side validation is built into the framework itself, which makes the identification of server-side validation code much simpler than with other web programming languages and frameworks. CAVEAT was designed to operate on validations expressed in a fragment of the Ruby programming language that admits deep, semantic analysis. We evaluated CAVEAT's effectiveness by constructing three new applications using CAVEAT, and we evaluated its applicability to 25 existing applications by investigating the proportion of validation checks that CAVEAT could replicate on the client.

ACKNOWLEDGMENTS

This research is support in-part by the following grants from the U.S. National Science Foundation: CNS-1141863, DUE-1241685, CNS-0845894, CNS-1065537, DGE-1069311.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] "Client-side validation Ruby gem," https://github.com/bcardarella/client_side_validations, last accessed: 01 Apr 2013.

[6] "CLib: Configuration benchmarks library," http://www.itu.dk/research/cla/externals/clib/, last accessed: 01 Apr 2013.

[7] 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.

[8] 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.

[9] 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.

[10] N. Skrupsky, M. Monshizadeh, P. Bisht, T. L. Hinrichs, V. N. Venkatakrishnan, and L. Zuck, "Waves: Automatic synthesis of client-side validation code for web applications," *ASE Science Journal*, 2013.

[11] "Ruby on Rails," http://rubyonrails.org/Last accessed: 01 Apr 2013.

[12] "Google Web Toolkit," http://code.google.com/webtoolkit, last accessed: 01 Apr 2013.

[13] "django: Python Web Framework," https://www.djangoproject.com/ Last accessed: 01 Apr 2013.

[14] "Component Framework for PHP5," http://www.pradosoft.com, last accessed: 01 Apr 2013.